

POLITECHNIKA LUBELSKA

WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI

INFORMATYKA



Zintegrowany
Program
Rozwoju
Politechniki
Lubelskiej -
część druga

Programowanie aplikacji mobilnych na platformę iOS

W2

Architektura systemu iOS

Maria Skublewska-Paszkowska



Fundusze
Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



Agenda

- Architektura systemu iOS
- Cocoa Touch
- Media Layer
- Core Services
- Core OS

Zintegrowany
Program
Rozwoju
Politechniki
Lubelskiej -
część druga

Architektura systemu iOS

- Architektura jest warstwowa:
 - Cocoa Touch – warstwa aplikacji
 - Media Layer
 - Core Services
 - Core OS
- wyższe warstwy - używane do zaawansowanych technologii
- niższe warstwy – odpowiedzialne za dostarczenie podstawowych technologii i funkcjonalności aplikacji
- każda warstwa – przypisane technologie

Architektura systemu iOS

Cocoa Touch

Frameworks:

*Address Book UI; EventKit UI; GameKit;
iAD; MapKit; Message UI; Notification Center;
PushKit; Twitter; UIKit*

Media

Frameworks:

*Assets Library; AV Foundation; AVKit; Core Audio;
Core AudioKit; Core Graphics; Core Image; Core Text;
Core Video; Game Controller; GLKit; Image I/O;
Media Accessibility; Media Player; Metal; OpenAL;
OpenGL ES; Photos; PhotoUI; Quartz Core;
SceneKit; SpriteKit*

Core Services

Frameworks:

*Accounts; Address Book; Ad Support; CFNetwork;
CloudKit; Core Data; Core Foundation; Core Location;
Core Media; Core Telephony; EventKit; Foundation;
HealthKit; HomeKit; JavaScript Core; Mobile Core; Services;
Multipeer Connectivity; NewsstandKit; PassKit; Quick Look;
Safari; Social; StoreKit; System Configuration; WebKit*

Core OS

Frameworks:

*Accelerate; Core Bluetooth; External Accessory;
Generic Security Services; Local Authentication;
Network Extension; Security; System; 64-Bit Support*

Architektura systemu iOS

- iOS działa jako pośrednik między podstawowym sprzętem, a tworzonymi aplikacjami
- Aplikacje nie komunikują się bezpośrednio z podstawowym sprzętem
- Niższe warstwy zapewniają podstawowe usługi, na których opiera się każda aplikacja, a warstwa wyższego poziomu zapewnia wyrafinowane usługi graficzne i związane z interfejsem

Architektura systemu iOS

- Podczas programowania wyższe warstwy są stosowane przy użyciu obiektowego języka
- Ułatwienie pisania kodu – abstrakcja i enkapsulacja
- Programiści powinni używać niższych warstw tak rzadko, jak to możliwe, głównie po to, aby korzystać z funkcji, które nie są zawarte w wyższych warstwach architektury iOS
- Z każdą warstwą połączone są szkielety programistyczne (ang. framework)
- Zapewniają wszystkie niezbędne interfejsy z klasami, metodami, funkcjami, typami i stałymi

Cocoa Touch

- Najwyższa warstwa zawiera najczęściej używane frameworki do pisania oprogramowania iOS
- Programiści mogą zdefiniować wizualizację aplikacji iOS, a także jej podstawową funkcjonalność
- Stosowana jest w realizacji tak ważnych działań, jak:
 - wielozadaniowość
 - obsługa dotykowa
 - powiadomienia push
 - AirDrop
 - inne

Cocoa Touch

- Szkielety programistyczne:
 - **EvenKit** – przeglądanie i edycja wydarzeń związanych z kalendarzem zainstalowanym w systemie iOS, zapewnia kontrolery widoku dla standardowego interfejsu systemowego
 - **GameKit** - udostępnienie danych związanych z grami online za pomocą Game Center
 - **MapKit** – obsługa mapy, którą można dołączyć do interfejsu użytkownika aplikacji (adnotacje, rysowanie połączeń, trasy)
 - **PushKit** – obsługa rejestracji
 - **iAd** – umożliwia dostarczanie reklam do aplikacji opartych na banerach
 - **Twitter** – obsługuje interfejs użytkownika do generowania tweetów i obsługuje tworzenia adresów URL w celu uzyskania dostępu do usługi Twitter
 - **PushKit** – zapewnia obsługę rejestracji aplikacji VoIP

Cocoa Touch

- Szkielety programistyczne:
 - **UIKit Framework** — zapewnia niezbędną infrastrukturę do stosowania graficznych aplikacji sterowanych zdarzeniami w systemie iOS
 - wsparcie wielozadaniowości (ang. multitasking)
 - podstawowe zarządzanie i infrastruktura aplikacji
 - zarządzanie interfejsem użytkownika
 - wsparcie dla zdarzenia dotyku i ruchu (ang. Touch and Motion)
 - obsługa wycinania, kopiowania i wklejania
 - integracja z iCloud
 - funkcje takie jak: wbudowany aparat, bibliotekę zdjęć, informacje o urządzeniu (nazwa, model), informacje o stanie baterii, informacje o czujnikach i informacje o zdalnym sterowaniu z podłączonych urządzeń (zestawów słuchawkowych)

Media Layer

- Druga warstwa architektury
- Używany, gdy trzeba wdrożyć technologię graficzną, audio lub wideo
- Umożliwia implementację wysokiej jakości aplikacji, które składają się z kilku technologii
- Współpracuje z widokiem UIKit
- Tworzenie standardowego interfejsu i niestandardowego

Media Layer

- Szkielety programistyczne do obsługi grafiki:
 - **UIKit Graphics** – obsługa obrazów, animowanie treści widoków
 - **Core Graphics** — silnik rysowania dla aplikacji na iOS, zapewnia obsługę niestandardowego renderowania wektorowego 2D i opartego na obrazach
 - **Core Animation** — optymalizuje animacje
 - **Core Images** – zaawansowana obsługa wideo i obrazów
 - **OpenGL ES i GLKit** – renderowanie 2D i 3D za pomocą interfejsów z akceleracją sprzętową
 - **Metal** – wysoka wydajność dla zaawansowanych prac związanych z renderowaniem grafiki i obliczeniami

Media Layer

- Szkielety programistyczne do obsługi audio:
 - **Media Player** – platforma wysokiego poziomu, która zapewnia korzystanie z biblioteki iTunes użytkownika i obsługę odtwarzania list odtwarzania
 - **AV Foundation** – interfejs do obsługi nagrywania i odtwarzania audio i wideo
 - **OpenAL** – technologia zapewniająca dźwięk
- Szkielety programistyczne do obsługi wideo:
 - **AV Kit** – zbiór interfejsów do prezentacji wideo
 - **AV Foundation** – zapewnia zaawansowane możliwości odtwarzania i nagrywania wideo
 - **Core Media** – opisuje niskopoziomowe interfejsy i typy danych dla mediów operacyjnych

Core Services

- Zapewnia podstawowe i niezbędne funkcje używane przez wszystkie aplikacje
- Core Foundation i Foundation to dwie podstawowe struktury, które definiują podstawowe typy danych
- Szkielety programistyczne:
 - **Address Book** – dostęp do bazy danych kontaktów użytkownika
 - **Cloud Kit** — zapewnia medium do przenoszenia danych między aplikacją a iCloud
 - **Core Data** — technologia do zarządzania modelem danych aplikacji Model View Controller (MVC)
 - **Core Foundation** — interfejsy, które zapewniają podstawowe funkcje zarządzania danymi i usług dla aplikacji na iOS

Core Services

- Szkielety programistyczne:
 - **Core Location**— udostępnia aplikacjom informacje o lokalizacji i nagłówkach
 - **Core Motion**— dostęp do wszystkich danych opartych na ruchu dostępnych na urządzeniu; korzystając z tej podstawowej struktury ruchu, można uzyskać dostęp do informacji opartych na akcelerometrze
 - **Foundation** – obsługuje: kolekcje typów danych, pakiety, zarządzanie ciągami, zarządzanie datą i godziną, zarządzanie blokami danych surowych, zarządzanie preferencjami, manipulację URL i strumieniem, wątki i pętle itp.
 - **Healthkit** – obsługa informacji zdrowotnych użytkownika
 - **Homekit** – komunikowanie się z podłączonymi urządzeniami i kontrolowania ich
 - **Social** – interfejs dostępu do kont w mediach społecznościowych użytkownika
 - **StoreKit** — obsługa kupowania treści i usług z poziomu aplikacji

Core OS

- Warstwa zapewnia funkcje niskiego poziomu, które są w większości osadzone w szkieletach programistycznych
- Wpierają one głównie bezpieczeństwo i komunikację z zewnętrznymi akcesoriami sprzętowymi
- Tworzenie aplikacji 64-bitowych i umożliwia szybsze działanie aplikacji
- Szkielety programistyczne:
 - **Core Bluetooth** - interakcja z akcesoriami Bluetooth, wspiera: skanowanie w poszukiwaniu akcesoriów Bluetooth, łączenie i rozłączanie ich, nadawanie informacji iBeacon z urządzeń iOS i innych urządzeń.

Core OS

- Szkielety programistyczne:
 - **Accelerate** – wspiera cyfrowe przetwarzanie sygnałów (DSP), algebrę liniową, obliczenia oparte na przetwarzaniu obrazu, zoptymalizowane dla urządzeń z systemem iOS
 - **External Accessory** – komunikacja z akcesoriami sprzętowymi podłączonymi do urządzenia iOS (poprzez Bluetooth, jak i złącze dokujące)
 - **Security Services** – dane bezpieczeństwa, którymi zarządza aplikacja (certyfikaty, klucze publiczne i prywatne oraz zasady zaufania), udostępnia bezpieczne kryptograficznie liczby pseudolosowe
 - **Local Authentication** – Touch ID / Face ID w celu uwierzytelnienia użytkownika i zapewnienia bezpieczeństwa dostępu do aplikacji i jej zawartości

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
INFORMATYKA



Zintegrowany
Program
Rozwoju
Politechniki
Lubelskiej -
część druga

Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego



Fundusze
Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny





Programowanie aplikacji mobilnych na platformę iOS

W12

Wytwarzanie aplikacji z obsługą gestów

Maria Skublewska-Paszkowska



**Fundusze
Europejskie**
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny

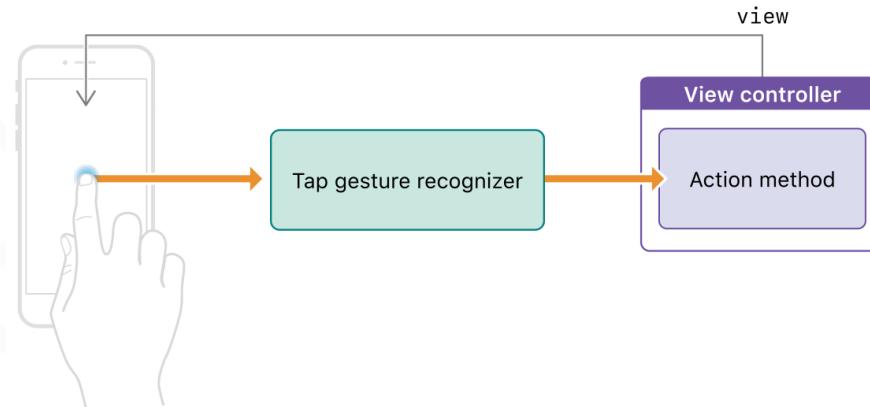


Agenda

- Gesty w iOS
- Obsługa gestów UIKit
- Implementacja gestów UIKit
- Obsługa gestów SwiftUI
- Implementacja gestów SwiftUI

Gesty w iOS

- Manipulacje przez:
 - dotyk ekranu
 - potrząsanie urządzeniem mobilnym
- Bardziej naturalny sposób obsługi urządzenia
- iOS interpretuje gesty i wysyła informacje do aplikacji



źródło:

https://developer.apple.com/documentation/uikit/touches_presses_and_gestures/handling_uikit_gestures

Gesty w iOS

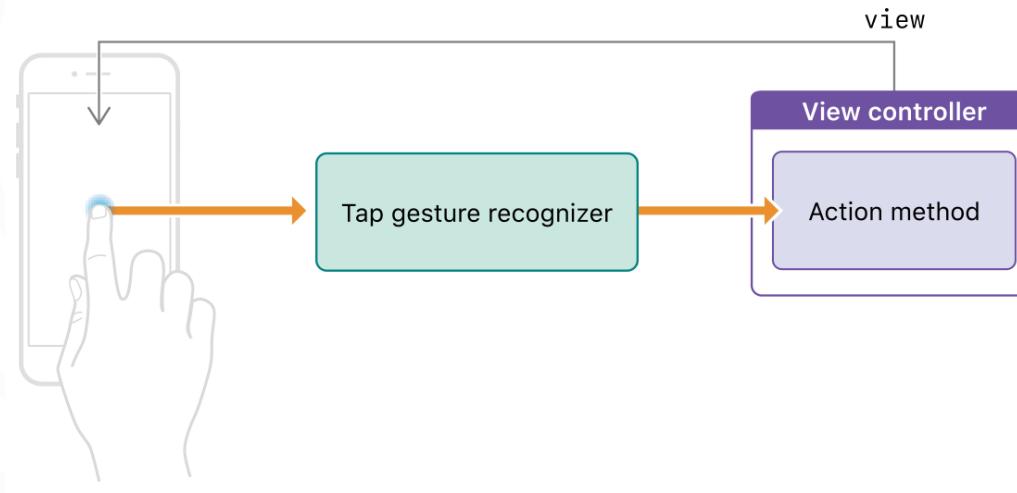
- Zdarzenia (ang. events) – obiekty informujące o działaniach użytkownika
- iOS – zdarzenia przyjmują formy:
 - multidotyk
 - zdarzenia ruchowe
 - zdarzenia do sterowania multimediami (pilot)
- Gesty:
 - tapping
 - pinching
 - panning (dragging)
 - swiping
 - rotating
 - long press

Obsługa gestów UIKit

- Rozpoznawanie gestów:
 - obsługa dotykowa
 - spójne środowisko użytkownika
- Do dowolnego widoku można dołączyć rozpoznawanie gestów (co najmniej jeden)
- Rozpoznawanie gestów hermetyzuje całą logikę niezbędną do przetwarzania i interpretowania przychodzących zdarzeń dla widoku i dopasowuje je do znanego wzorca
- Rozpoznawania gestów dla wykrytego dopasowania powiadamia o przypisanym obiekcie docelowym (np. kontrolerze widoku)

Obsługa gestów UIKit

- Rozpoznawanie gestów używają wzorca projektowego target-action do wysyłania powiadomień:
 - *UITapGestureRecognizer*



źródło:

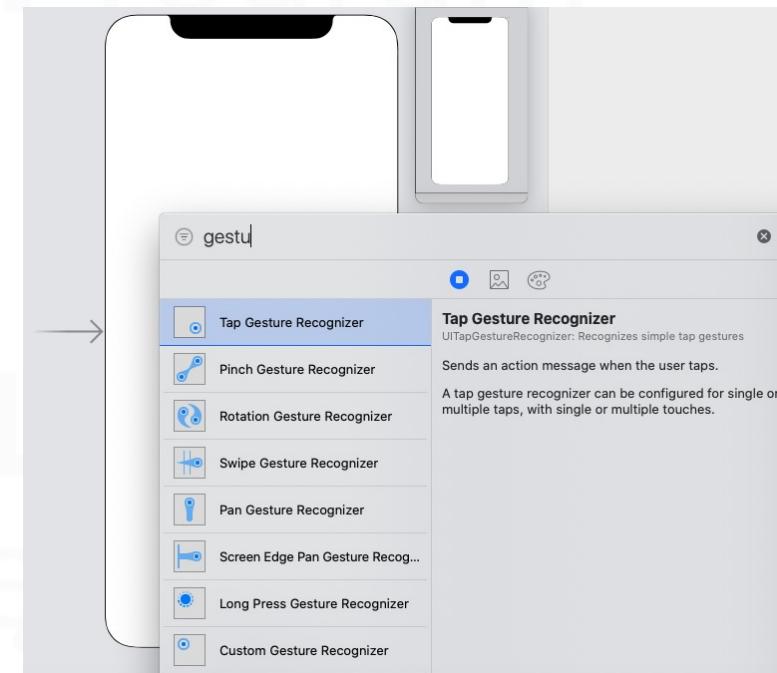
https://developer.apple.com/documentation/uikit/touches_presses_and_gestures/handling_uikit_gestures

Obsługa gestów UIKit

- Rozpoznawanie gestów:
 - dyskretne
 - ciągłe
- Rozpoznawanie gestów dyskretne (np. tap) wywołuje metodę akcji dokładnie raz po rozpoznaniu gestu
- Ciągłe rozpoznawanie gestów (np. rotacja urządzenia) polega na wielokrotnym wywołaniu metody akcji, powiadamiając o każdej zmianie informacji w zdarzeniu gestu
- Jeśli rozpoznawanie gestów rozpozna swój gest, pozostałe dotknięcia widoku zostaną anulowane
- Ścieżka: *CancelsTouchesInView, delaysTouchesBegan, delaysTouchesEnded*

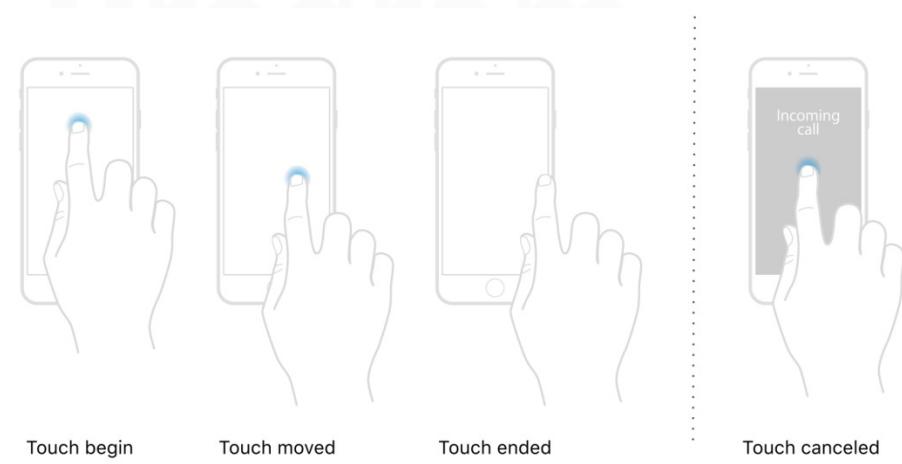
Obsługa gestów UIKit

- Interface Builder zawiera obiekty dla każdego standardowego rozpoznawania gestów
- Interface Builder zawiera niestandardowy obiekt rozpoznawania gestów, którego można użyć do reprezentowania niestandardowych podklas *UIGestureRecognizer*



Obsługa gestów UIKit

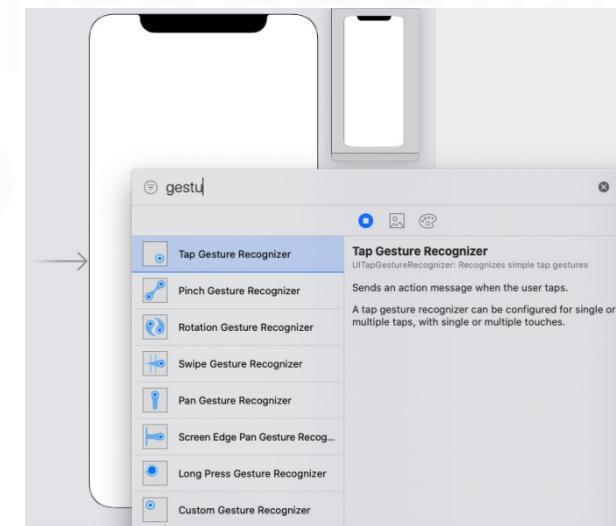
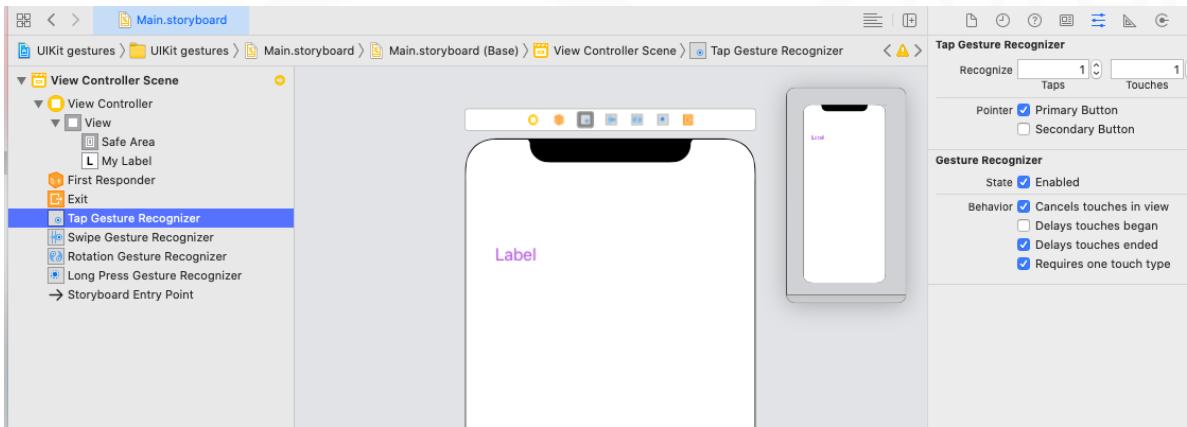
- Obsługa zdarzenia dotykowe bezpośrednio z widoku:
 - widoki są responderami, mogą obsługiwać zdarzenia Multi-Touch i inne
 - UIKit określa, że wystąpiło zdarzenie dotykowe:
 - `touchesBegan(_:with:)`
 - `touchesMoved(_:with:)`
 - `touchesEnded(_:with:)`
 - należy zmodyfikować te metody, aby zapewnić odpowiednie reakcje na zdarzenia dotykowe



źródło:

https://developer.apple.com/documentation/uikit/touches_presses_and_gestures/handling_touches_in_your_view

Implementacja gestów UIKit



Implementacja gestów UIKit

```
@IBOutlet weak var myLabel: UILabel!  
@IBAction func handlingTapGesture(_ sender: Any) {  
        myLabel.text = "Tap Gesture" }  
@IBAction func handlingSwipeGesture(_ sender: Any) {  
        myLabel.text = "Swipe Gesture"}  
@IBAction func handlingRotationGesture(_ sender: Any) {  
        myLabel.text = "Rotation Gesture" }  
@IBAction func handlingLongPressGesture(_ sender: Any) {  
        myLabel.text = "Long press Gesture" }
```

Obsługa gestów SwiftUI

- Rozpoznawanie gestów jest podklassą *UIGestureRecognizer*, dołączoną do *UIView*
- Zarządzanie rozpoznawaniem gestów:
 - *addGestureRecognizer(_ :)*
 - *removeGestureRecognizer(_ :)*
- Jeśli wykryty jest nowy gest, jest on powiązany z rozpoznawaniem gestów dołączonym do danego widoku
- Każde rozpoznawanie gestu posiada swój stan jako zarejestrowane zdarzenie oraz rodzaj gestu
- Jeśli wykryje swój gest, wysyła pojedynczą wiadomość (np. podczas dotyku) lub serię wiadomości (np. podczas przesuwania palcem) – rozróżnienie dyskretnego i ciągłego gestu

Obsługa gestów SwiftUI

- *UIGestureRecognizer* posiada metody:
 - *init(target: action:)* – albo brak parametrów, albo jeden parametr reprezentujący rozpoznawanie gestu
 - *numberOfTouches* – liczba gestów, która jest śledzona
 - *location(ofTouch:in:)* – indeks identyfikujący dotyk (mniejszy niż *numberOfTouches*), drugi parametr dotyczy widoku obsługującego gest
 - *isEnabled* – możliwość wyłączenia gestu bez jego usuwania
 - *allowedTouchTypes* – tablica liczb *NSNumber* reprezentująca wartości rodzajów dotyku w postaci typu wyliczeniowego:
 - *.direct*: – dla palca
 - *.pencil*: - ołówek Apple
 - *.indirectPointer*: - urządzenie wskazujące, np. mysz
 - *.indirect*: - nic z powyższych, np. scroll

Obsługa gestów SwiftUI

- Rozpoznawane gesty:
 - *UITapGestureRecognizer*
 - *UIPinchGestureRecognizer*
 - *UIRotationGestureRecognizer*
 - *UISwipeGestureRecognizer*
 - *UIPanGestureRecognizer*
 - *UIScreenEdgePanGestureRecognizer* – rozpoznaje zdarzenie wykonywane przy krawędzi urządzenia
 - *UILongPressGestureRecognizer*
- Na urządzeniach iPad dodatkowy gest:
 - *UIHoverGestureRecognizer* – wykrywa mysz albo kurSOR wchodzący lub opuszczający widok

Obsługa gestów SwiftUI

- Rozpoznawanie gestów może posiadać delegaty
- Wykonują dwie akcje: blokowanie operacji lub pośredniczą w konfliktach
- Metody pośredniczące w konflikatch
 - `gestureRecognizer(_:shouldReceive:)` – drugim parametrem może być `UITouch` lub `UIEvent`; wysyła informację w pierwszej kolejności do delegata
 - `gestureRecognizerShouldBegin(_:)` – wysyła informację do delegata przed tym jak rozpoznawanie gestów uzyskuje stan `.possible`; metoda zwraca `false` przy wymuszeniu rozpoznananiu gestów do przejścia w stan `.false`

Obsługa gestów SwiftUI

- Metody pośredniczące w konfliktach
 - *gestureRecognizer(_:shouldRecognizeSimultaneouslyWith:)* – wysyłany, gdy rozpoznanie gestu prowadzi do braku wykrycia kolejnego; metoda zwraca true, gdy konflikt zostanie zażegnany
 - *gestureRecognizer(_:shouldRequireFailureOf:)* – pyta delegat, czy rozpoznawanie gestów powinien wymagać innego rozpoznawania gestów, aby akcja zakończyła się niepowodzeniem
 - *gestureRecognizer(_:shouldBeRequiredToFailBy:)* – pyta delegat, czy rozpoznawania gestów powinno wymagać od innego rozpoznawania gestów niepowodzenia

Implementacja gestów SwiftUI

```
struct ContentView: View {  
    @State private var isPressed = false  
    var body: some View {  
        VStack{  
            Text("Gesture recognition")  
            .onTapGesture(count: 2, perform: {  
                print("Double tap")  
            })  
        }  
    }  
}
```

Implementacja gestów SwiftUI

...

```
Image("rose")
    .resizable()
    .scaleEffect(isPressed ? 0.5 : 1.0)
        .animation(.easeInOut)
        .foregroundColor(.green)
    .gesture(
        TapGesture()
            .onEnded({
                self.isPressed.toggle()
            })
    )}}
```

Implementacja gestów SwiftUI



Obsługa gestów SwiftUI

- Tap Gesture

```
struct ContentView: View {  
    @State var tapped: Bool = false  
    @State var fillColor: Color = .blue  
  
    var body: some View {  
        VStack{  
            Text("Kliknij w kwadrat")  
                .padding()  
                .font(.title2)  
            Rectangle()  
                .fill(fillColor)  
                .frame(width: 150, height: 150, alignment: .center)  
        }  
    }  
}
```

Obsługa gestów SwiftUI

- Tap gesture

```
        .frame(width: 150, height: 150, alignment: .center)
  
    .onTapGesture {
        tapped = true
        if fillColor == .red {
            fillColor = .blue
        } else{
            fillColor = .red
        }
        tapped = false
    }
}
}
```

Obsługa gestów SwiftUI

- Utworzenie własnego gestu

```
struct ContentView: View {  
    @State var tapped: Bool = false  
  
    var tappedGest: some Gesture {  
        TapGesture(count: 2)  
            .onEnded{  
                _ in self.tapped = !self.tapped}  
    }  
}
```

Obsługa gestów SwiftUI

- Utworzenie własnego gestu

```
var body: some View {  
    VStack{  
        Text("Kliknij w kwadrat ")  
            .padding()  
            .font(.title2)  
        Rectangle()  
            .fill(self.tapped ? .green : .blue)  
            .frame(width: 150, height: 150, alignment:  
                .center)  
            .gesture(tappedGest)  
    }  
}
```



Obsługa gestów SwiftUI

- *LongPressGesture*

```
struct ContentView: View {  
    var fillColors: [Color] = [.blue, .red, .yellow, .green]  
    @State var color: Color = .cyan  
  
    var body: some View {  
        VStack{  
            Text("Kliknij w kwadrat")  
                .padding()  
                .font(.title2)  
            Rectangle()  
                .fill(self.color)  
                .frame(width: 150, height: 150, alignment:  
            .center)  
    }  
}
```

Obsługa gestów SwiftUI

- *LongPressGesture*

```
.frame(width: 150, height: 150, alignment: .center)  
    .onLongPressGesture(minimumDuration: 3, maximumDistance: 2, perform: {  
        color = fillColors.randomElement()!  
  
    }, onPressingChanged: {_ in  
        self.color = .indigo  
    }  
}  
}
```

Obsługa gestów SwiftUI

- *LongPressGesture – utworzenie gestu*

```
struct ContentView: View {  
    @GestureState var isDetected = false  
    var fillColors: [Color] = [.blue, .red, .yellow, .green]  
    @State var color: Color = .cyan  
var myLongPressgesture: some Gesture{  
    LongPressGesture(minimumDuration: 3, maximumDistance: 2)  
        .updating($isDetected){currentState,gestureState,transaction in  
            gestureState = currentState  
            transaction.animation = Animation.easeIn(duration: 2)  
        }  
        .onEnded{finished in  
            color = fillColors.randomElement()! }  
.onChanged{_ in self.color = .indigo }  
}
```

Obsługa gestów SwiftUI

- *LongPressGesture – utworzenie gestu*

```
var body: some View {  
    VStack{  
        Text("Kliknij w kwadrat")  
            .padding()  
            .font(.title2)  
        Rectangle()  
            .fill(self.color)  
            .frame(width: 150, height: 150, alignment: .center)  
            .gesture(myLongPressgesture)  
    }  
}
```



Obsługa gestów SwiftUI

- *Pinch gesture*

```
struct ContentView: View {  
    @State private var current = 0.0  
    @State private var final = 1.0  
  
    var body: some View {  
        VStack{  
            Text("Zmiana tekstu")  
                .scaleEffect(final+current)  
                .gesture(MagnificationGesture())  
                    .onEnded{ i in  
                        final += current  
                        current = 0 }  
                    .onChanged{ i in  
                        current = i - 1 }  
        )  
    }  
}
```

Obsługa gestów SwiftUI

- *Rotation gesture*

```
struct ContentView: View {  
    ...  
    var body: some View {  
        VStack{  
            Text("Obrót tekstu")  
                .padding()  
                .rotationEffect(final+current)  
                .gesture(RotationGesture()  
                    .onEnded{ angle in  
                        final += current  
                        current = Angle.zero  
                    }  
                    .onChanged{ angle in  
                        current = angle }  
                ) } } }
```



Obsługa gestów SwiftUI

- *Drag gesture*

```
struct ContentView: View {  
    @State var location: CGPoint = CGPoint(x: 70, y: 50)  
    var body: some View {  
        Rectangle()  
            .fill(Color.blue)  
            .frame(width: 100, height: 100, alignment:  
                .center)  
            .position(location)  
        ➔ .gesture(DragGesture())  
            .onChanged{ value in  
                location = value.location  
            }  
    }  
}
```

Obsługa gestów SwiftUI

- *Łączenie gestów*

```
struct ContentView: View {  
  
    var body: some View {  
        VStack{  
            Text("Pierwszy tekst")  
                .onTapGesture {  
                    print("Kliknięto w tekst")  
                }  
            }  
            .onTapGesture {  
                print("Kliknięto poza tekstem")  
            }  
        }  
    }  
}
```



Obsługa gestów SwiftUI

- *Łączenie gestów*

```
struct ContentView: View {  
    var body: some View {  
        VStack{  
            Text("Pierwszy tekst")  
                .onTapGesture {  
                    print("Kliknięto w tekst")  
                }  
        }  
        .simultaneousGesture(TapGesture()  
            .onEnded{_ in  
                print("Kliknięto poza tekstem")  
            } ) } }
```



Obsługa gestów SwiftUI

- Łączenie gestów

```
var body: some View {  
    VStack{  
        Text("Pierwszy tekst")  
            .onTapGesture {  
                print("Kliknięto w tekst")  
            }  
    }  
    .highPriorityGesture(TapGesture()  
        .onEnded{_ in  
            print("Kliknięto poza  
tekstem")  
        } ) } }
```



Obsługa gestów SwiftUI

- Łączenie gestów w jeden

```
struct ContentView: View {  
    @State private var str: String =  
        "Programowanie iOS"  
    var body: some View {  
        VStack{  
            let longPress = LongPressGesture()  
                .onEnded { _ in  
                    str = "Long Press"  
                }  
        }  
    }  
}
```

Obsługa gestów SwiftUI

- Łączenie gestów w jeden

```
let tap = TapGesture(count: 1)
    .onEnded() {
        str = "Tap"
    }

let twoGesture = longPress.sequenced(before:
    tap)

Text(str)
    .font(.title2)
    .gesture(twoGesture)

}
```

Obsługa gestów SwiftUI

- *Swipe Gesture*

```
struct ContentView: View {  
    @State var animals = ["pies", "krokodyl", "wiewiórka", "bóbr"]  
  
    var body: some View {  
        VStack{  
            List{  
                ForEach(animals, id:\.self){ animal in  
                    Text(animal)  
                        .swipeActions(allowsFullSwipe: false){  
                            Button{  
                                let ind: Int = animals.firstIndex(of: animal)!  
                                animals.remove(at: ind)  
                            } label: {  
                                Label("Delete", systemImage: "trash.fill")  
                            }  
                        }  
                }  
            }  
        }  
    }  
}
```

Obsługa gestów SwiftUI

- *Swipe Gesture*

```
Button{  
    let ind: Int = animals.firstIndex(of: animal)!  
    animals[ind] = "wąż"  
} label: {  
    Label("Edit", systemImage: "pencil")  
}  
}  
}  
}  
}  
}  
}  
}
```

Obsługa gestów SwiftUI

- *Shake Gesture*
 - nie jest obsługiwany w SwiftUI
 - w celu implementacji:
 1. dodać rozszerzenie do UIDevice w celu śledzenia informacji o tym geście

```
extension UIDevice{  
    static let info =  
        Notification.Name(rawValue: "ShakeHappened")  
}
```

Obsługa gestów SwiftUI

- *Shake Gesture*

2. dodać rozszerzenie UIWindow, aby nadpisać metodę motionEnded()

```
extension UIWindow{  
    override open func motionEnded(_ motion:  
        UIEvent.EventSubtype, with event: UIEvent?) {  
        NotificationCenter.default.post(name:  
            UIDevice.info, object: nil)  
    }  
}
```

Obsługa gestów SwiftUI

- *Shake Gesture*

3. utworzenie modyfikatora widoku w celu śledzenia gestu

```
struct ShakeGesture: ViewModifier{  
    let action: ()-> Void  
  
    func body(content: Content)-> some View{  
        content  
            .onAppear()  
            .onReceive(NotificationCenter.default.publisher(  
                for: UIDevice.info)){ _ in  
                action()  
            }  
    }  
}
```

Obsługa gestów SwiftUI

- *Shake Gesture*

4. utworzenie rozszerzenia widoku z modyfikatorem

```
extension View{  
    func shakeHappned(perform action: @escaping ()  
        ->Void) -> some View{  
        self.modifier(ShakeGesture(action: action))  
    }  
}
```

Obsługa gestów SwiftUI

- *Shake Gesture*

5. utworzenie widoku

```
struct ContentView: View {  
    @State private var str = "Gest potrąśnięcia"  
    var body: some View {  
        VStack{  
            Text(str)  
                .shakeHappned {  
                    str = "Wykryto"  
                }  
        }  
    }  
}
```

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
INFORMATYKA



Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita Polska

Unia Europejska
Europejski Fundusz Społeczny



POLITECHNIKA LUBELSKA

WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI

INFORMATYKA



Zintegrowany
Program
Rozwoju
Politechniki
Lubelskiej -
część druga

Programowanie aplikacji mobilnych na platformę iOS

W15

Testowanie i debugowanie aplikacji

Maria Skublewska-Paszkowska



**Fundusze
Europejskie**
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



Agenda

- Testy jednostkowe
- Testy jednostkowe w Xcode
- Debugowanie aplikacji

Zintegrowany
Program
Rozwoju
Politechniki
Lubelskiej -
część druga

Testy jednostkowe

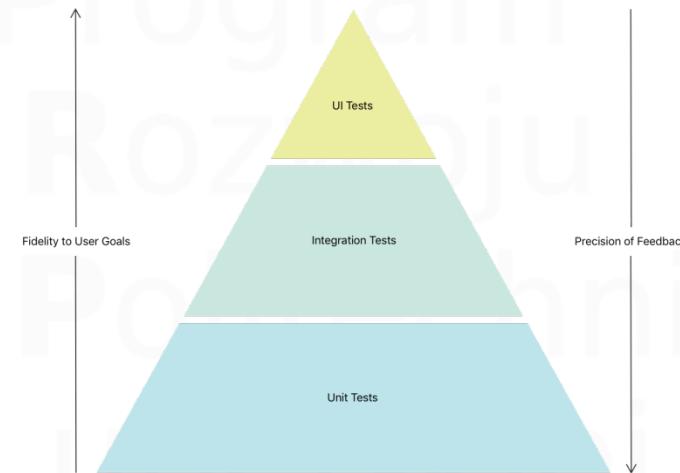
- Testy jednostkowe – metoda testowania tworzonego oprogramowania poprzez wykonywanie testów weryfikujących poprawność działania pojedynczych elementów programu (tj. metod, czy obiektów)
- Testowany fragment programu poddawany jest testowi, który wykonuje go i porównuje wynik (np. zwrócone wartości, stan obiektu, zgłoszone wyjątki) z oczekiwany mi wynikami pozytywnymi i negatywnymi
- Zaletą testów jednostkowych jest możliwość wykonywania na bieżąco w pełni zautomatyzowanych testów na modyfikowanych elementach programu, co pozwala na wychwycenie błędu natychmiast po jego pojawieniu się i szybką jego lokalizację zanim dojdzie do wprowadzenia błędnego fragmentu do programu

Testy jednostkowe

- Należy testować:
 - funkcjonalność aplikacji:
 - klasy
 - metody
 - Interakcje z kontrolerem
 - Przepływ danych w interfejsie użytkownika
 - warunki brzegowe
 - błędy (np. wycieki pamięci)
- Kryteria testowania jednostkowego:
 - szybkie wykonanie
 - testy są izolowane – testy powinny być niezależne
 - powtarzalność – testy powinny dawać takie same wyniki
 - testy powinny być zautomatyzowane (zaliczone lub niezaliczone)
 - terminowo – testy należy pisać przed kodem (Test-Driven Development)

Testy jednostkowe w Xcode

- Xcode zapewnia możliwości testowania oprogramowania
- Framework XCTest – testy jednostkowe dla projektów Xcode, które integrują się z testowaniem
- Testy potwierdzają, że
 - warunki są spełnione podczas wykonywania kodu
 - rejestrowanie niepowodzeń testów (z opcjonalnymi wiadomościami), jeśli te warunki nie są spełnione



źródło:
<https://developer.apple.com/documentation/xcode/testing-your-apps-in-xcode>

Testy jednostkowe w Xcode

- Każdy test jednostkowy powinien potwierdzić oczekiwane zachowanie pojedynczej ścieżki poprzez metodę lub funkcję w projekcie (jeden test dla każdego scenariusza)
 - *rozmieszczenie (arrange)* – należy utworzyć dowolne obiekty lub struktury danych
 - *wywołanie (act)* – należy wywołać testowaną metodę lub funkcję, używając parametrów i właściwości skonfigurowanych w fazie arrange
 - *zapewnienie (assert)* – *Test Assertions* do porównania zachowanie kodu (faza act) z oczekiwaniemi wynikami

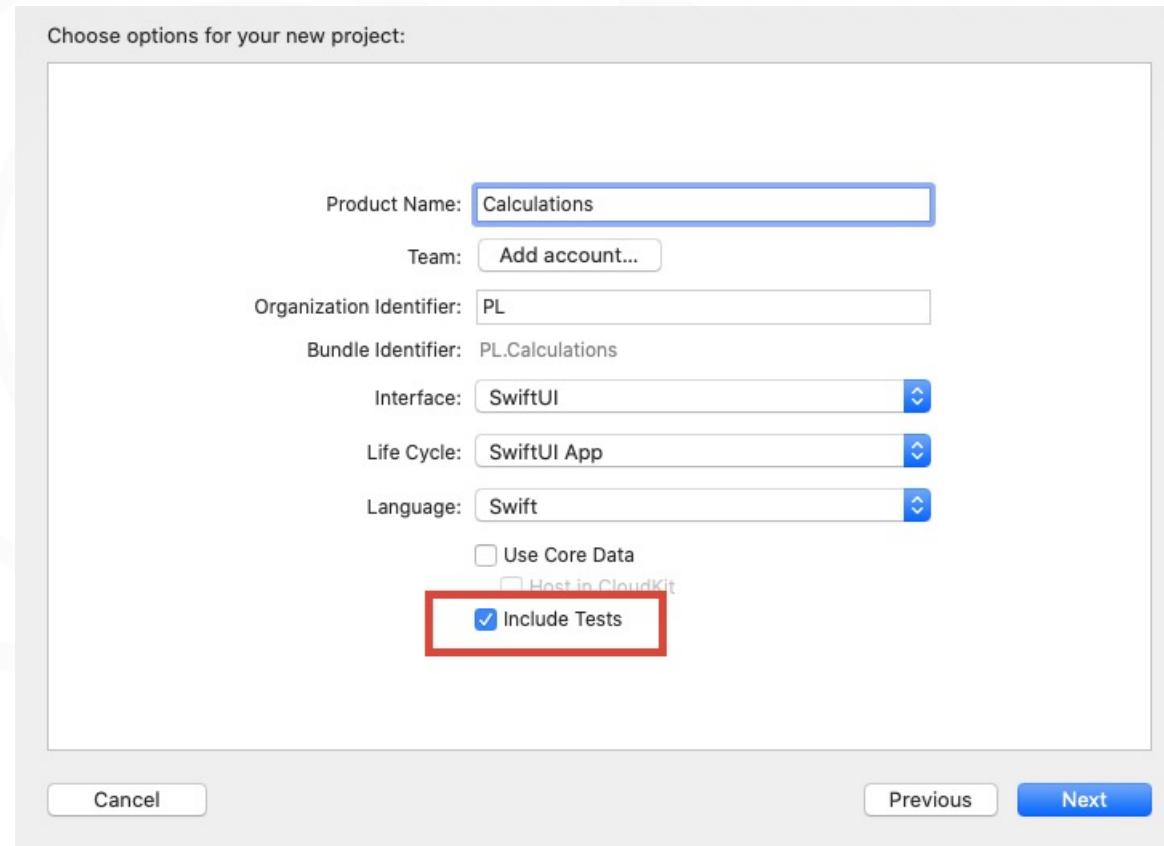
Testy jednostkowe w Xcode

- Testy integracyjne -- podobne do testów jednostkowych
 - wzór *Arrange-Act-Assert*
 - sprawdzają zachowanie większego podsystemu lub kombinacji klas i funkcji
 - testują, jak komponenty współpracują ze sobą, aby osiągnąć cele aplikacji w ważnych sytuacjach
 - obejmują testowanie, czy wartość otrzymana z kontrolera jest prawidłowo przechowywana w modelu oraz czy błąd generowany przez żądanie sieciowe jest przekazywany do interfejsu użytkownika i przez niego prezentowany

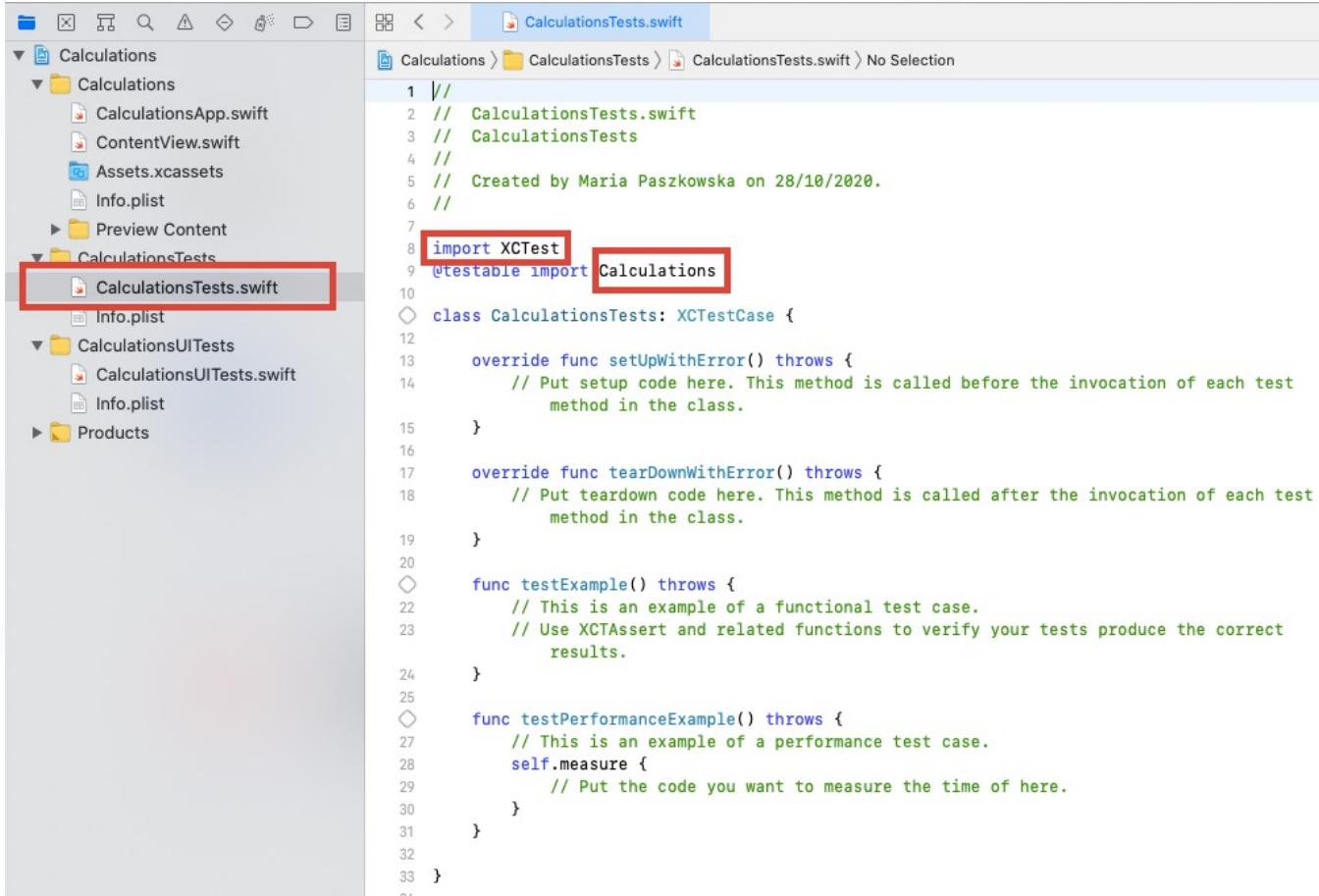
Testy jednostkowe w Xcode

- Testy UI – metody podklas XCTestCase
 - używa elementów sterujących interfejsu użytkownika aplikacji tak, jak zrobiłby to prawdziwy użytkownik, aby określić, czy użytkownik może wykonać określone zadanie za pomocą aplikacji
 - sprawdza, czy w aplikacji można wykonać ważne zadania użytkownika i czy nie zostały wprowadzone błędy, które zakłócają działanie kontrolek interfejsu użytkownika
 - replikowanie rzeczywistych działań użytkownika daje pewność, że aplikacja może być używana zgodnie z zamierzonym zadaniem

Testy jednostkowe w Xcode



Testy jednostkowe w Xcode

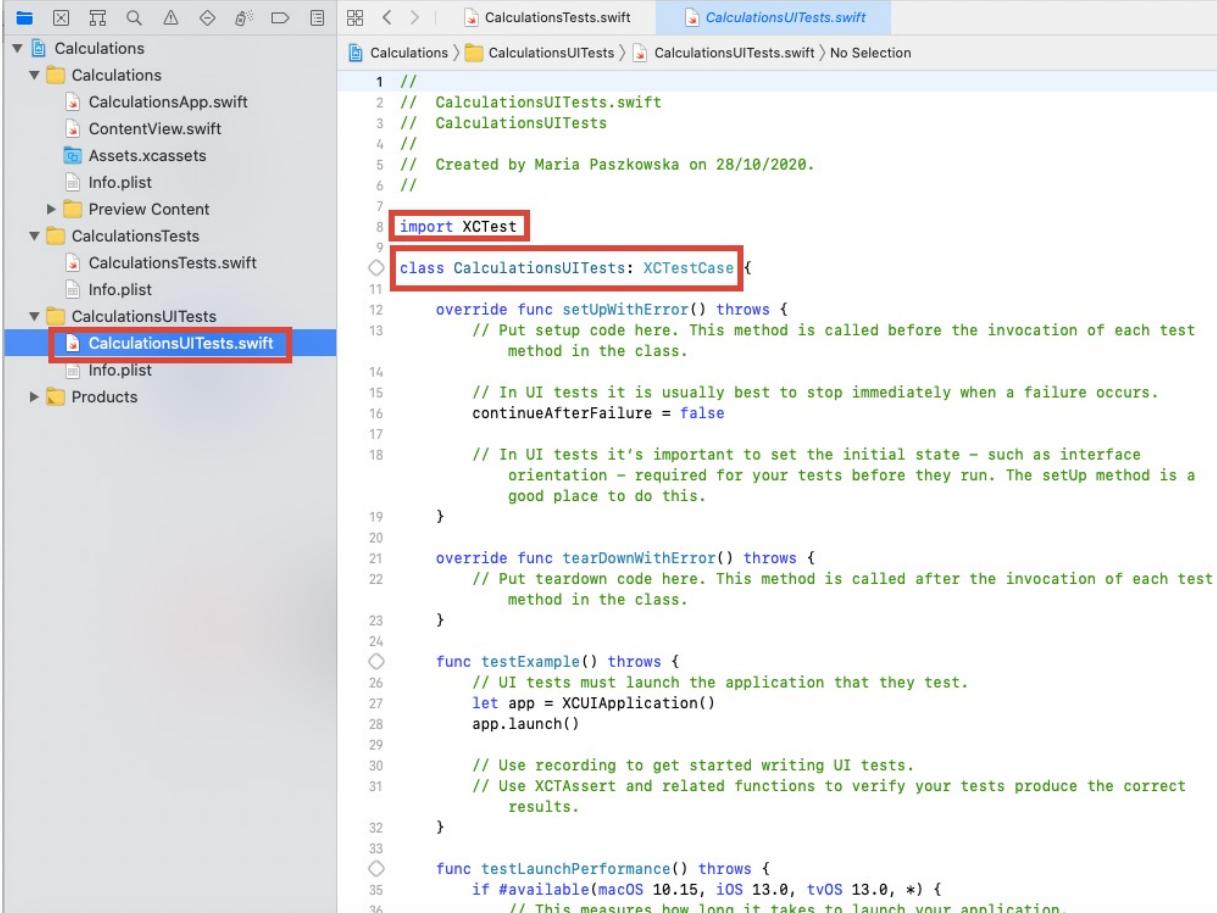


The screenshot shows the Xcode interface with the following details:

- Project Navigator:** Shows the project structure under the "Calculations" target:
 - Calculations:** CalculationsApp.swift, ContentView.swift, Assets.xcassets, Info.plist, Preview Content.
 - CalculationsTests:** CalculationsTests.swift (highlighted with a red box).
 - CalculationsUITests:** CalculationsUITests.swift, Info.plist.
 - Products:**
- Editor:** Displays the content of the `CalculationsTests.swift` file.

```
1 //  
2 // CalculationsTests.swift  
3 // CalculationsTests  
4 //  
5 // Created by Maria Paszkowska on 28/10/2020.  
6 //  
7  
8 import XCTest  
9 @testable import Calculations  
10  
11 class CalculationsTests: XCTestCase {  
12  
13     override func setUpWithError() throws {  
14         // Put setup code here. This method is called before the invocation of each test  
15         // method in the class.  
16     }  
17  
18     override func tearDownWithError() throws {  
19         // Put teardown code here. This method is called after the invocation of each test  
20         // method in the class.  
21     }  
22  
23     func testExample() throws {  
24         // This is an example of a functional test case.  
25         // Use XCTAssertEqual and related functions to verify your tests produce the correct  
26         // results.  
27     }  
28  
29     func testPerformanceExample() throws {  
30         // This is an example of a performance test case.  
31         self.measure {  
32             // Put the code you want to measure the time of here.  
33         }  
34     }  
35 }
```

Testy jednostkowe w Xcode



The screenshot shows the Xcode interface with the project navigation on the left and the code editor on the right. The project structure is as follows:

- Calculations** folder:
 - Calculations** folder:
 - CalculationsApp.swift
 - ContentView.swift
 - Assets.xcassets
 - Info.plist
 - Preview Content** folder
 - CalculationsTests** folder:
 - CalculationsTests.swift
 - Info.plist
 - CalculationsUITests** folder:
 - CalculationsUITests.swift** (highlighted with a red box)
 - Info.plist
 - Products** folder

Testy jednostkowe w Xcode

- Wykonywanie testów

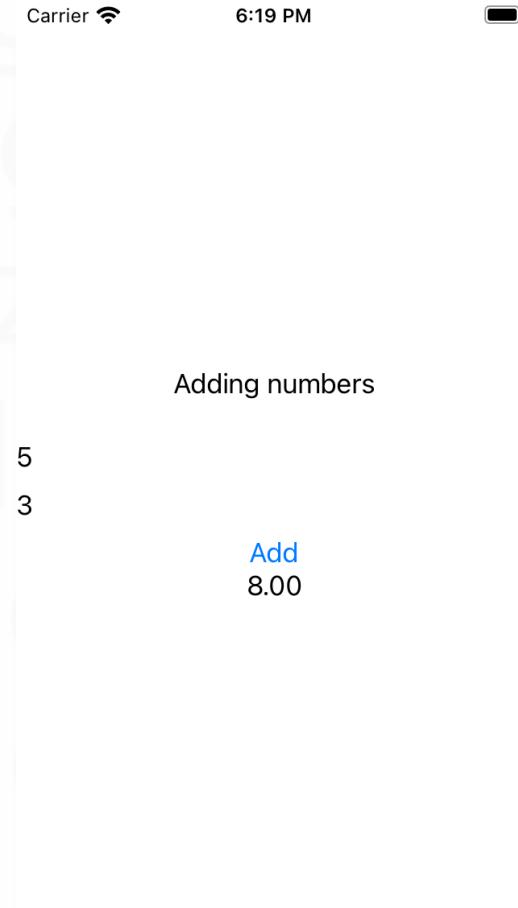
- *Product -> Test (Command-U)*
- Wybranie przycisku *(View->Navigators->Tests)*

The screenshot shows the Xcode interface with the following components:

- Left Sidebar:** Shows the project structure with two test targets: "CalculationsTests" and "CalculationsUITests". Each target contains one or more test cases.
- Code Editor:** Displays the source code for the "CalculationsTests.swift" file. It includes standard XCTest setup methods like `setUpWithError()` and `tearDownWithError()`, and two test functions: `testExample()` and `testPerformanceExample()`. The `testPerformanceExample()` function uses `self.measure` to time a specific calculation.
- Bottom Navigator:** Shows the output of the last run, indicating the start time, setup duration, and time taken for the calculation.

Testy jednostkowe w Xcode

```
Button(action: {  
    clicked = true  
    let op = Operations(numa: self.numa, numb:  
        self.numb)  
    self.addRes = op.addNumbers(a:  
        op.numa, b: op.numb)  
}) {  
    Text("Add")  
}  
if clicked == true{  
    Text("\(String(format: "%.2f",  
        self.addRes))")  
}  
}
```



Testy jednostkowe w Xcode

```
public class Operations{  
    public var numa : Double  
    public var numb : Double  
    init (numa : Double, numb :  
          Double) {  
        self.numa = numa  
        self.numb = numb  
    }  
    public func addNumbers  
        (a : Double, b : Double)  
        -> Double {  
        return a + b  
    }  
}
```

→ **func testAddingTwoValues() {**

```
    let op =  
        Operations(numa:  
                    5.0, numb: 3.0)  
  
    let res = op.addNumbers(a:  
                           op.numa, b: op.numb)  
  
    → XCTAssertEqual(res, 8.0)  
}
```

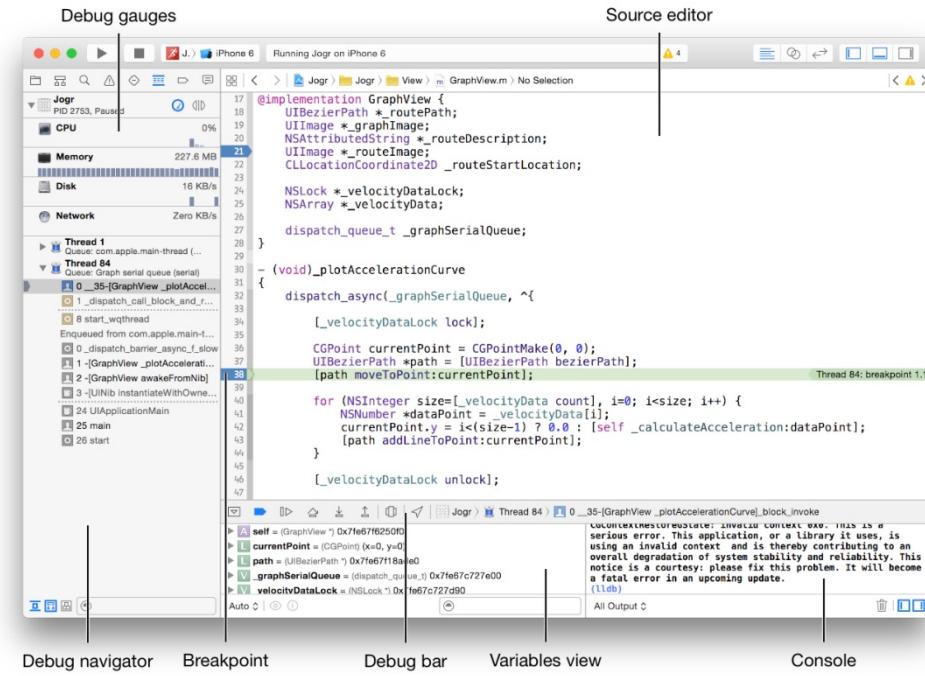
Testy jednostkowe w Xcode

```
7  
8 import XCTest  
9 @testable import Adding  
10  
11 class AddingTests: XCTestCase {  
12  
13     func testAddingTwoValues() {  
14         let op = Operations(numa: 5.0, numb: 3.0)  
15         let res = op.addNumbers(a: op.numa, b: op.numb)  
16         XCTAssertEqual(res, 8.0)  
17     }  
18 }  
19
```

```
Test Suite 'Selected tests' passed at  
2020-10-31 18:13:39.649.  
Executed 1 test, with 0 failures (0  
unexpected) in 0.002 (0.006)  
seconds
```

Debugowanie aplikacji

- Narzędzia do debugowania są zintegrowane w całym Xcode
- Interfejs debugowania jest dynamiczny – rekonfiguruje się podczas budowania i uruchamiania aplikacji
- Dostosowanie sposobu wyświetlania części interfejsu użytkownika przez Xcode: *Xcode Preferences > Behaviors*



źródło:

https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/debugging_with_xcode/chapters/debugging_tools.html

Debugowanie aplikacji

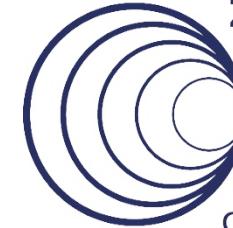
The screenshot shows the Xcode interface during a debugging session. On the left, the 'Adding' process (PID 6335) is monitored, showing minimal resource usage (CPU 0%, Memory 8 MB, Disk 1,2 MB/s, Network Zero KB/s). Thread 1 Queue (serial) is active, with a breakpoint at line 35: `if clicked == true {`. The code editor displays the `ContentView.swift` file, specifically the `body` property implementation. The line with the breakpoint is highlighted in green, and a tooltip indicates it is a 'Thread 1: breakpoint 2.1 (1)'. The bottom status bar shows the current thread is Thread 1, and the stack shows `self (Adding.ContentView)`.

```
Adding PID 6335
CPU 0%
Memory 8 MB
Disk 1,2 MB/s
Network Zero KB/s

Adding > Adding > ContentView.swift > body

11
12 struct ContentView: View {
13     @State var numa : Double = 0.0
14     @State var numb : Double = 0.0
15     @State var addRes : Double = 0.0
16     @State var clicked : Bool = false
17
18     var body: some View {
19         Text("Adding numbers")
20             .padding()
21
22         TextField("Input first value", value: $numa, formatter:
23             NumberFormatter())
24         TextField("Input second value", value: $numb, formatter:
25             NumberFormatter())
26
27         Button(action: {
28
29             clicked = true
30             let op = Operations(numa: self.numa, numb: self.numb)
31             self.addRes = op.addNumbers(a: op.numa, b: op.numb)
32
33             print("\(self.addRes)")
34         }) {
35             Text("Add")
36         }
37     }
38 }
39
40
41 struct ContentView_Previews: PreviewProvider {
```

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
INFORMATYKA



Zintegrowany
Program
Rozwoju
Politechniki
Lubelskiej -
część druga

Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego



Fundusze
Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



Programowanie aplikacji mobilnych na platformę iOS

W14

Lekka baza danych, zarządzanie danymi

Maria Skublewska-Paszkowska



Fundusze
Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



Agenda

- Wprowadzenie
- Lekka baza danych
- Core Data
- Zarządzanie danymi

Zintegrowany
Program
Rozwoju
Politechniki
Lubelskiej -
część druga

Wprowadzenie

- Często aplikacje wymagają przechowywania, wyszukiwania i filtrowania danych
- Przechowywanie danych w bazie danych pozwala na zarządzanie nimi
- Wybór najbardziej odpowiedniej bazy danych jest znaczący:
 - aplikacja powinna być łatwa do ponownego zaimplementowania (logika trwałości danych)
 - może zależeć od złożoności i ilości danych do przetworzenia
- Niepoprawnie dobrana baza może spowodować problem z ponownym zaimplementowaniem logiki trwałości danych aplikacji, co może zwiększyć koszty rozwoju

Lekka baza danych

- SQLite:
 - popularny system zarządzania bazą danych
 - dane przechowywane są w tabelach
 - nie wymaga żadnej konfiguracji ani serwera
 - stworzony w 2000 roku
 - darmowe i otwarte oprogramowanie
 - nie wykorzystuje architektury klient-serwer
 - umożliwia przechowywanie aplikacji na urządzeniu mobilnym
 - bardzo lekka i wydajna (wydajność – kluczowy czynnik)
 - prosty w konfiguracji i obsłudze (bez dodatkowej konfiguracji)
 - zawiera wbudowany silnik SQL, dzięki czemu można zastosować prawie polecenia SQL

Lekka baza danych

- SQLite:
 - działa jako część aplikacji (nie wymaga żadnych usług)
 - jest niezawodny
 - jest szybki
 - jest w pełni obsługiwany przez Apple, ponieważ jest używany zarówno w iOS, jak i Mac OS
 - posiada ciągłe wsparcie programistów
 - jest przenośny, co oznacza, że jest kompatybilny z praktycznie każdą platformą, od Windows przez macOS, od Linux do Androida i oczywiście iOS.
 - może być obsługiwany przy użyciu różnych języków programowania, w tym C++, C#, JavaScript, Objective-C, Swift, PHP, Ruby, Java, czy Python

Core Data

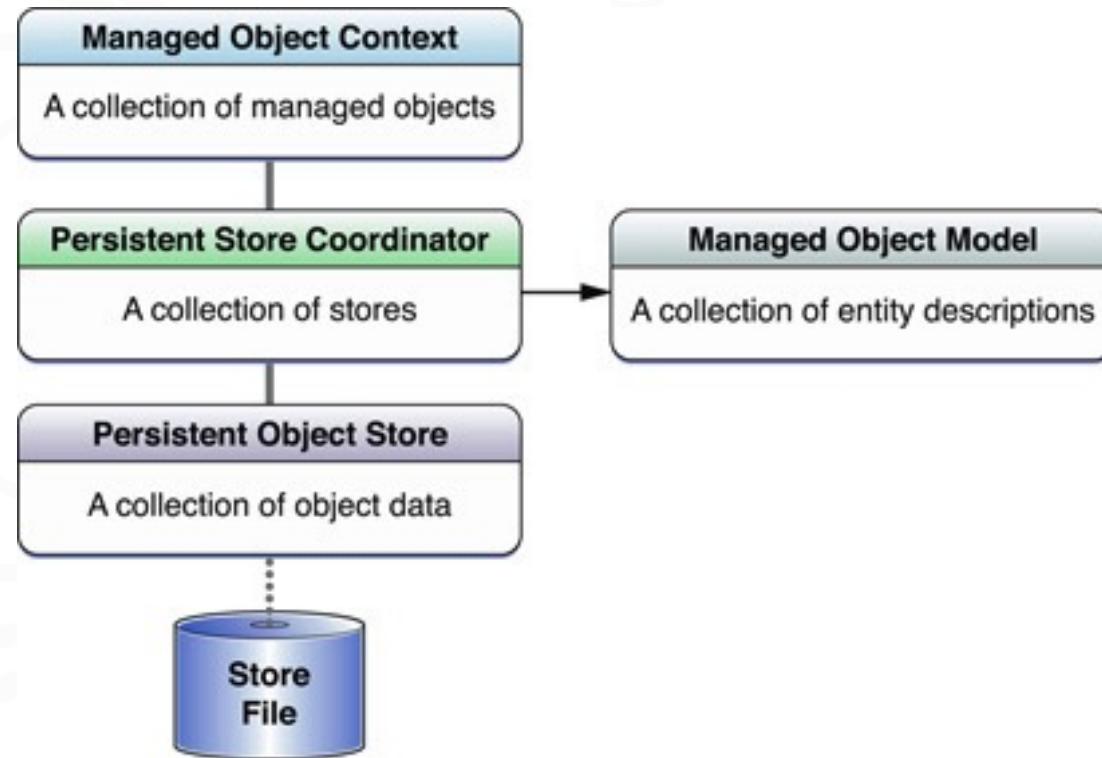
- Core Data:
 - nie jest bazą danych
 - framework do zarządzania grafem obiektów
 - zapewnia przechowywanie danych i ich zarządzanie
 - domyślnie używa bazy danych SQLite jako magazynu
 - można także użyć dowolnej bazy SQL
 - posiada wiele dodatkowych funkcji (np. wersjonowanie danych, automatyczna walidacja, cofanie/ponawianie)
 - przechowuje dane na jednym urządzeniu lub synchronizuj te dane z wieloma urządzeniami za pomocą CloudKit
 - jest zorganizowany w postaci encji oraz grafu obiektu

Core Data

- Właściwości Core Data:
 - operacje cofania i ponawiania, z wyjątkiem podstawowej edycji tekstu
 - relacje pomiędzy obiektami
 - zmniejszenie pamięci przypisanej do aplikacji
 - automatyczna walidacja (np. zakresy dla wartości)
 - migracja schematu
 - synchronizacja GUI
 - wsparcie dla kodowania i obsługi klucz-wartość
 - obsługa przechowywania danych w repozytoriach
 - tworzenie i wykonywanie zapytań
 - zasady łączenia danych

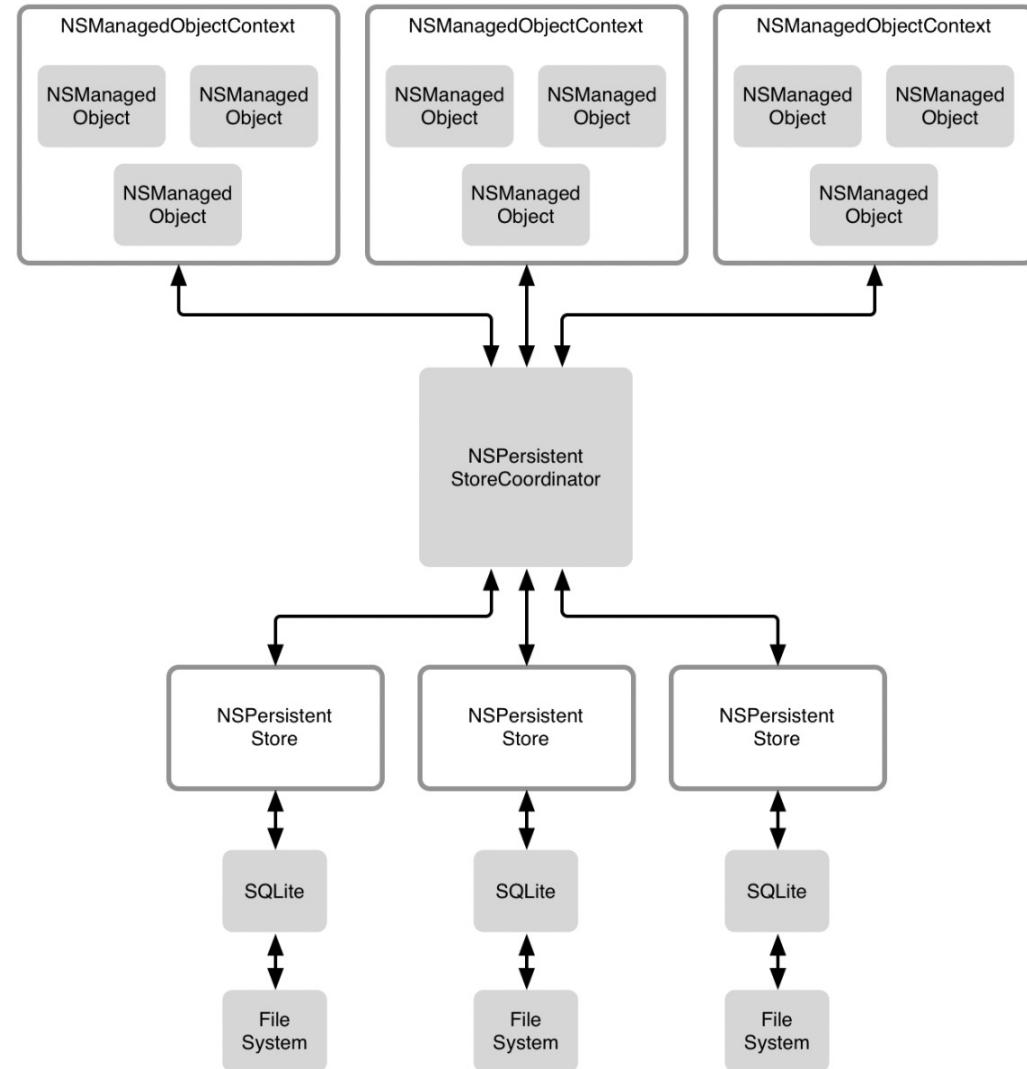
Core Data

- Architektura



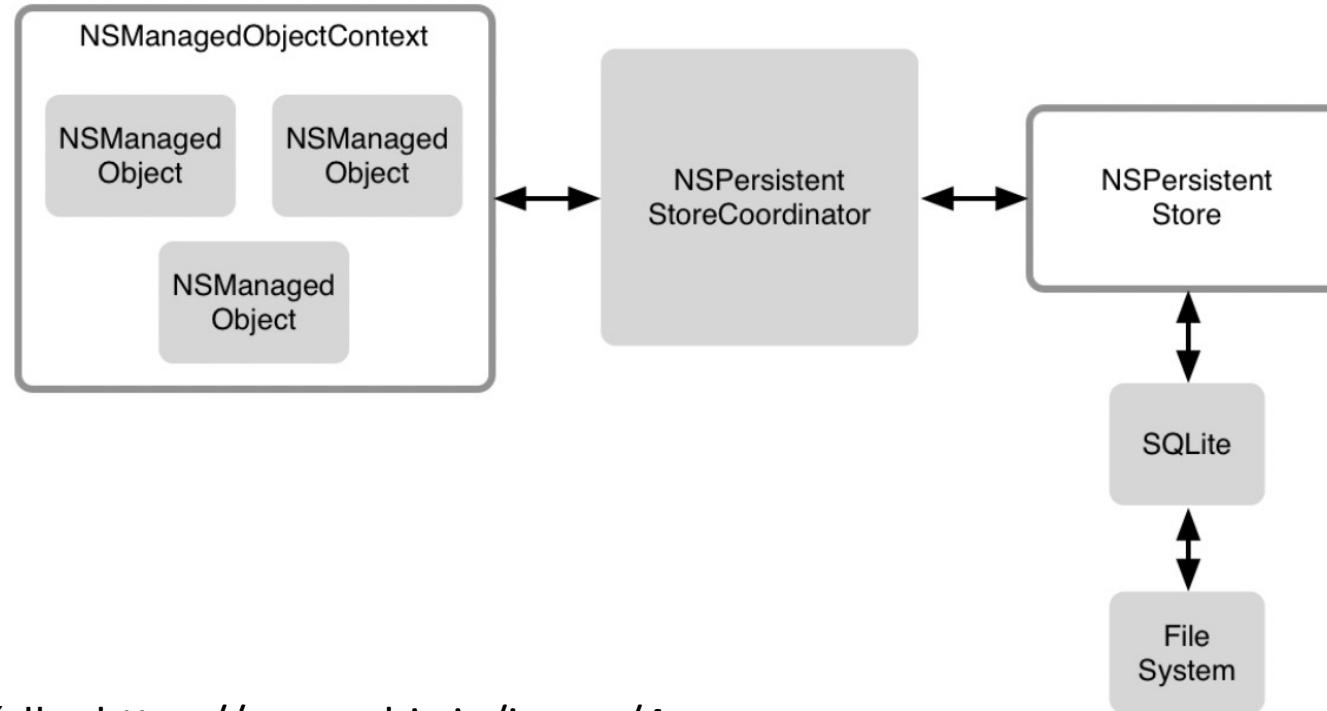
źródło:<https://developer.apple.com/library/archive/documentation/DataManagement/Devpedia-CoreData/coreDataStack.html>

Core Data



źródło: <https://www.objc.io/issues/4-core-data/core-data-overview/>

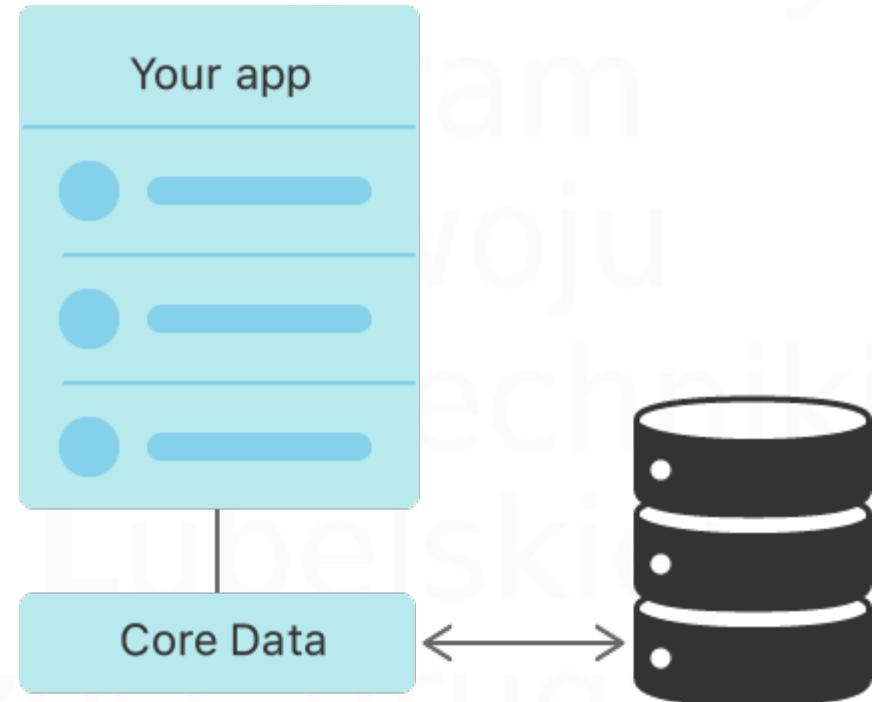
Core Data



źródło: <https://www.objc.io/issues/4-core-data/core-data-overview/>

Core Data

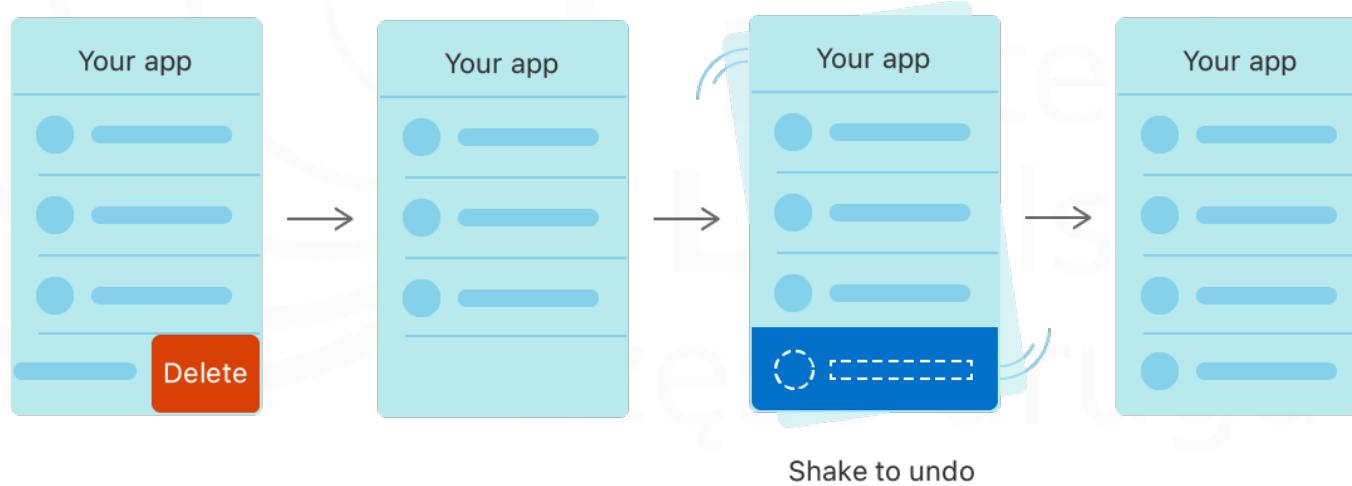
- Persistence
 - Core Data podsumowuje szczegóły mapowania obiektów
 - ułatwia zapisywanie danych bez bezpośredniego administrowania bazą danych



źródło:
<https://developer.apple.com/documentation/coredata/>

Core Data

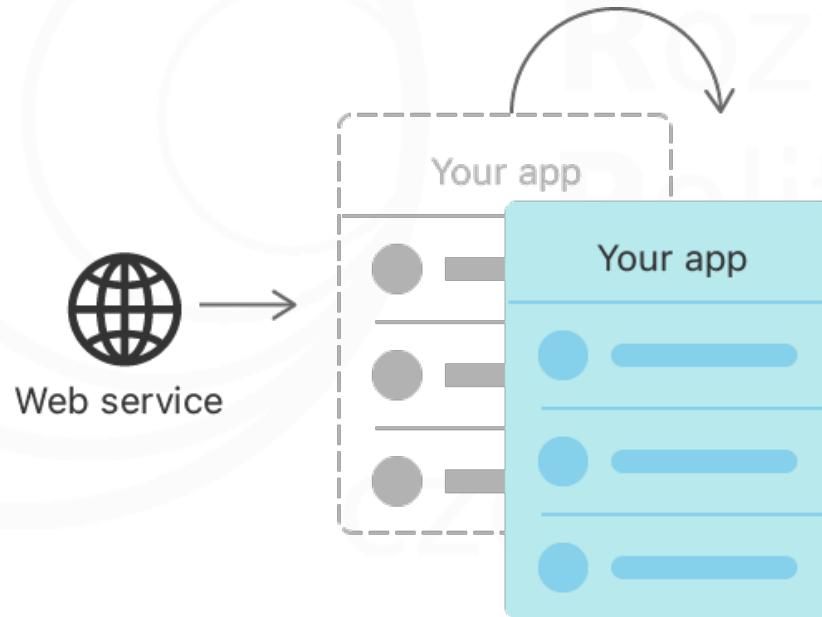
- Operacje cofania i ponawiania
- Menedżer cofania w Core Data śledzi zmiany i może je wycofywać pojedynczo, w grupach lub wszystkie naraz, ułatwiając dodawanie obsługi cofania i ponawiania do aplikacji



źródło: <https://developer.apple.com/documentation/coredata/>

Core Data

- Operacje wykonywane w tle
- blokujące interfejs użytkownika dla zadań takich jak analizowanie JSON



źródło: <https://developer.apple.com/documentation/coredata/>

Core Data

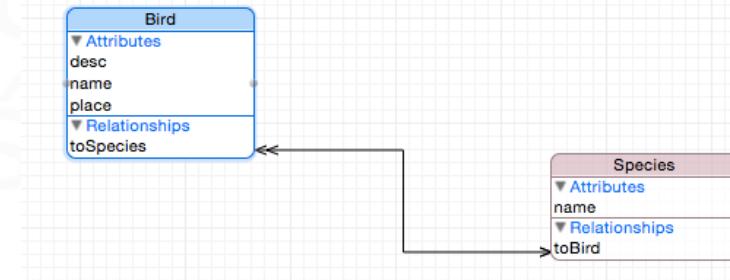
- Model danych
 - struktura danych
 - dla nowego projektu
 - dodawany do istniejącego

The screenshot shows the Xcode Core Data editor interface. On the left, there's a sidebar with sections for ENTITIES, FETCH REQUESTS, and CONFIGURATIONS. Under ENTITIES, 'Bird' is selected, highlighted with a yellow background. The main area is divided into three sections: Attributes, Relationships, and Fetched Properties.

- Attributes:** Shows three attributes: desc (String), name (String), and place (String). 'desc' is currently selected, highlighted with a blue background.
- Relationships:** Shows one relationship: toSpecies (Destination: Species, Inverse: toBird).
- Fetched Properties:** An empty section.

At the bottom, there are buttons for Outline Style, Add Entity, Add Attribute, and Editor Style.

A detailed view of the Attribute configuration for the 'desc' attribute. The 'Name' field is set to 'desc'. The 'Properties' section includes checkboxes for 'Transient' (unchecked), 'Optional' (checked), and 'Indexed' (unchecked). The 'Attribute Type' is set to 'String'. Validation options include 'No Value' (selected) and 'Min Length' (unchecked). A 'Default Value' field contains 'Default Value'. A 'Reg. Ex.' field contains 'Regular Expression'. Advanced options include 'Index in Spotlight' (unchecked) and 'Store in External Record File' (unchecked).



Core Data

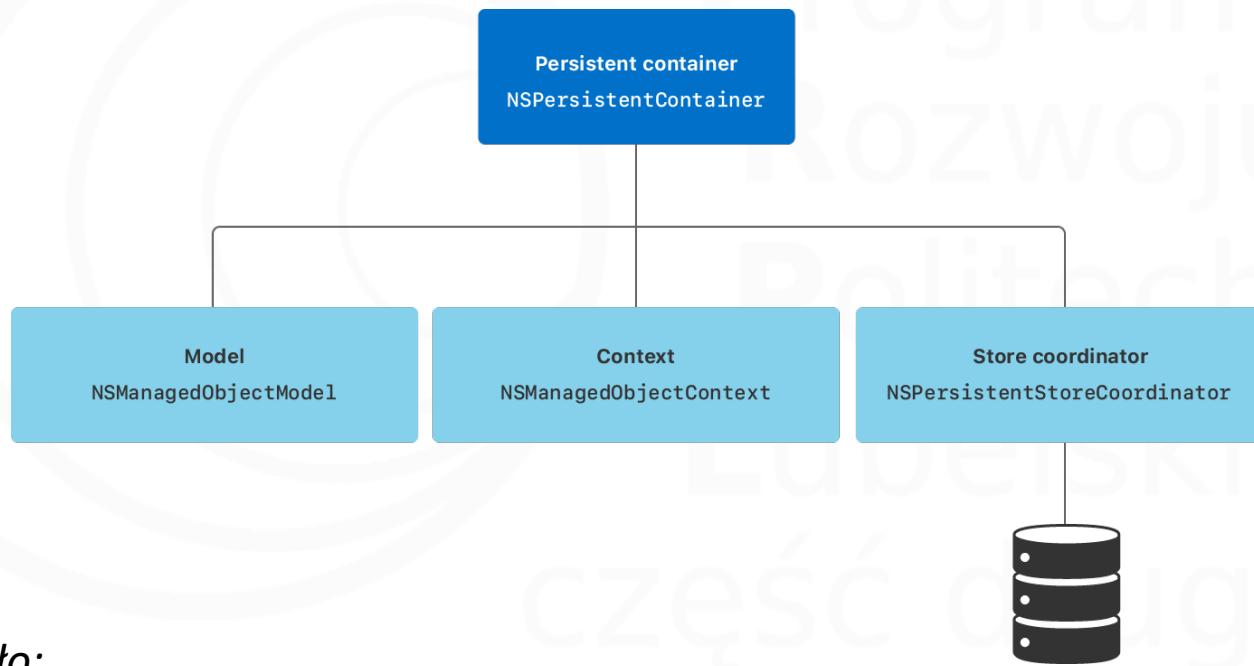
- Model danych:
 - tworzony zazwyczaj za pomocą edytora modelu danych Xcode
 - NSManagedObjectModel reprezentuje plik .xcdatamodeld
 - definiuje się tam encje używane do generowania podklas NSManagedObject
 - tworzone encje są instancjami NSEntityDescription
 - właściwości encji są podklasami NSPropertyDescription
 - NSAttributeDescription dla atrybutów
 - NSRelationshipDescription dla relacji
 - NSFetchedPropertyDescription dla pobranych właściwości

Zarządzanie danymi

- *Core Data persistence stack* – zbiór obiektów współpracujących ze sobą w celu pobierania danych oraz ich zapisywania
- *Core Data persistence stack*:
 - *NSManagedObjectModel* – opisujący strukturę danych
 - *NSManagedObjectContext* – komunikujący się z danymi
 - *NSPersistentStoreCoordinator* – zarządzający aktualnie przechowywanymi danymi
- Od iOS 10 cały stos (stack) jest tworzony za pomocą obiektu *NSPersistentContainer*

Zarządzanie danymi

- *Core Data persistence stack*



źródło:

https://developer.apple.com/documentation/coredata/setting_up_a_core_data_stack

Zarządzanie danymi

- Szablon inicjalizatora dla obiektu NSPersistentContainer

```
lazy var persistentContainer: NSPersistentContainer = {
    let con = NSPersistentContainer(name: "DataName")
    if let err = err {
        fatalError("Blad danych \(err)")
    }
    return con
}()
```

Zarządzanie danymi

- Managed object context jest trwałym kontenerem viewContext
- Managed object context – miejsce, w którym istnieją dane
- Aby odczytać dane, należy je pobrać (fetch) z managed object context
- Aby utworzyć nowe dane, należy je dodać (insert) do managed object context
- Aby zapisać dane, należy zapisać zmieniony managed object context

Zarządzanie danymi

- Zapisanie danych:

```
func saveContext() {  
    let context = self.persistentCointainer.viewContext  
    if context.hasChanged(){  
        try? context.save()  
    }  
}
```

Zarządzanie danymi

- Dodanie managedObjectContext jako wartość przechowywaną w środowisku widoku

- @Environment(\.managedObjectContext) private var viewContext

- Odczytanie danych:

```
@FetchRequest(  
    sortDescriptors: [NSSortDescriptor(keyPath:  
        \Dogs.name, ascending: true)],  
    animation: .default)
```

- Utworzenie listy danych:

```
private var dogs: FetchedResults<Dogs>
```

Zarządzanie danymi

- Filtrowanie danych – **NSPredicate**
 - NSPredicate(format: "imie == %@", "Anna"))
 - NSPredicate(format: "imie < %@", "K"))
 - NSPredicate(format: "stanowisko IN %@",
["Programista", "Administrator",
"Projektant"])
 - NSPredicate(format: "imie BEGINSWITH %@",
"K"))
 - NSPredicate(format: "imie BEGINSWITH [c]
%@", "k"))
 - NSPredicate(format: "NOT imie BEGINSWITH [c]
%@", "a"))

Zarządzanie danymi

- Przykładowa aplikacja

Zintegrowany
Program
Rozwoju
Politechniki
Lubelskiej -
część druga

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
INFORMATYKA



Zintegrowany
Program
Rozwoju
Politechniki
Lubelskiej -
część druga

Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego



Fundusze
Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



POLITECHNIKA LUBELSKA

WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI

INFORMATYKA



Zintegrowany
Program
Rozwoju
Politechniki
Lubelskiej -
część druga

Programowanie aplikacji mobilnych na platformę iOS

W11

Wytwarzanie aplikacji opartych o mapy i lokalizacje

Maria Skublewska-Paszkowska



**Fundusze
Europejskie**
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



Agenda

- Mapy – wprowadzenie
- MapKit
- Core Location
- Przykład implementacji
 - UIKit
 - SwiftUI
- Jak zaimplementować MapKit w SwiftUI – iOS 13?

Mapy – wprowadzenie

- Aplikacje wyświetlają interfejs mapy oraz umożliwiają zaznaczanie adnotacji
- Należy zimportować szkielet programistyczny *MapKit*
- Lokalizacje są definiowane w postaci:
 - *latitude* – szerokości geograficznej
 - *longitude* – długości geograficznej
- Ich nazwy rozpoczynają się od *CL* (*Core Location*)
- Mapa jest wyświetlana poprzez podkласę *UIView* oraz *MKMapView*

Mapy – wprowadzenie

- Typy map:
 - *.standard*
 - *.satelite*
 - *.hybrid*
 - *mutedStandard* – przyciemnia elementy mapy, aby dodatki do widoku aplikacji wyróżniały się
- Definiowanie regionu mapy – MKCoordinateRegion
 - *center* (*CLLocationCoordinate2D*) – szerokość i długość geograficzne punktu znajdującego się w środku *CLLocationCoordinate2DMake()*
 - *span* (*MKCoordinateSpan*) – liczba szerokości i długości geograficznej objętej regionem (zdefiniowanej jako delta)
- Region jest przypisywany do mapy

Mapy – wprowadzenie

- Definiowanie regionu mapy – *MKMapRect* – struktura budowana na podstawie *MKMapPoint* oraz *MKMapSize*
 - zdefiniowanie prostokątnego obszaru mapy na podstawie jednostek, według których mapa jest rysowana
 - *MKMapPoint(_ :)*
 - *coordinate*
 - *distance(to:)* – metry pomiędzy punktami
 - *MKMetersPerMapPointAtLatitude(_ :)*
 - *MKMapPointsPerMeterAtLatitude(_ :)*
 - właściwość *visibleMapReact* – określa jaki widok mapy jest wyświetlany

Mapy – wprowadzenie

- Przewijanie i powiększanie (*Scrolling, Zooming*)
 - domyślnie użytkownik może przewijać i powiększać mapę przy użyciu standardowych gestów
 - *isZoomEnabled*
 - *isScrollEnabled*
 - od iOS 13 można zdefiniować właściwości:
 - *cameraZoomRange* – jak daleko można oddalić obraz od jego środka; inicjalizator bierze pod uwagę maksymalną i minimalną odległość od środka w metrach
 - *cameraBoundary* – maksymalny region, który może zostać wyświetlony; widok mapy nie może być przewijany

Mapy – wprowadzenie

- Zmiana wyświetlanego regionu:
 - `setRegion(_: animated:)`
 - `setCenter(_: animated:)`
 - `setVisibleMapRect(_: animated:)`
 - `setVisibleMapRect(_: edgePadding: animated:)`
- Delegat mapy (`MKMapViewDelegate`) jest wywoływany, gdy mapa jest ładowana i przy zmianie regionu
 - `mapViewWillStartLoadingMap(_:)`
 - `mapViewDidFinishLoadingMap(_:)`
 - `mapViewDidFailLoadingMap(_: withError:)`
 - `mapViewDidChangeVisibleRegion(_:)`
 - `mapView(_: regionWillChangeAnimated:)`
 - `mapView(_: regionDidChangeAnimated:)`

MapKit

- Zapewnia interfejs do osadzania map w oknach i widokach
- Obsługuje:
 - opisywanie mapy
 - dodawanie nakładek
 - informacje o oznaczeniu miejsca dla danej współrzędnej mapy
 - pokazywanie lokalizacji
 - rysowanie tras

MapKit

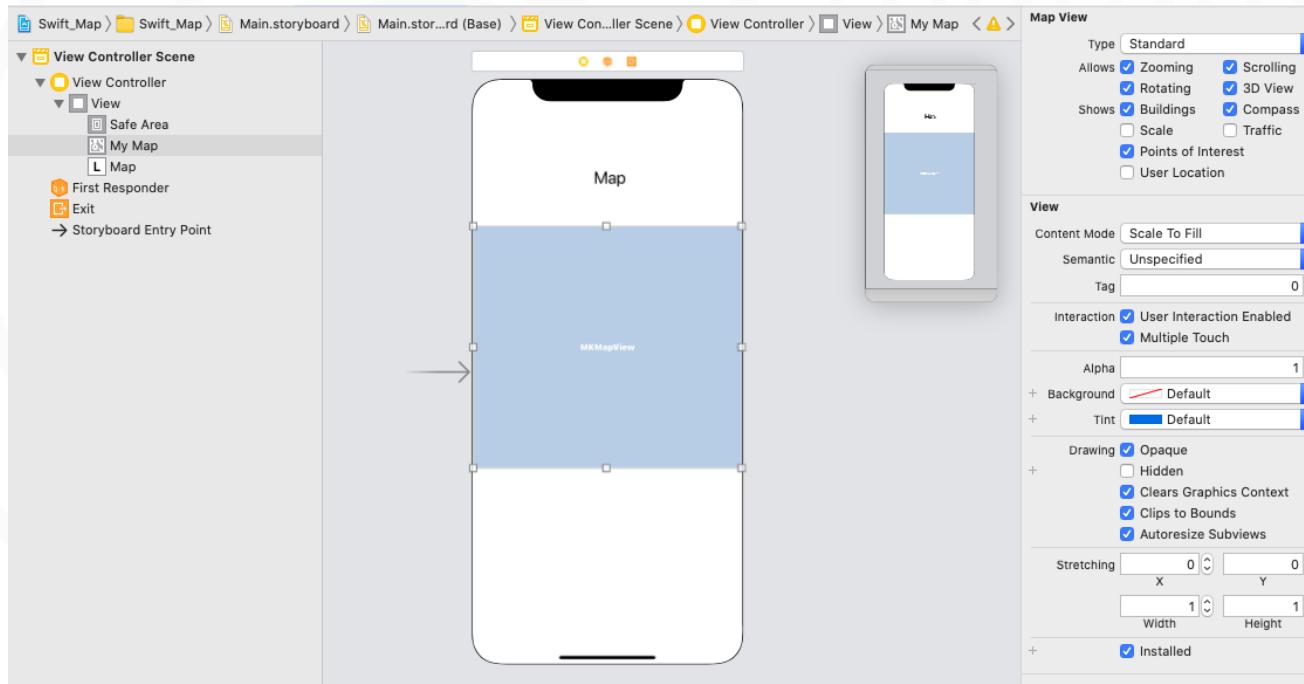
- Protokoły:
 - *MKAnnotation*
 - *MKMapViewDelegate*
 - *MKOverlay*
- Podstawowe klasy:
 - *MKMapView* – zapewnia kanwę, na której mogą być prezentowane informacje o mapach i satelitach
 - *MKMapViewDelegate* – pozwala na otrzymywanie informacji o zmianie lokalizacji lub części pokazywanej mapy lub awarii

Core Location

- Określa aktualną lokalizację użytkownika (położenie urządzenia) i kierunek
- Składa się z klas i protokołów do konfiguracji dostarczania zdarzeń lokalizacji i nagłówków
- Najważniejsze klasy:
 - *CLGeocoder*
 - *CLRegion*
- Definiuje regiony geograficzne
- Monitoruje, kiedy użytkownik przekracza granice tych regionów
- iOS – definiuje region wokół beaconu Bluetooth

Przykład implementacji

- UIKit



Przykład implementacji

- UIKit

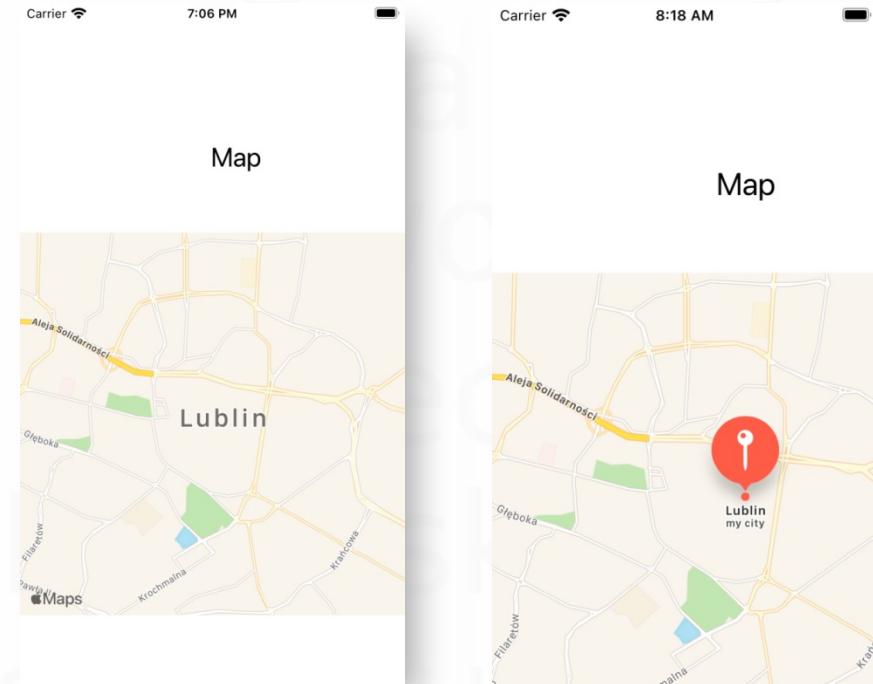
```
super.viewDidLoad()
```

```
→ let centerLocation =  
    CLLocationCoordinate2D(latitude: 51.246452,  
     longitude: 22.568445)  
→ let region = MKCoordinateRegion(center:  
    centerLocation, latitudinalMeters: 100,  
     longitudinalMeters: 100)  
→ myMap.setCameraBoundary(MKMapView.CameraBoundary  
    (coordinateRegion: region), animated: true)  
→ let zoomRange = MKMapView.CameraZoomRange(  
    maxCenterCoordinateDistance: 10000)  
myMap.setCameraZoomRange(zoomRange, animated: true)  
}
```

Przykład implementacji

- UIKit
 - dodawanie adnotacji

```
let lublin = MKPointAnnotation()  
lublin.title = "Lublin"  
lublin.subtitle = "my city"  
lublin.coordinate = centerLocation  
myMap.addAnnotation(lublin)
```



Przykład implementacji

- SwiftUI

```
struct ContentView: View {  
     @State private var region =  
     MKCoordinateRegion(center:  
        CLLocationCoordinate2D(latitude: 51.246452, longitude:  
            22.568445), span:  
     MKCoordinateSpan(latitudeDelta: 0.5, longitudeDelta: 0.5))  
    var body: some View {  
        Text("Map")  
        .padding()  
         Map(coordinateRegion: $region)  
    }  
}
```

Przykład implementacji

- SwiftUI

```
→ struct Places: Identifiable {  
    let id = UUID()  
    let coordinate: CLLocationCoordinate2D  
}  
  
struct ContentView: View {  
    ...  
    → @State private var places: [Places] = [  
        Places(coordinate: .init(latitude: 51.246452,  
                                longitude: 22.568445))]  
    ...
```

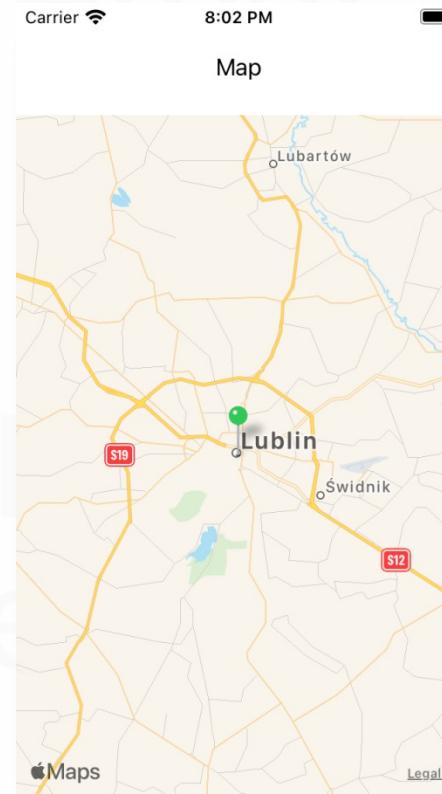
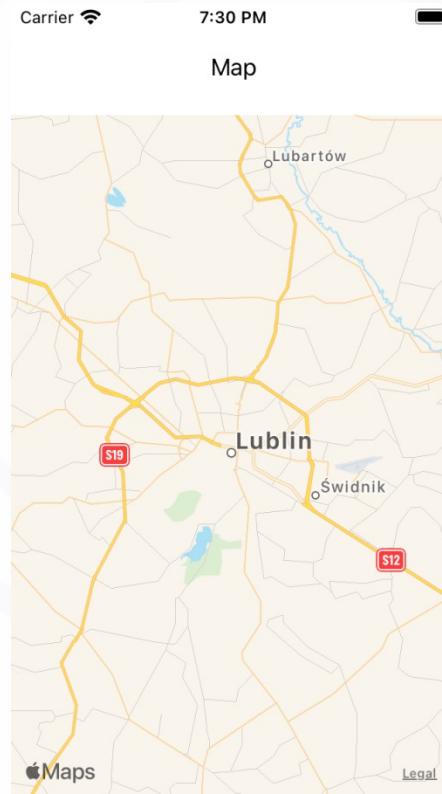
Przykład implementacji

- SwiftUI

```
var body: some View {  
    Text("Map")  
        .padding()  
    ➔ Map(coordinateRegion: $region, annotationItems:  
          places) {  
        place in  
    ➔ MapPin(coordinate: place.coordinate, tint: .green)  
    }  
    }  
}
```

Przykład implementacji

- SwiftUI



Jak zaimplementować MapKit w SwiftUI – iOS 13?

- SwiftUI nie wspierał wszystkich funkcjonalności UIKit oraz AppKit
- Integracja SwiftUI z UIKit pozwala na:
 - implementację brakujących elementów w SwiftUI
 - stopniową migrację projektu z UIKit do SwiftUI
 - ponowne użycie napisanych już wcześniej komponentów
 - tworzenie widoków w SwiftUI opartych o UIKit

Jak zaimplementować MapKit w SwiftUI – iOS 13?

- SwiftUI nie wspiera kontrolerów widoku, które są stosowane w UIKit
- Należy zastosować protokół ***UIViewRepresentable***
- Protokół ten umożliwia zdefiniowanie typów pomocowych pomiędzy UIKit oraz SwiftUI
- Typy te umożliwiają stosowanie instancji klasy UIView w taki sposób, aby była ona kompatybilna z SwiftUI
- Dla widoków nieinteraktywnych (ang. non-interactive) dostępne są dwie metody:
 - tworzenia widoku (*makeUIView*)
 - uaktualniania widoku (*updateUIView*)

Jak zaimplementować MapKit w SwiftUI – iOS 13?

- SwiftUI zawiera opis widoku niż jego konkretną reprezentację
- Nie należy przyjmować żadnych założeń dotyczących stosowanego cyklu życia
- Należy tworzyć podstawowy widok `UIView` w metodzie `makeUIView`, a następnie go aktualizować w metodzie `updateUIView`
- Zdefiniowane właściwości w `makeUIView` nie będą aktualizowane wraz ze zmianą stanu widoku

Jak zaimplementować MapKit w SwiftUI – iOS 13?

- Klasa MyAnnotation

```
class MyAnnotation: NSObject, MKAnnotation {  
    let title: String?  
    let subtitle: String?  
    let coordinate: CLLocationCoordinate2D  
  
    init(title: String?,  
         subtitle: String?,  
         coordinate: CLLocationCoordinate2D) {  
        self.title = title  
        self.subtitle = subtitle  
        self.coordinate = coordinate  
    }  
}
```

Jak zaimplementować MapKit w SwiftUI – iOS 13?

- Utworzenie nowego widoku

```
struct MapView: UIViewRepresentable{  
  
    @Binding var myAnnotation : MyAnnotation  
  
    func makeUIView(context: Context) -> MKMapView  
    {  
        let myMap = MKMapView(frame: .zero)  
        return myMap  
    }  
    ...  
}
```

Jak zaimplementować MapKit w SwiftUI – iOS 13?

- Utworzenie nowego widoku

```
struct MapView: UIViewRepresentable{  
    ...  
    func updateUIView(_ uiView: MKMapView, context: Context) {  
  
        let span = MKCoordinateSpan(latitudeDelta: 0.1,  
                                     longitudeDelta: 0.1)  
        let region = MKCoordinateRegion(center:  
                                         myAnnotation.coordinate, span: span)  
        uiView.setRegion(region, animated: true)  
  
        uiView.addAnnotation(myAnnotation)  
    }  
}
```

Jak zaimplementować MapKit w SwiftUI – iOS 13?

- Utworzenie widoku

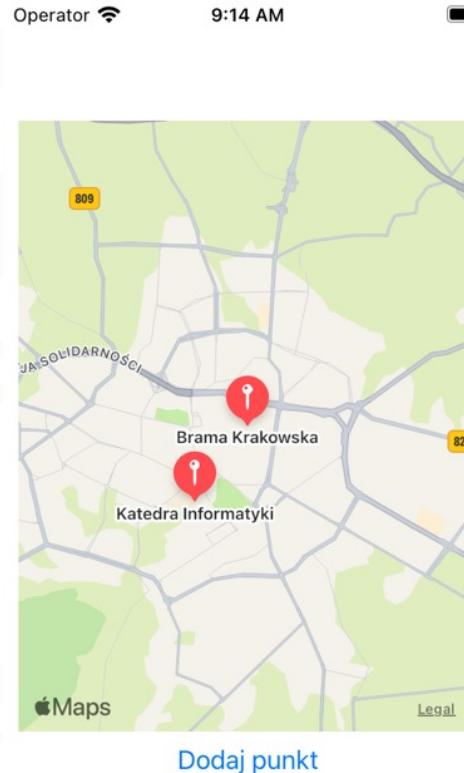
```
struct ContentView: View {  
    @State var myAnnotation = MyAnnotation(title: "Katedra  
        Informatyki", subtitle: "Politechnika Lubelska",  
        coordinate: CLLocationCoordinate2D(latitude:  
            51.2353112433304, longitude: 22.55289822681853))  
  
    var body: some View {  
        MapView(myAnnotation: $myAnnotation)  
            .frame(width: 300, height: 400, alignment: .center)  
        Button("Dodaj punkt", action: {  
            self.addAnnotation()  
        })  
    }  
}
```

Jak zaimplementować MapKit w SwiftUI – iOS 13?

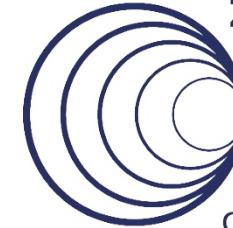
- Utworzenie widoku

```
struct ContentView: View {  
    ...  
    private func addAnnotation() {  
        self.myAnnotation = MyAnnotation(title:  
            "Brama Krakowska",  
            subtitle: "Lublin",  
            coordinate:  
            CLLocationCoordinate2D(latitude:  
                51.24775457385513, longitude:  
                22.56658679983392))  
    }  
}
```

Jak zaimplementować MapKit w SwiftUI – iOS 13?



POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
INFORMATYKA



Zintegrowany
Program
Rozwoju
Politechniki
Lubelskiej -
część druga

Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego



Fundusze
Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny





Programowanie aplikacji mobilnych na platformę iOS

W10

Wytwarzanie aplikacji opartych o widok tabeli

Maria Skublewska-Paszkowska



**Fundusze
Europejskie**
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



Agenda

- Widok tabeli
- UITableView
- List – SwiftUI

Zintegrowany
Program
Rozwoju
Politechniki
Lubelskiej -
część druga

Widok tabeli

- Wiele aplikacji mobilnych dedykowanych dla systemu iOS bazuje na widokach tabelarycznych
- Sterowanie służy do budowania GUI dla jednego widoku (ekranu)
- *TableView* wyświetla dane na pionowej liście, którą użytkownik może przewijać, aby zobaczyć wszystkie elementy
- Składa się z jednej kolumny i wielu wierszy
- Każda sekcja może mieć stopkę i nagłówek
- Tekst i obrazy można umieszczać w rzędach
- Można dodać nagłówek i stopkę dla całego widoku tabeli

Widok tabeli

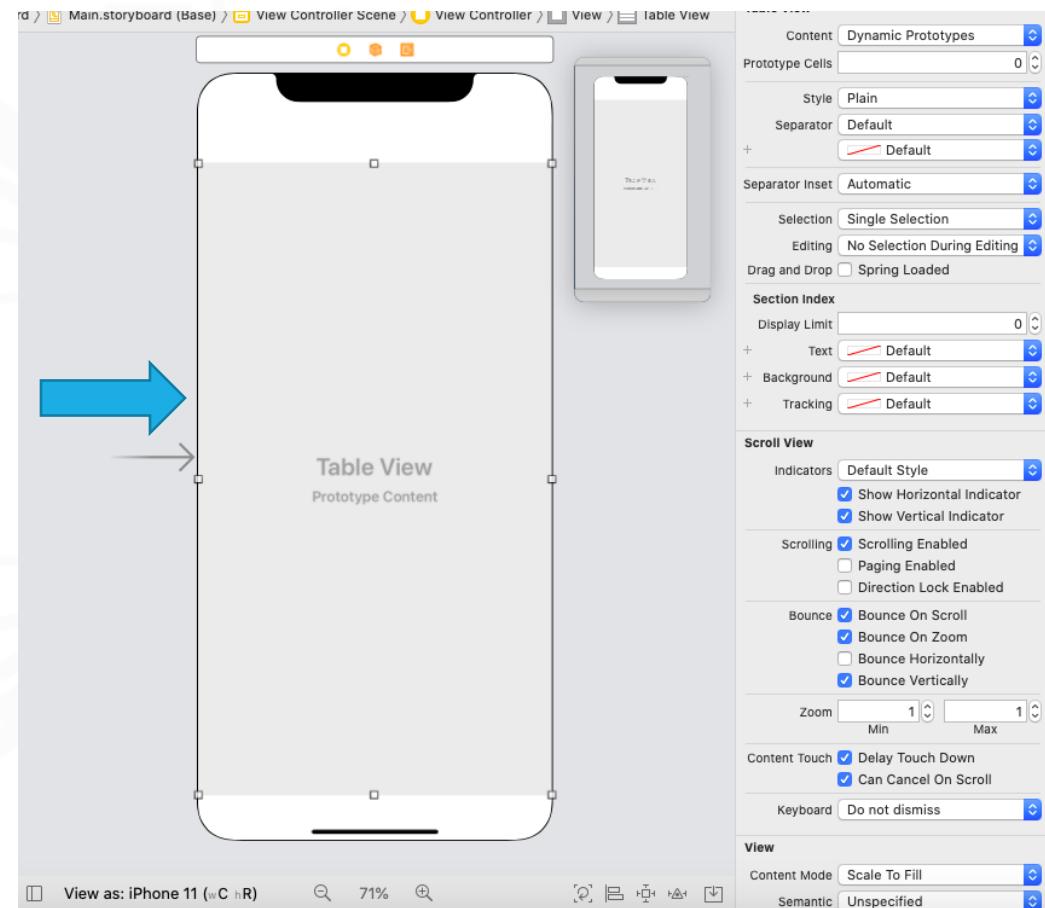
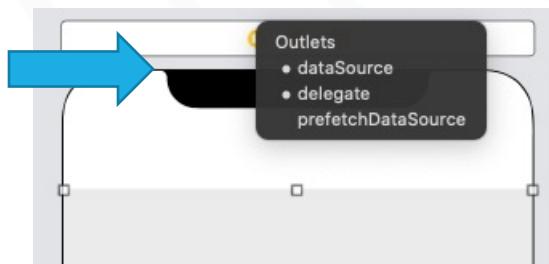
- Jest używany, gdy wyświetlane dane są zorganizowane w sposób hierarchiczny – od najbardziej ogólnych do najbardziej szczegółowych
- Dane ogólne prezentowane są w formie listy, szczegółowe dane w innych widokach (w widokach tabel lub prostych ekranach)
- Częstym zastosowaniem jest pokazanie danych w formie listy indeksowanej, gdzie każdy element ma przypisany unikalny numer
- Istnieje wiele źródeł danych:
 - obiekty NSArray
 - baza danych
- Aplikacje mobilne prezentujące użytkownikowi listę różnych opcji, często też korzystają z tego typu widoków

UITableView

- *UITableView* – widok prezentujący dane za pomocą wierszy ułożonych w jednej kolumnie
- Klasa *UITableView* należy do szkieletu oprogramowania UIKit
- Treść jest wyświetlana w komórkach — obiekty *UITableViewCell*:
 - *standard* – wyświetlanie tekstów i obrazów
 - *custom* – dowolna zdefiniowana treść, rozlokowana według ustaleń programisty
- Możliwe akcje: wyświetlanie danych, przewijanie listy, filtrowanie danych oraz wykonywanie innych akcji po kliknięciu

UITableView

- Dodanie widoku tabeli
- Dodanie:
 - *delegate*
 - *datasource*



UITableView

- Implementacja *metod*:

```
func numberOfSections(in tableView: UITableView) -> Int {
```

→ *return 2;*

```
}
```

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
```

→ *switch section {*

case 0 : return 6

case 1: return 4

default: return 1

```
}
```

UITableView

- Implementacja *metod*:

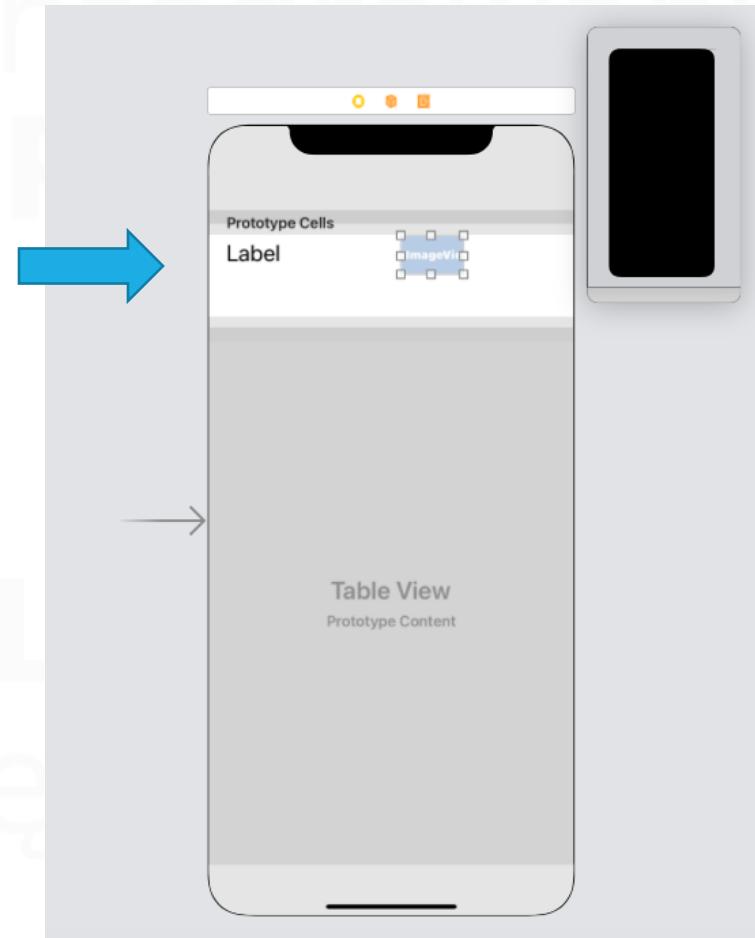
```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {  
    let cell = tableView.dequeueReusableCell(withIdentifier: "newTable", for: indexPath)  
  
    cell.textLabel?.text = "Section: \(indexPath.section)  
                           row: \(indexPath.row)"  
  
    return cell  
}
```

UITableView



UITableView

- Cell:
 - tekst
 - obraz
- Custom Cell:
 - dostosowanie liczby i typów elementów
 - utworzenie nowej klasy i przypisanie jej do komórki



UITableView

```
let car = ["volkswagen", "ford", "mercedes", "porsche", "audi"]  
let logo = ["vw.jpg", "ford.png", "merc.jpg", "porsche.jpg", "audi.jpg"]
```

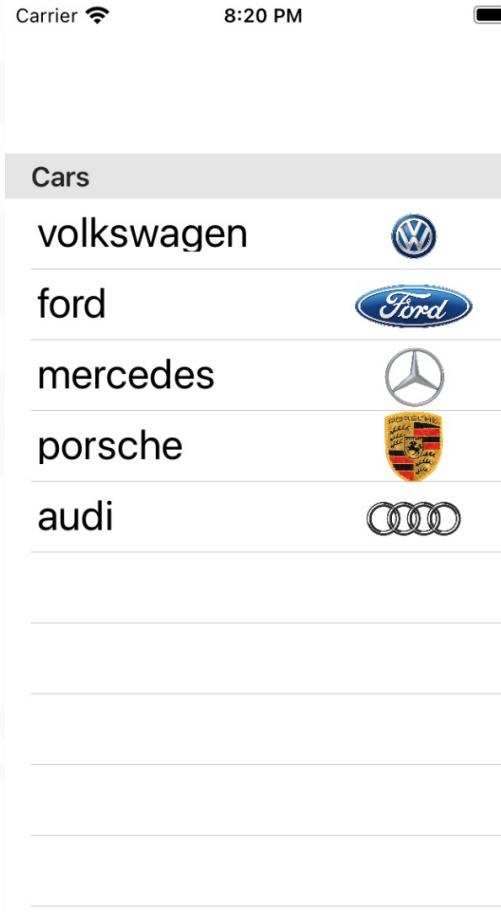
```
func numberOfRows(in tableView: UITableView) -> Int {  
    return 1;  
}  
  
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {  
    return car.count  
}
```

UITableView

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {  
    let cell = tableView.dequeueReusableCell(withIdentifier: "cars", for: indexPath) as!  
        CarTableViewCell  
      
    cell.carImg.image = UIImage(named: logo[indexPath.row])  
    cell.carLabel.text = car[indexPath.row]  
    return cell  
}  
func tableView(_ tableView: UITableView,  
titleForHeaderInSection section: Int) -> String? {  
    return "Cars"  
}
```

UITableView

```
func tableView(_ tableView:  
    UITableView,  
    titleForHeaderInSection  
    section: Int) -> String? {  
    return "Cars"  
}  
  
• header  
  
• footer
```



UITableView

- Dane z podziałem na sekcje

```
struct Komputer{  
    let model: String  
    let producent: String  
    let liczbaRdzieni: Int  
    let pamieci: Int  
    let cena: Double  
    let stacjonarny: Bool  
}
```

```
struct Drukarka{  
    let model: String  
    let producent: String  
    let czyDupleks: Bool  
    let czySkaner: Bool  
    let liczbaStrMin: Int  
    let cena: Double  
}
```

UITableView

- viewDidLoad()

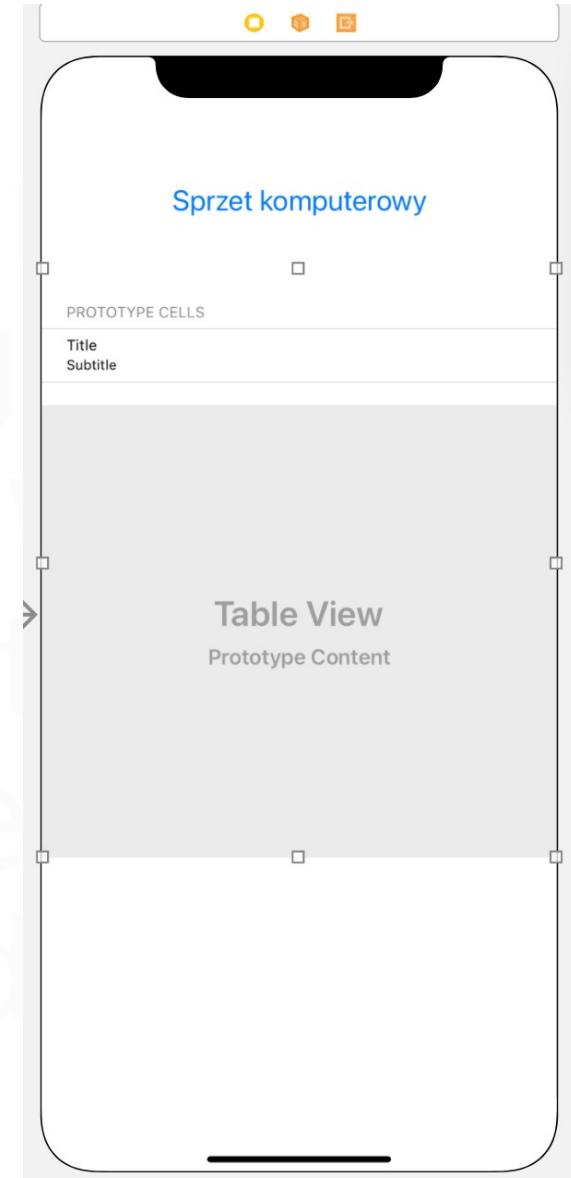
```
komputery = [Komputer(model: "MacBook Pro",  
producent: "Apple", liczbaRdzeni: 8, pamieci: 8,  
cena: 7200, stacjonarny: false),  
Komputer(model: "Legion T5", producent:  
"Lenovo", liczbaRdzeni: 8, pamieci: 16, cena:  
5000, stacjonarny: true)]
```

```
drukarki = [Drukarka(model: "B215", producent:  
"Xerox", czyDupleks: true, czySkaner: true,  
liczbaStrMin: 25, cena: 2500),  
Drukarka(model: "Smart Tank 515", producent:  
"HP", czyDupleks: false, czySkaner: true,  
liczbaStrMin: 11, cena: 799)]
```

UITableView

- class ViewController:
UIViewController,
UITableViewDelegate,
UITableViewDataSource {

```
var komputery:  
    [Komputer] = []  
  
var drukarki:  
    [Drukarka] = []  
  
@IBOutlet weak var  
myTableView:  
UITableView!
```



UITableView

```
func numberOfRows(in tableView: UITableView)  
-> Int {  
    return 2  
}  
  
func tableView(_ tableView: UITableView,  
numberOfRowsInSection section: Int) -> Int {  
    switch(section) {  
        case 0: return komputery.count  
        case 1: return drukarki.count  
        default: return 0  
    }  
}
```



UITableView

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {  
    let cellId = "cell1"  
  
    var cell =  
        tableView.dequeueReusableCell(withIdentifier:  
            cellId) as? UITableViewCell  
  
    if cell == nil {  
  
        let cell = UITableViewCell(style: .value2,  
            reuseIdentifier: "cellId")  
  
    }  
}
```

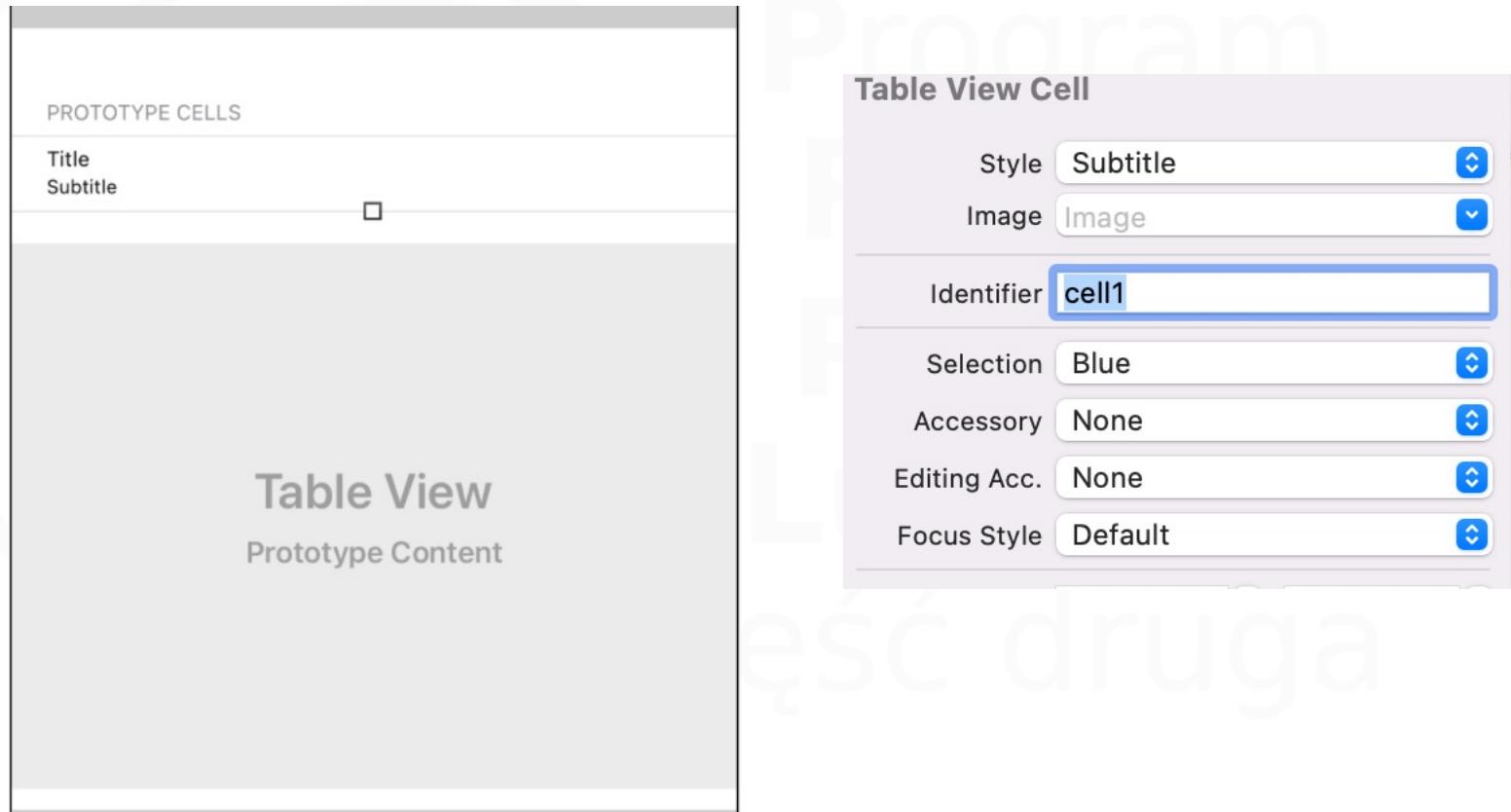
UITableView

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell  
...  
    if(indexPath.section == 0) {  
        cell?.textLabel?.text =  
            komputery[indexPath.row].model  
        cell?.detailTextLabel?.text =  
            komputery[indexPath.row].producent}  
    else {  
        cell?.textLabel?.text =  
            drukarki[indexPath.row].model  
        cell?.detailTextLabel?.text =  
            drukarki[indexPath.row].producent }  
    return cell!  
}
```

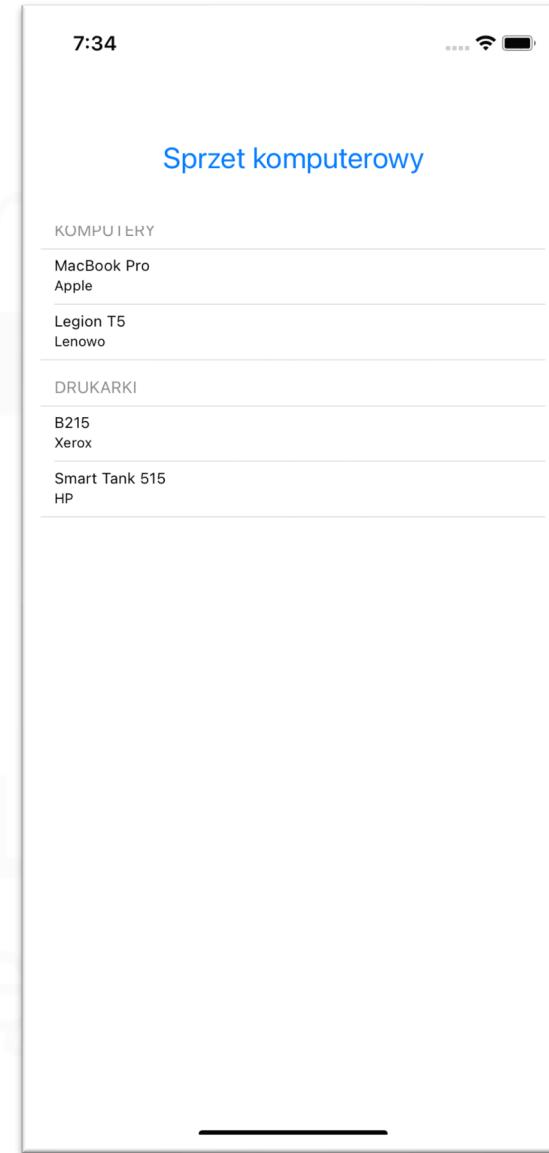
UITableView

```
func tableView(_ tableView: UITableView,  
titleForHeaderInSection section: Int) -> String? {  
    if(section == 0){  
        return "Komputery"  
    }  
    else{  
        return "Drukarki"  
    }  
}
```

UITableView



UITableView



List - SwiftUI

- Szkielet programistyczny SwiftUI
- Kodowanie w *ContentView.swift*
- Tworzenie listy:
 - utworzenie wiersza na liście (podobnie jak UITableView)
 - połączenie wiersza z danymi
 - dodanie paska nawigacyjnego (opcjonalnie)

List - SwiftUI

- Lista bazująca na tabeli

```
struct carInfo {  
    var carName: String!  
    var imgName : String!  
}
```

```
let cars: [carInfo] = [carInfo(carName: "volkswagen", imgName: "vw"),  
                      carInfo(carName: "ford", imgName: "ford"),  
                      carInfo(carName: "mercedes", imgName: "merc"),  
                      carInfo(carName: "porsche", imgName: "porsche"),  
                      carInfo(carName: "audi", imgName: "audi"),  
]
```

List - SwiftUI

- Lista bazująca na tabeli

```
struct ContentView: View {  
    var body: some View {  
        List(0..            VStack(alignment: .leading)  
            Text(cars[i].carName)  
                .padding()  
        }  
        Image(cars[i].imgName)  
            .resizable().scaledToFit()  
    }  
}
```



List - SwiftUI

- Tworzenie list dynamicznych na podstawie kolekcji danych

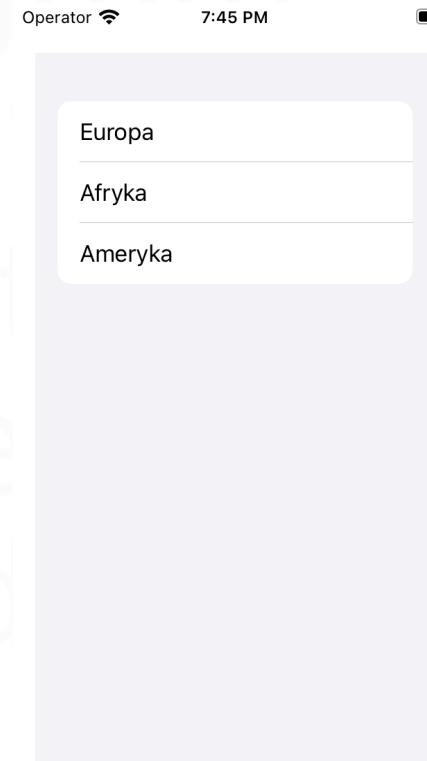
```
struct Kontynent: Identifiable{  
    let nazwa: String  
    let id = UUID()  
}
```

```
struct ContentView: View {  
    let kontynenty =  
        [Kontynent(nazwa: "Europa") ,  
         Kontynent(nazwa: "Afryka") ,  
         Kontynent(nazwa: "Ameryka") ]
```

List - SwiftUI

- Tworzenie list dynamicznych na podstawie kolekcji danych

```
var body: some View {  
    List(kontynenty) {  
        Text($0.nazwa)  
    }  
    .padding()  
}
```



List - SwiftUI

- Tworzenie list dynamicznych z możliwością zaznaczania

```
struct ContentView: View {  
    let kontynenty =  
        [Kontynent(nazwa: "Europa"),  
         Kontynent(nazwa: "Afryka"),  
         Kontynent(nazwa: "Ameryka")]
```



```
@State private var zaznaczenie =  
    Set<UUID>()
```

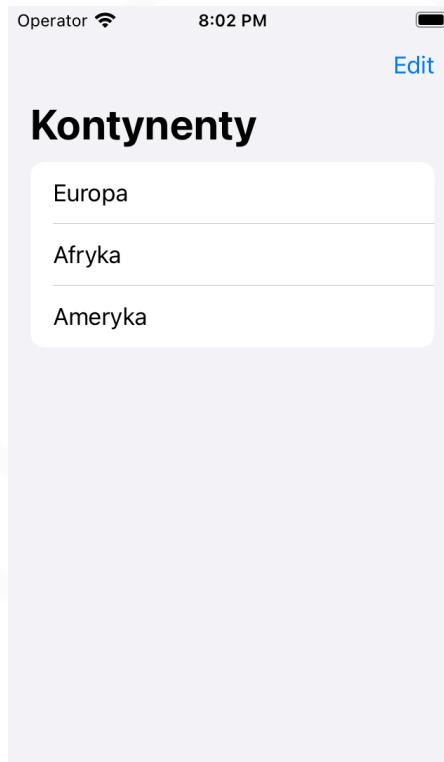
List - SwiftUI

- Tworzenie list dynamicznych z możliwością zaznaczania

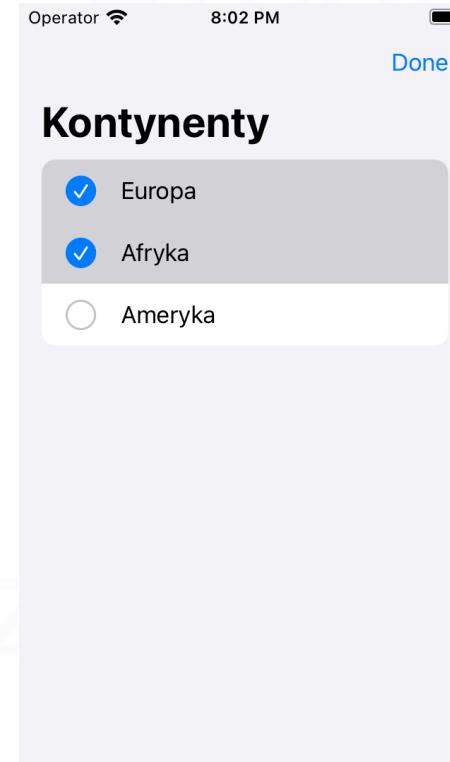
```
var body: some View {  
    NavigationView{  
        List(kontynenty, selection: $zaznaczenie) {  
            Text($0.nazwa)  
        }  
        .navigationTitle("Kontynenty")  
        .toolbar{  
            EditButton()  
        }  
    }  
    Text("\(zaznaczenie.count) zaznaczeń")  
}
```

List - SwiftUI

- Tworzenie list dynamicznych z możliwością zaznaczania



0 zaznaczeń



2 zaznaczeń

List - SwiftUI

- Tworzenie list ze wzorem komórki

```
struct Danie{  
    let nazwa: String  
    let rozmiar: String  
    let cena: Double  
    let obraz: String  
}
```

List - SwiftUI

- Tworzenie list ze wzorem komórki
 - utworzenie nowego widoku

```
struct Row: View {  
    let danie: Danie  
    var body: some View {  
        HStack{  
            Image(danie.obraz)  
                .resizable()  
                .frame(width: 40, height: 40, alignment: .center)  
            VStack{  
                Text(danie.nazwa)  
                Text("\\"(danie.cena, specifier: "%.2f") zł")  
            }  
        }  
    }  
}
```

List - SwiftUI

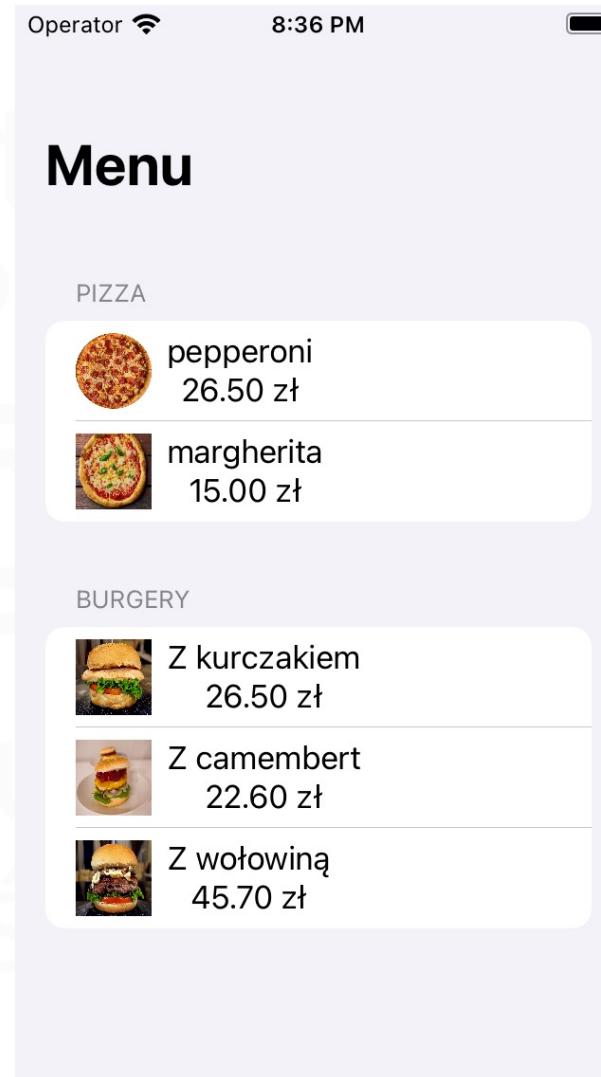
- Tworzenie list ze wzorem komórki

```
struct ContentView: View {  
    let pizzy: [Danie] = [  
        Danie(nazwa: "pepperoni", rozmiar: "45", cena: 26.5,  
               obraz: "p1"),  
        Danie(nazwa: "margherita", rozmiar: "25", cena: 15.0, obraz:  
               "p2")  
    ]  
    let burgery: [Danie] = [  
        Danie(nazwa: "Z kurczakiem", rozmiar: "XL", cena: 26.5,  
               obraz: "b1"),  
        Danie(nazwa: "Z camembert", rozmiar: "L", cena: 22.60, obraz:  
               "b2"),  
        Danie(nazwa: "Z wołowina", rozmiar: "XXL", cena: 45.7, obraz:  
               "b3")  
    ]
```

List - SwiftUI

- Tworzenie list ze wzorem komórki

```
var body: some View {  
    NavigationView{  
        List{  
            Section("Pizza"){  
                ForEach(pizzi, id:\.nazwa){ p in  
                    Row(danie: p)  
                }  
            }  
            Section("Burgery"){  
                ForEach(burgery, id:\.nazwa){b in  
                    Row(danie: b)  
                }  
            }  
        }.navigationTitle("Menu")  
    }  
}
```



List - SwiftUI

- Tworzenie list z nawigacją
 - utworzenie nowego widoku

```
struct WybraneDanie: View {  
    let wybraneDanie: Danie  
    var body: some View {  
        VStack{  
            Image(wybraneDanie.obraz)  
                .resizable()  
                .frame(width: 150, height: 120, alignment: .center)  
            Text("Wybrano: \(wybraneDanie.nazwa)")  
            Text("Rozmiar: \(wybraneDanie.rozmiar)")  
            Text("Cena: \(wybraneDanie.cena, specifier: "%.2f") zł")  
        }  
    }  
}
```

List - SwiftUI

- Tworzenie list z nawigacją
 - dodanie nawigacji

```
NavigationView{  
    List{  
        Section("Pizza") {  
            ForEach(pizzy, id:\.nazwa) { p in  
                NavigationLink(destination:  
                    WybraneDanie(wybraneDanie: p)) {  
                    Row(danie: p)  
                }  
            }  
        }  
    }  
}
```



List - SwiftUI

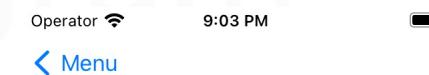
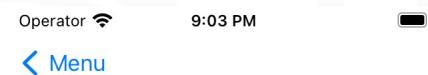
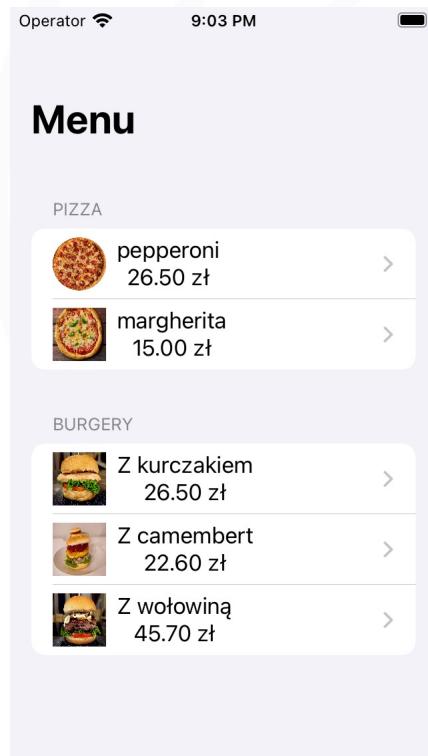
- Tworzenie list z nawigacją
 - dodanie nawigacji

```
NavigationView{  
    List{  
        ...  
        Section("Burgery") {  
            ForEach(burgery, id:\.nazwa) {b in  
                NavigationLink(destination:  
                    WybraneDanie(wybraneDanie: b)) {  
                    Row(danie: b)  
                }  
            }  
        }  
        .navigationTitle("Menu")  
    }  
}
```



List - SwiftUI

- Tworzenie list z nawigacją



List - SwiftUI

- Tworzenie list rozwijanych
 - wprowadzona od iOS 14
 - listy SwiftUI mają ulepszony inicjator, który pozwala tworzyć rozwijające się sekcje z elementami podroznymi, renderowane za pomocą strzałek
 - do takiej postaci listy, trzeba mieć dane w precyzyjnej formie: model danych powinien zawierać opcjonalną tablicę dzieci tego samego typu w celu utworzenia drzewa (np. JSON)
 - należy zdefiniować elementy główne i dzieci

List - SwiftUI

- Tworzenie list rozwijanych
 - wprowadzona od iOS 14

```
struct Favourite: Identifiable{  
    let id = UUID()  
    let name: String  
    let desc: String
```

→ var favouriteItems: [Favourite]?

```
static let dog1 = Favourite(name: "Owczarek niemiecki",  
                           desc: "lat 7")  
static let dog2 = Favourite(name: "Husky", desc:  
                           "lat 4")
```

List - SwiftUI

- Tworzenie list rozwijanych
 - wprowadzona od iOS 14

```
static let book1 = Favourite(name: "Skazanie",  
desc: "R. Mróz")
```

```
static let book2 = Favourite(name: "Cyfrowa  
twierdza", desc: "D. Brown")
```

```
static let dogs = Favourite(name: "Psy", desc: "",  
favouriteItems: [dog1, dog2])
```

```
static let books = Favourite(name: "Książki", desc:  
"", favouriteItems: [book1, book2])  
}
```

List - SwiftUI

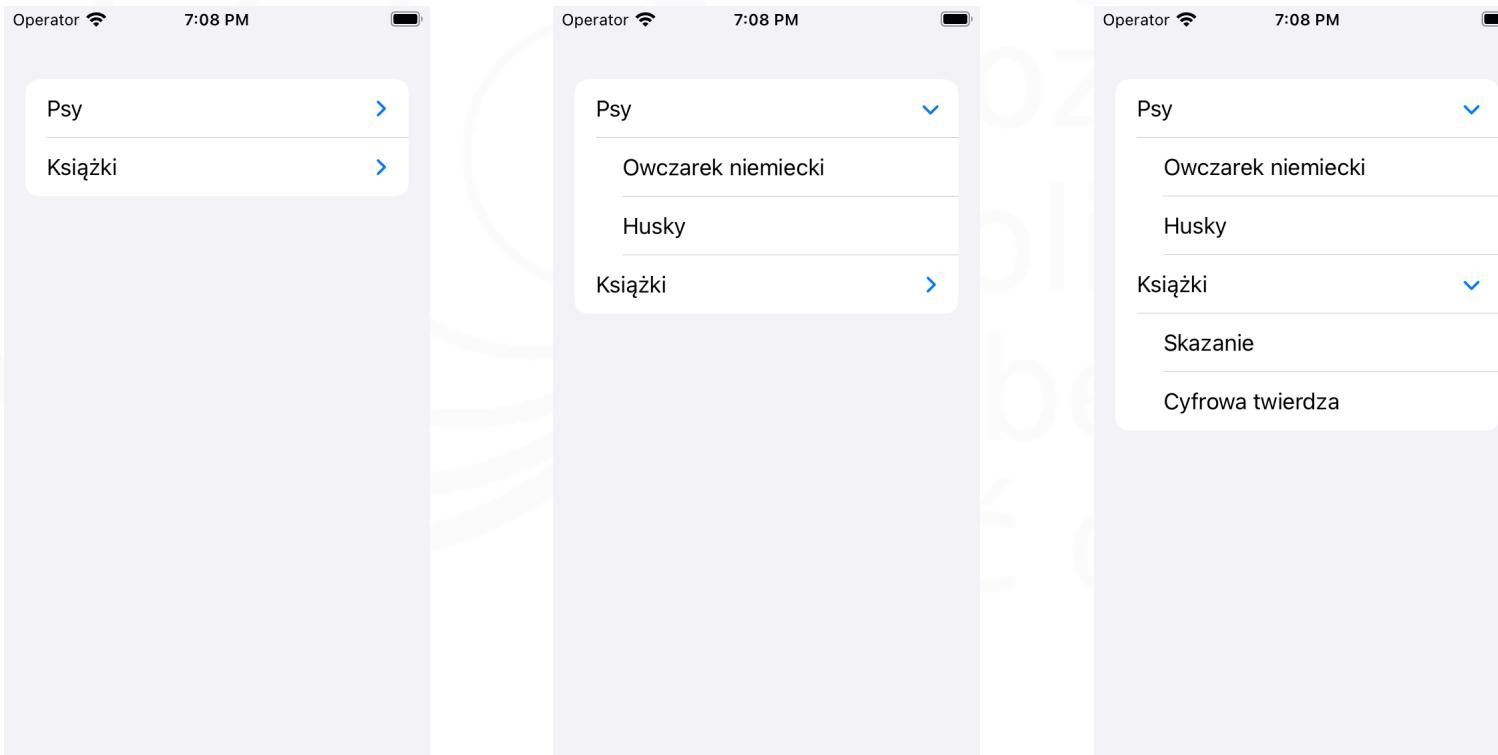
- Tworzenie list rozwijanych
 - wprowadzona od iOS 14

```
struct ContentView: View {  
    let listItems: [Favourite] = [.dogs, .books]  
    var body: some View {  
        List(listItems, children:  
            \.favouriteItems) { item in  
            Text(item.name)  
        }  
    }  
}
```



List - SwiftUI

- Tworzenie list rozwijanych
 - wprowadzona od iOS 14



POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
INFORMATYKA



Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego



**Fundusze
Europejskie**
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny





Programowanie aplikacji mobilnych na platformę iOS

W9

Wytwarzanie prostych aplikacji iOS

Maria Skublewska-Paszkowska



**Fundusze
Europejskie**
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



Agenda

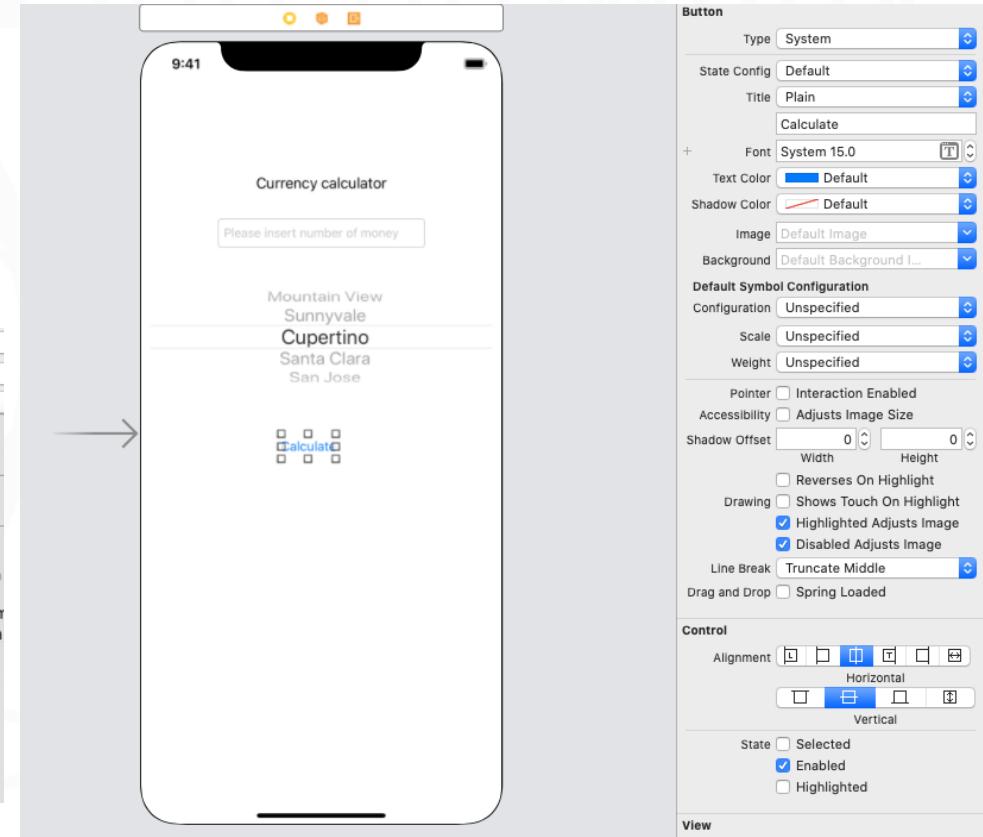
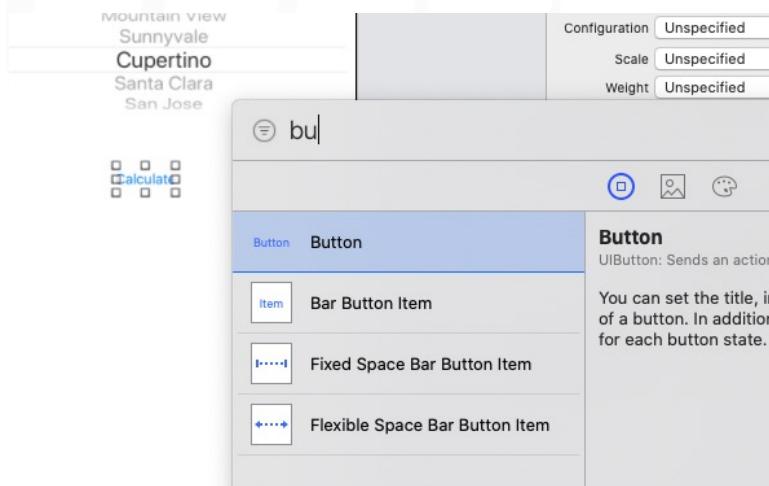
- Przykładowa aplikacja z jednym widokiem
 - UIKit
 - SwiftUI
- Aplikacja z wieloma widokami
- Przykładowa aplikacja z wieloma widokami
 - UIKit
 - SwiftUI
- Przekazywanie danych między widokami
 - UIKit
 - SwiftUI

Przykładowa aplikacja z jednym widokiem

- Aplikacja *Currency calculator* z pojedynczym widokiem przeliczającą podaną wartość kwoty na PLN
- Waluta, z której wartość jest przeliczana, wybierana jest z rozwijanej listy
- Domyślnie wybrana jest waluta USD
- W aplikacji przechowywana jest wartość waluty do przeliczenia
- Jeśli wartość waluty nie została wprowadzona, etykieta wyświetlająca tekst powinna być pusta
- Przeliczona wartość wyświetlona jest w etykiecie

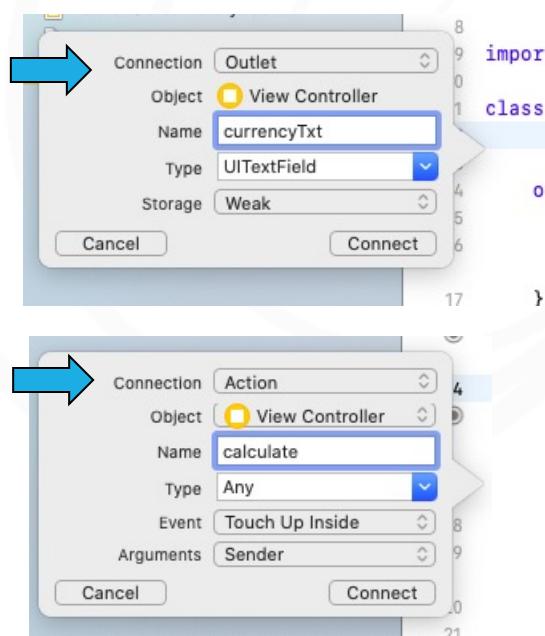
Przykładowa aplikacja z jednym widokiem

- Aplikacja UIKit
 - utworzenie widoku w Storyboard



Przykładowa aplikacja z jednym widokiem

- Aplikacja UIKit
 - utworzenie powiązania z *ViewController.swift*



import UIKit

```
class ViewController:  
    UIViewController {  
@IBOutlet weak var currencyTxt:  
    UITextField!  
@IBOutlet weak var currencyPicker:  
    UIPickerView!  
@IBOutlet weak var resultLabel:  
    UILabel!  
@IBAction func calculate(_ sender: Any)  
{  
}  
}
```

Przykładowa aplikacja z jednym widokiem

- Aplikacja UIKit
 - UIPickerView implementation

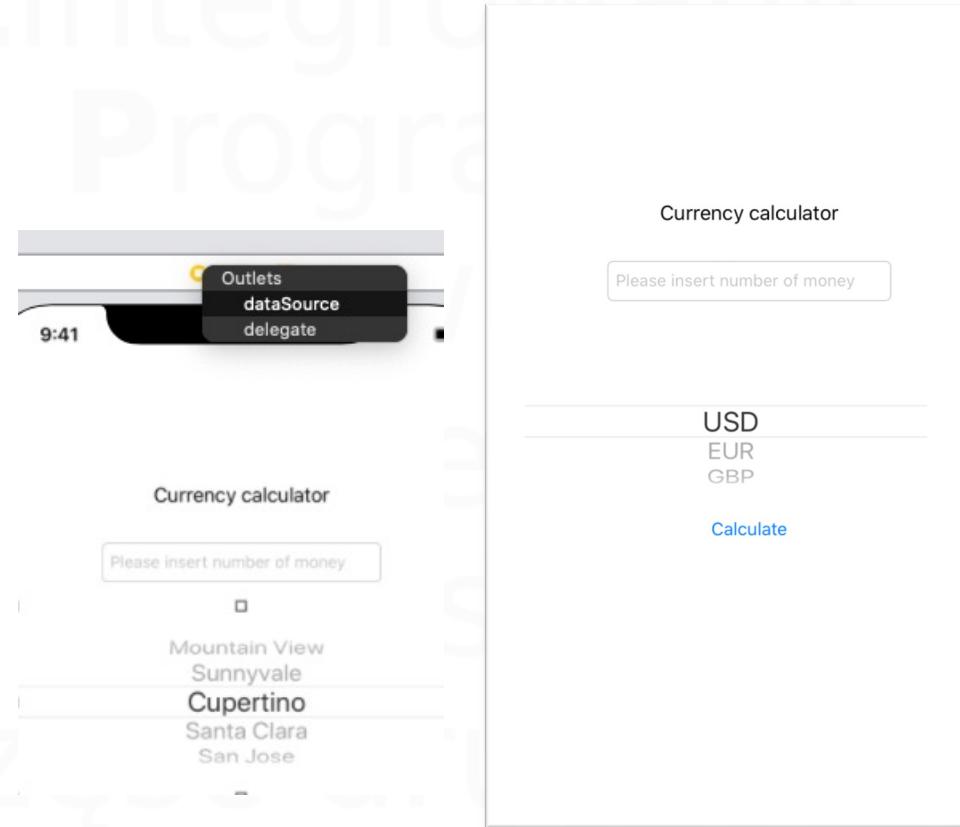
```
class ViewController:  
    UIViewController,  
    UIPickerViewDelegate,  
    UIPickerViewDataSource {  
  
    var pickerItem : [String] =  
        [String]()  
    ...
```

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    pickerItem = ["USD", "EUR",  
    "GBP"]  
}  
  
func numberOfComponents (in  
pickerView: UIPickerView) -> Int {  
    return 1  
}  
  
func pickerView(_ pickerView:  
UIPickerView,  
numberOfRowsInComponent  
component: Int) -> Int {  
    return pickerItem.count  
}
```

Przykładowa aplikacja z jednym widokiem

- Aplikacja UIKit
 - implementacja *PickerView*

```
func pickerView(_ pickerView:  
UIPickerView, titleForRow row: Int,  
forComponent component: Int) ->  
String? {  
    return pickerItem[row]  
}
```



Przykładowa aplikacja z jednym widokiem

- Aplikacja UIKit

```
func pickerView(_ pickerView:  
    UIPickerView, didSelectRow row:  
    Int, inComponent component: Int) {  
  
    selectedItem = row  
}  
}
```

```
@IBAction func calculate  
    (_ sender: Any) {  
  
    if self.currencyTxt.text!.isEmpty {  
        resStr = "Please input the  
        amount of money!"  
    } else {  
        if let val = Double(currencyTxt.text!) {  
            valPln = val * currVal[selectedItem]  
            resStr = "\((val)) PLN is " + "\((valPln)) "  
            + pickerItem[selectedItem]  
        }  
    }  
    resLabel.text = resStr  
}
```

Przykładowa aplikacja z jednym widokiem

The image displays two screenshots of a mobile application interface for a currency calculator. Both screenshots have a light gray header bar with the text "Currency calculator".

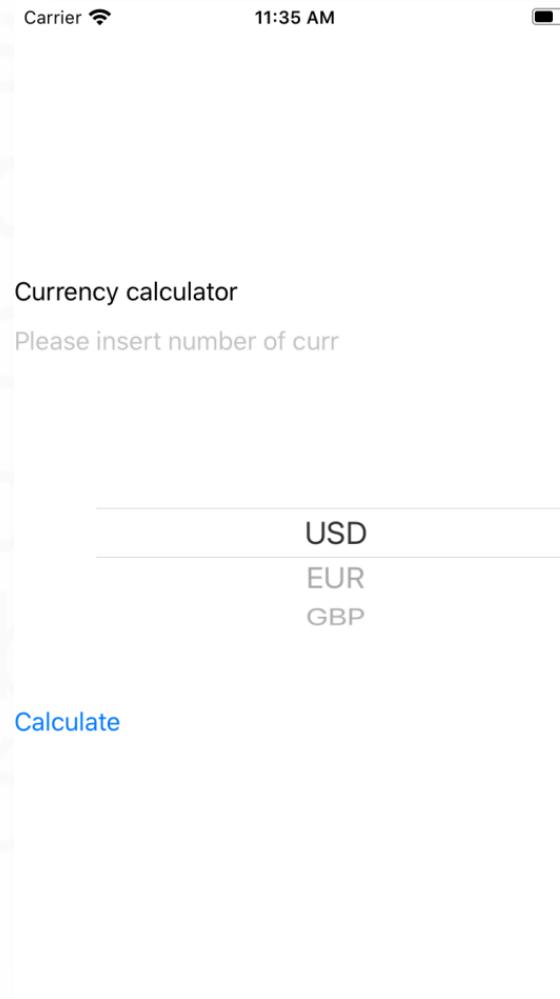
Screenshot 1: The first screenshot shows an empty input field with the placeholder "Please insert number of money". Below the input field are three currency selection buttons: "USD", "EUR", and "GBP". A blue "Calculate" button is located below these. At the bottom of the screen, there is a message: "Please input the amount of money!".

Screenshot 2: The second screenshot shows the same interface after an action. The input field now contains the number "3". The message at the bottom has disappeared. The currency selection buttons remain the same. A blue "Calculate" button is visible. At the bottom of the screen, there is a message: "3.0 PLN is 12.36 EUR".

Przykładowa aplikacja z jednym widokiem

- Aplikacja SwiftUI

```
struct ContentView: View {  
    @State var curr = ["USD", "EUR", "GBP"]  
    @State var currVal = [3.5, 4.12, 5.07]  
    @State var selectedCurr = 0  
    @State private var num: String = ""  
    @State var pln : Double = 0  
    @State var str : String = ""
```



Przykładowa aplikacja z jednym widokiem

- Aplikacja SwiftUI

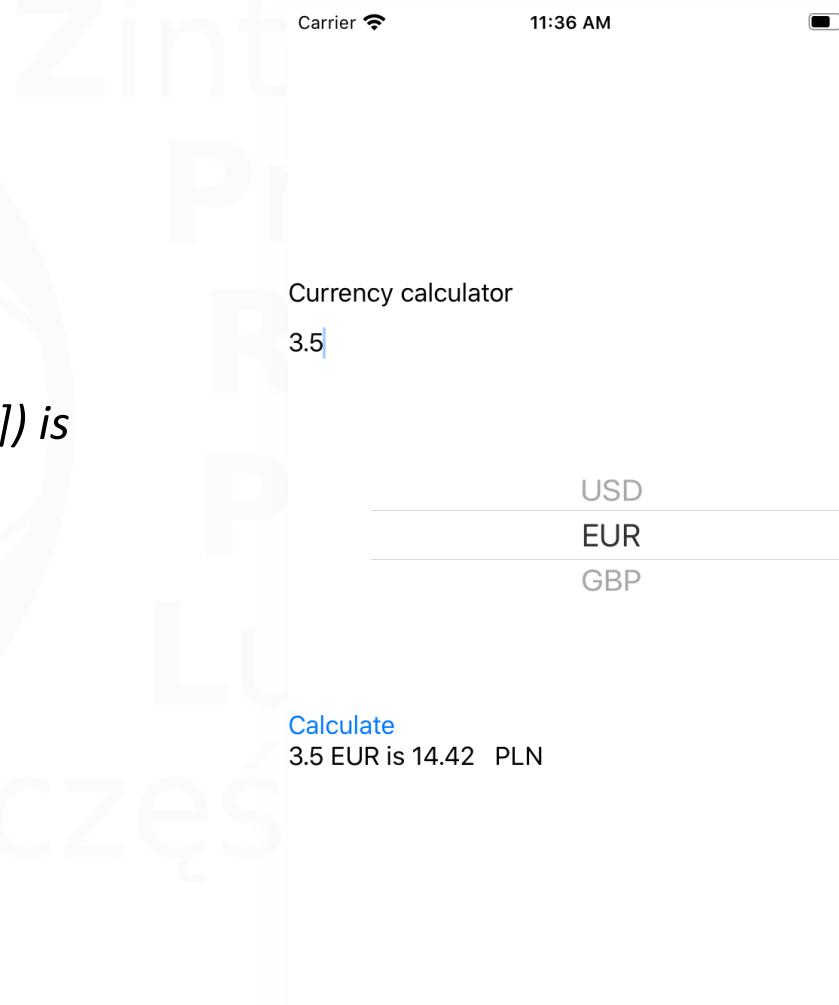
```
 VStack (alignment:.leading){  
     Text("Currency calculator")  
  
     TextField("Please insert  
number  
of curr", text:  
$num)  
  
     Picker(selection:  
$selectedCurr,  
            label: Text("")) {  
         ForEach(0..<curr.count){  
             Text(self.curr[$0])  
         }  
     }  
 }
```

```
 Button(action: {  
     if self.num.isEmpty{  
         self.str = "Please input the  
number"  
     } else {  
         let valpln : Double =  
             self.currVal[self.selectedCurr]  
         if let currNum =  
             Double(self.num){  
             self.pln = valpln*currNum  
         }  
     }  
 } ... }  
 }
```

Przykładowa aplikacja z jednym widokiem

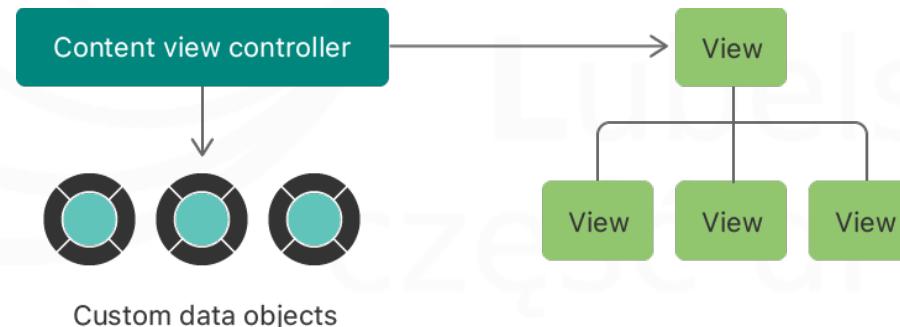
- Aplikacja SwiftUI

```
...
self.str = "\(self.num)
\(self.curr[self.selectedCurr]) is
\($0,2) PLN"
}
}) {
    Text("Calculate")
}
Text(self.str)
}
```



Aplikacja z wieloma widokami

- W UIKit kontroler widoków zarządza hierarchią widoków i informacjami o stanie potrzebnymi do utrzymywania aktualności tych widoków
- Aplikacja UIKit opiera się na kontrolerach widoku do prezentowania zawartości
- Kontroler widoku jest właścicielem wszystkich swoich widoków i zarządza interakcjami z tymi widokami



źródło:https://developer.apple.com/documentation/uikit/view_controllers/displaying_and_managing_views_with_a_view_controller

Aplikacja z wieloma widokami

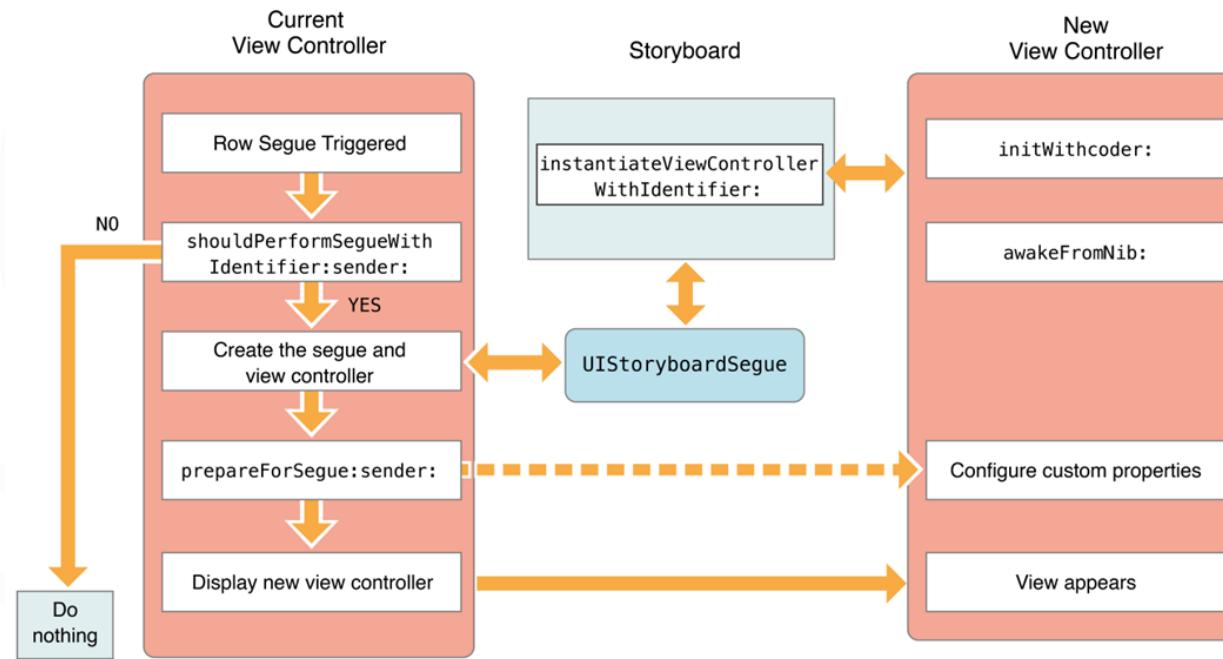
- Segue definiuje przejście między dwoma kontrolerami widoku w pliku storyboard
- Punktem początkowym przejścia jest przycisk, wiersz tabeli lub aparat rozpoznawania gestów, który inicjuje przejście
- Punktem końcowym przejścia jest kontroler widoku, który będzie wyświetlony

Aplikacja z wieloma widokami

- Typy przejść (segue):
 - Show (Push)
 - Show Detail (Replace)
 - Present Modally
 - Present asPopover
- Po utworzeniu przejścia należy wybrać obiekt przejścia i przypisać mu identyfikator
- Identyfikator służy do określenia, która sekwencja została uruchomiona

Aplikacja z wieloma widokami

- Zachowanie przejść

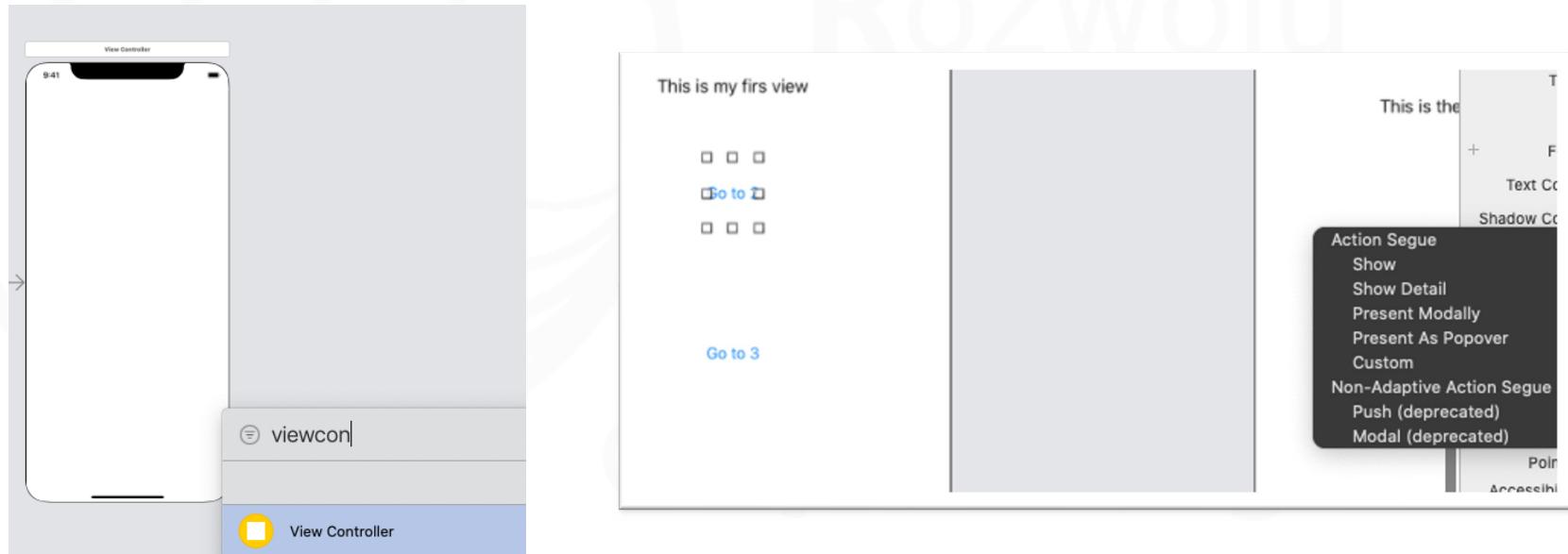


źródło:

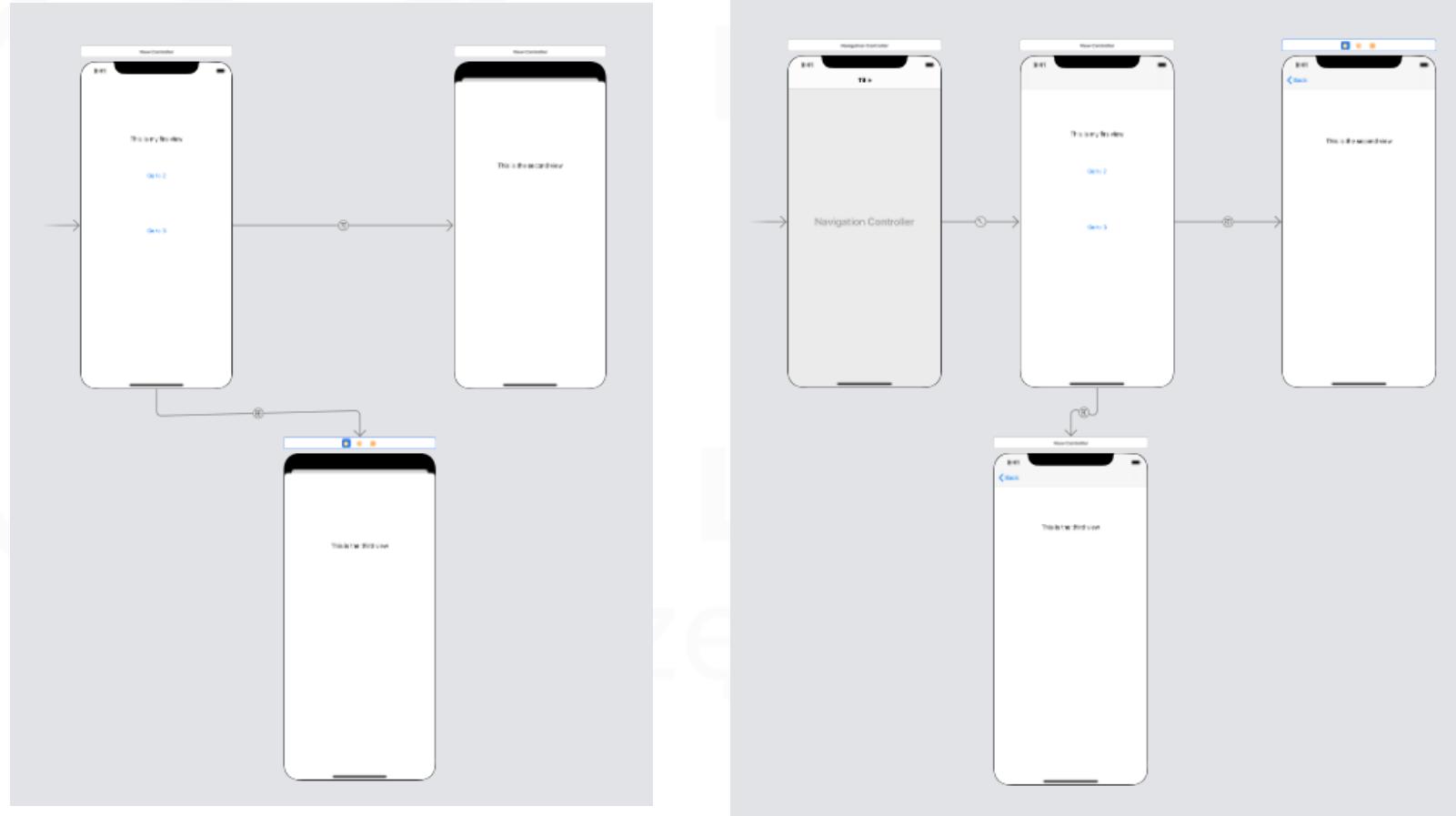
<https://developer.apple.com/library/archive/featuredarticles/ViewControllerPGforiPhoneOS/UsingSegues.html>

Aplikacja z wieloma widokami

- Aplikacja UIKit
 - tworzenie GUI
 - dodawanie kontrolera
 - tworzenie powiązań między widokami

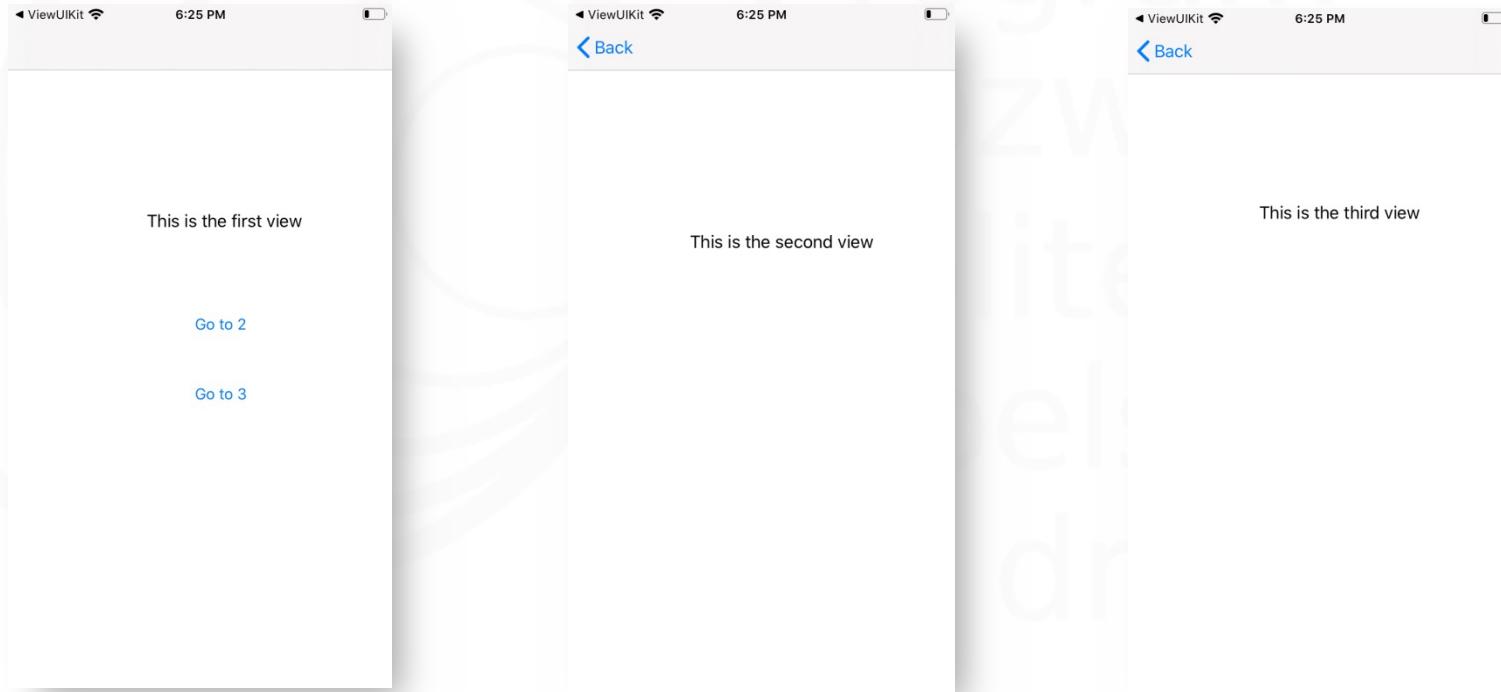


Aplikacja z wieloma widokami



Przykładowa aplikacja z wieloma widokami

- UIKit



Przykładowa aplikacja z wieloma widokami

- Nawigacja między widokami – SwiftUI
 - jeden z najważniejszych elementów aplikacji SwiftUI
 - pozwala na poruszanie się pomiędzy widokami
 - pozwala prezentować informacje w czytelny i hierarchiczny sposób
 - deklaracja

```
struct NavigationView<Content> where  
Content : View
```

Przykładowa aplikacja z wieloma widokami

- Nawigacja między widokami – SwiftUI

```
struct ContentView: View {  
    var body: some View {  
        NavigationView{  
            NavigationLink(destination:  
                Text("This is the second  
view!")){  
                Text("This is the first view")}  
            .navigationBarTitle("Navigation")  
        }  
    }  
}
```

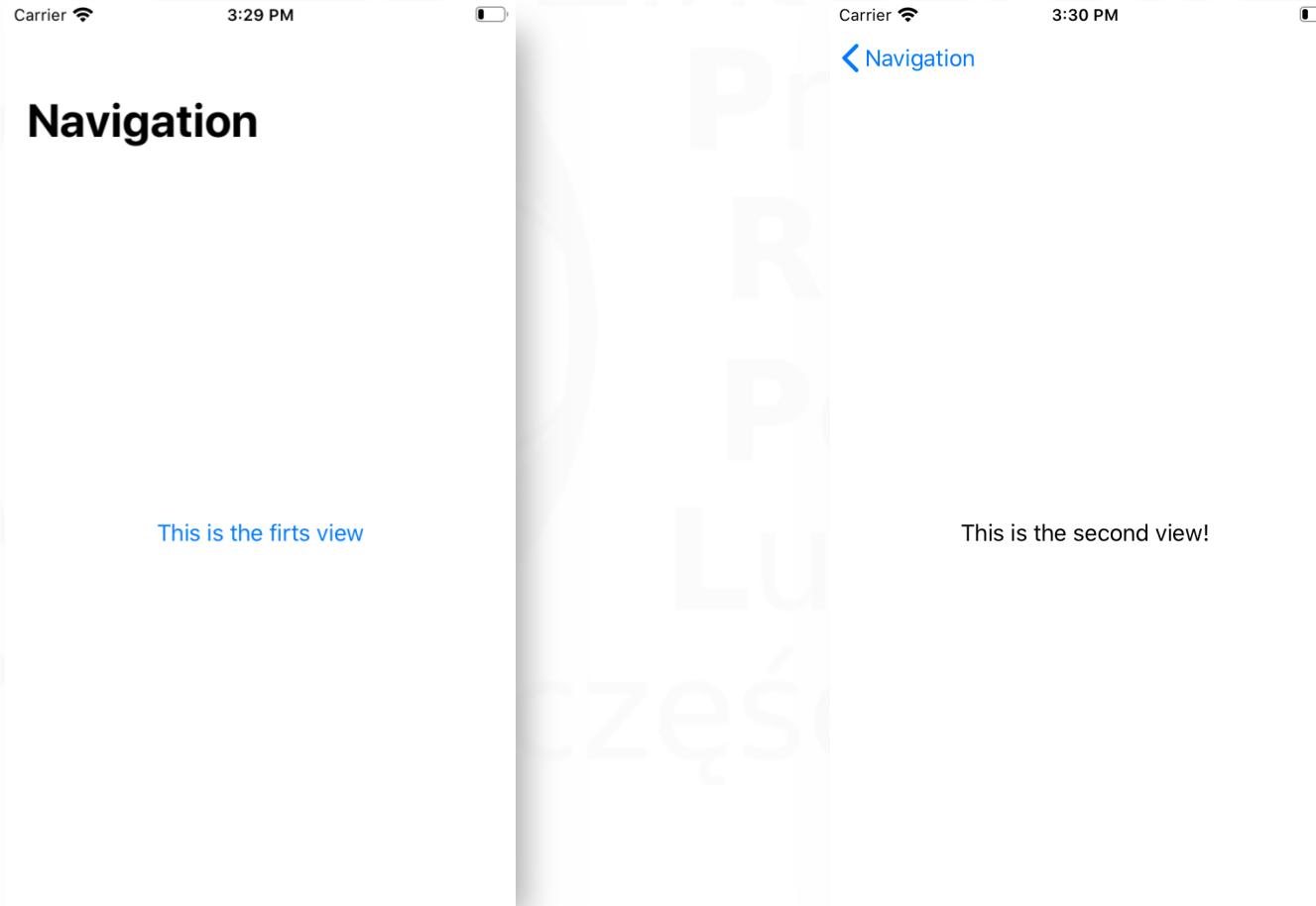


tekst 2.go widoku

tekst 1.go widoku

tytuł nawigacji

Przykładowa aplikacja z wieloma widokami

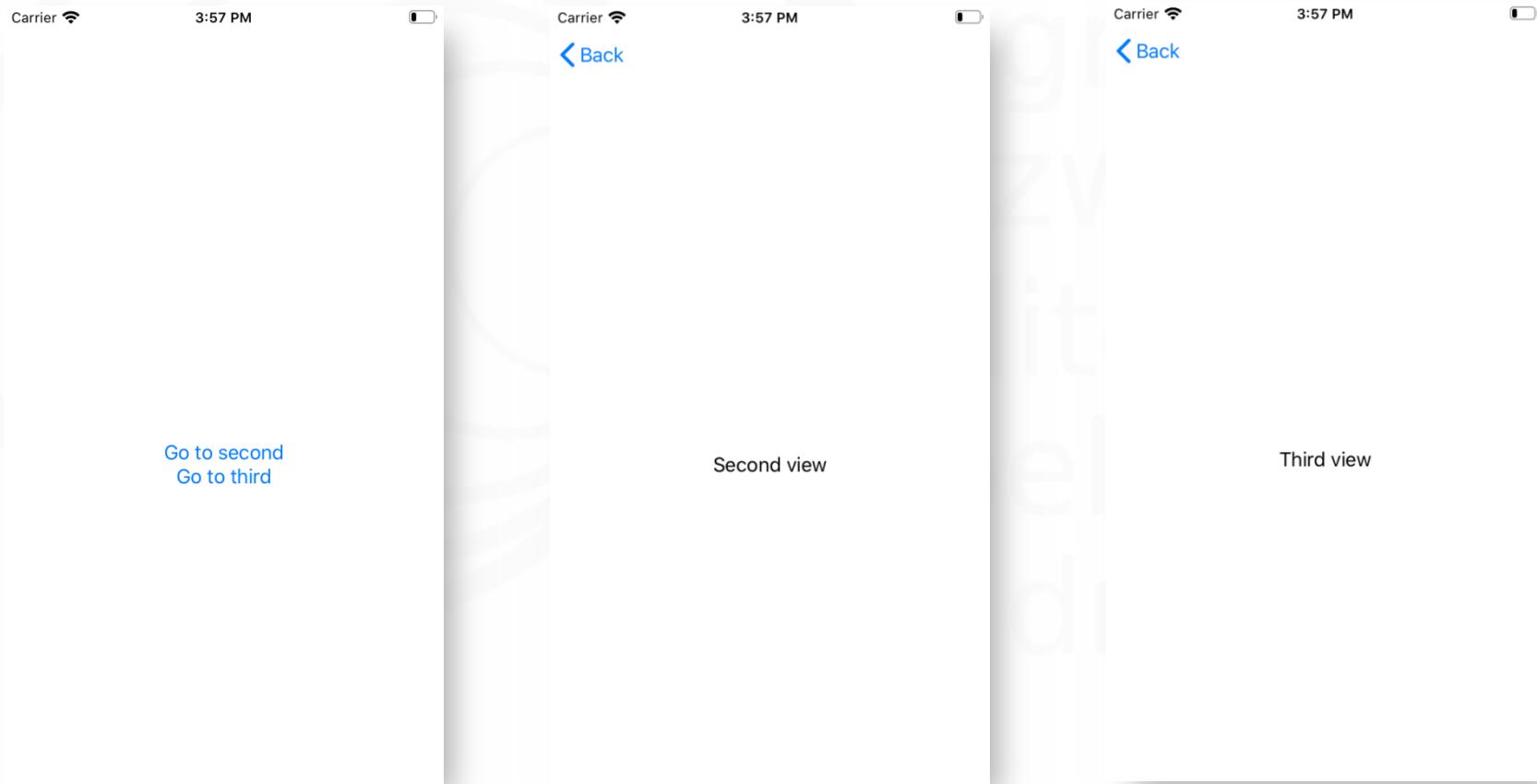


Przykładowa aplikacja z wieloma widokami

- SwiftUI – nawigacja przy pomocy przycisków

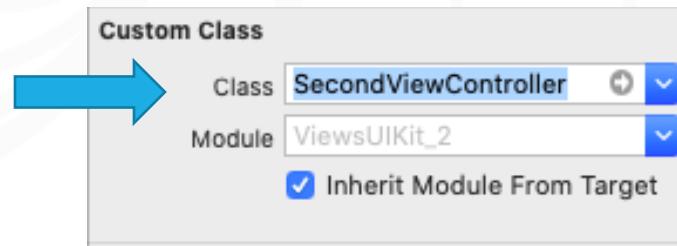
```
@State private var txt : String? = nil  
...  
NavigationView{  
    VStack{  
        NavigationLink(destination:  
            Text("Second view"), tag:  
            "Second", selection:$txt) {  
            EmptyView()  
        }  
        NavigationLink(destination:  
            Text("Third view"), tag:  
            "Third", selection:$txt) {  
            EmptyView() }  
    }  
    Button("Go to second") {  
        self.txt = "Second"  
    }  
    Button("Go to third") {  
        self.txt = "Third"  
    }  
}.navigationBarTitle("Navigation")
```

Przykładowa aplikacja z wieloma widokami



Przekazywanie danych między widokami

- UIKit
 - dodaj nową klasę: *Cocoa Touch Class*
File → New → File ..
nowa klasa dziedziczy po: *UIViewController*
 - przypisz odpowiednią klasę dla widoku w *Identity Inspector*



Przekazywanie danych między widokami

- UIKit
 - dodaj nową zmienną w *SecondViewController*
 - zmodyfikuj metodę *viewDidLoad()*

```
var text: String = ""  
@IBOutlet weak var lab: UILabel! ←  
  
override func viewDidLoad() {  
    super.viewDidLoad()  
    lab.text = text; ←  
}  
}
```

Przekazywanie danych między widokami

- UIKit
 - zaimplementowanie metody w głównym kontrolerze widoku:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?)  
{  
    if segue.identifier == "segue2"{ ←  
        let destinationViewContr = segue.destination as!  
            SecondViewController  
        destinationViewContr.text = myTxt.text ?? "none" ←  
    }  
}
```

Przekazywanie danych między widokami

- SwiftUI
 - dodanie nowej struktury dziedziczącej po klasie `View`
 - przekazywanie danych poprzez *property*



```
struct SecView: View {  
    let destName: String  
    var body: some View {  
        Text("Hello " + destName)  
    }  
}
```

Przekazywanie danych między widokami

- SwiftUI

- dodanie *NavigationView* oraz *NavLink*

```
var body: some View {
```

```
    NavigationView{
```

```
        VStack (alignment:.leading){
```

```
            Text("Welcom in the first view!")
```

```
            TextField("Input You name", text:$name)
```

```
        NavigationLink(destination: SecView(destName: self.name)){
```

```
            Text("Go to the second view")
```

```
        } }
```

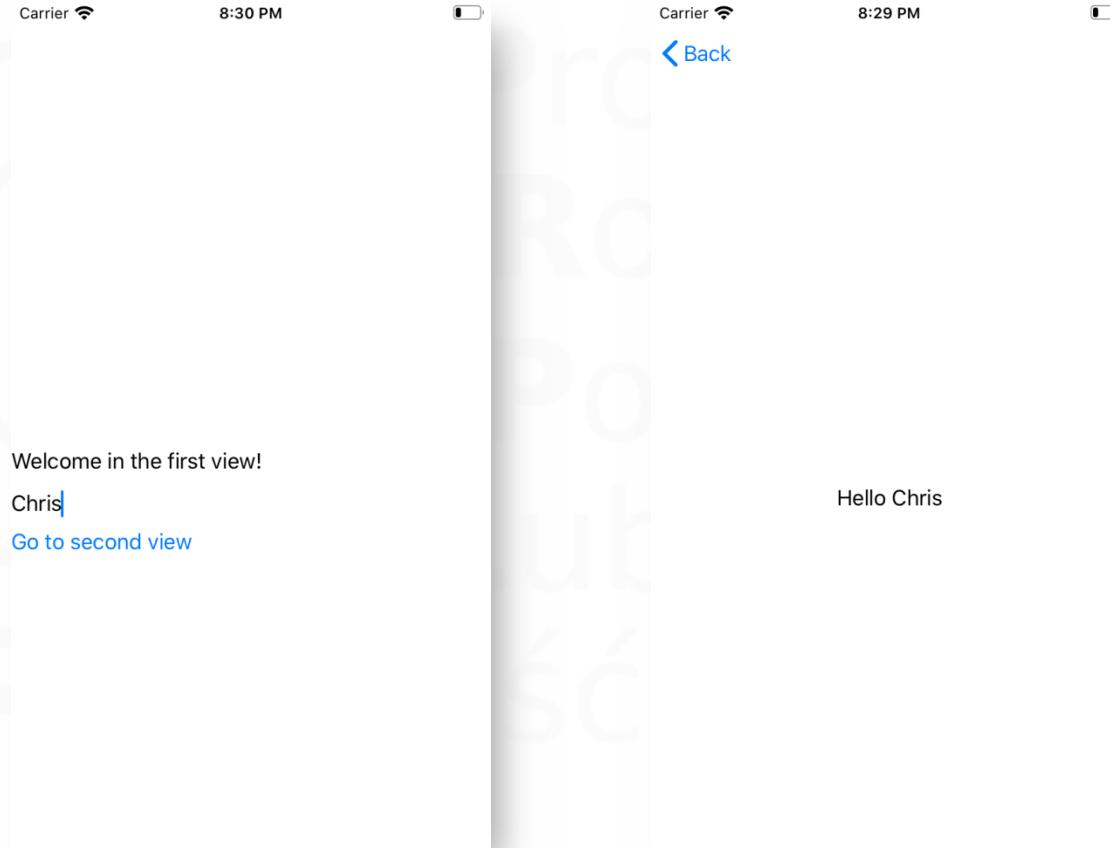
```
    }.navigationBarTitle("My navigation")
```

```
    }
```

```
}
```

Przekazywanie danych między widokami

- SwiftUI



Przekazywanie danych między widokami

- SwiftUI - `@Binding`
 - przekazanie danej wartości do innego widoku nie zawsze jest wystarczające
 - tworzenie powiązań między danymi pochodzących z różnych widoków
 - należy zastosować `@Binding` zamiast `@State` w drugim widoku
 - w pierwszym widoku należy użyć zmiennej z `@State` – jak tylko nastąpi zmiana wartości, zostanie uaktualniony widok
 - zmiana wartości w drugim widoku pozwoli na przekazanie jej do pierwszego

Przekazywanie danych między widokami

- SwiftUI - `@Binding` - `ContentView`

```
→ @State var isOn: Bool = true
    var body: some View {
        NavigationView{
            VStack{
                Rectangle()
                    .fill(isOn ? .yellow : .white)
                    .border(.green)
                    .frame(width: 100, height: 70, alignment: .center)
                NavigationLink(destination: RectangleView(isOn: $isOn), label:{ Text("Zarządzaj") })
            }
        }
    }
```

Przekazywanie danych między widokami

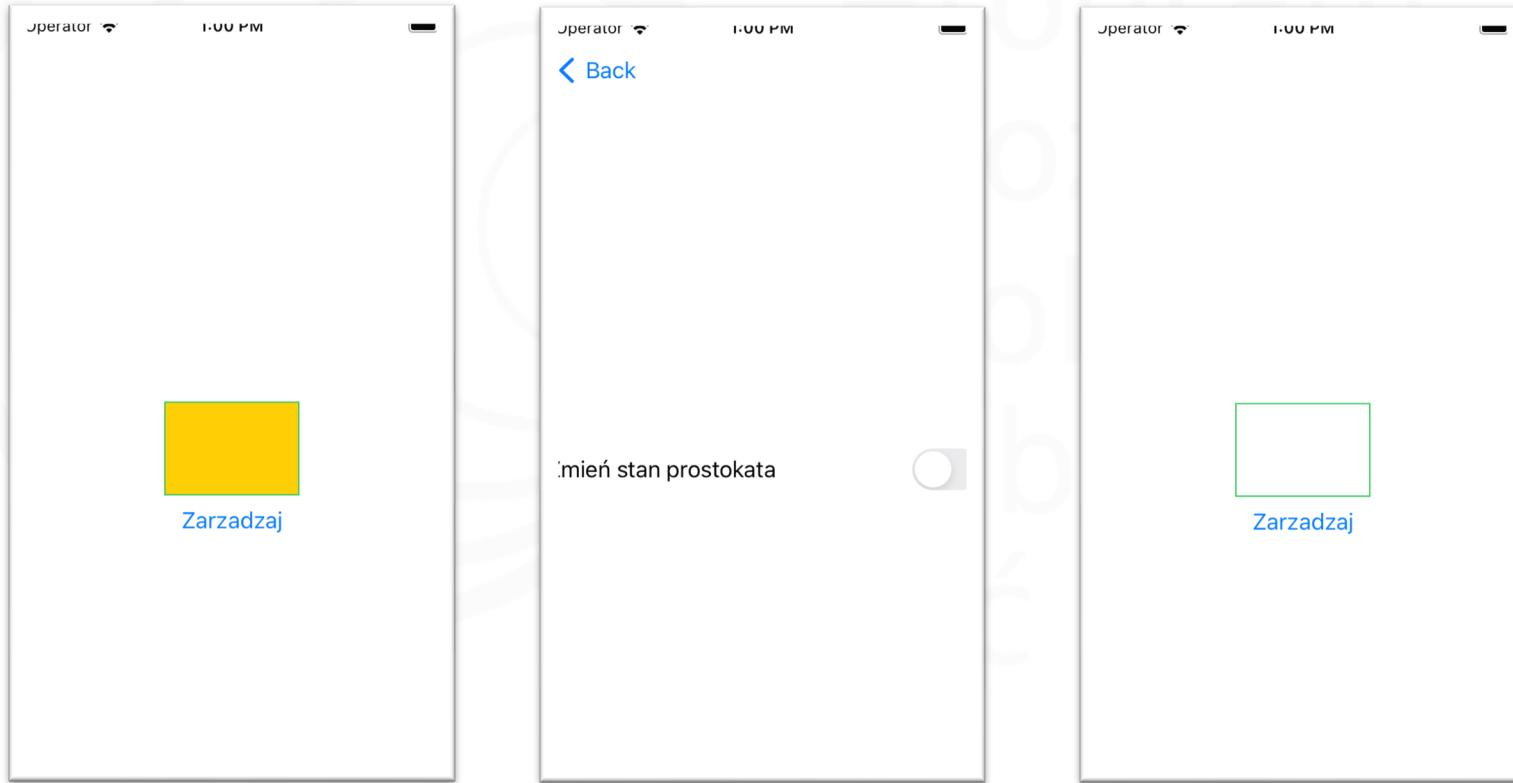
- SwiftUI - `@Binding` - `SecondView`

```
struct RectangleView: View {  
    @Binding var isOn: Bool  
    var body: some View {  
        Toggle("Zmień stan prostokata", isOn: $isOn)  
    }  
}
```

```
struct RectangleView_Previews: PreviewProvider {  
    static var previews: some View {  
        RectangleView(isOn: .constant(true))  
    }  
}
```

Przekazywanie danych między widokami

- SwiftUI - @Binding



Przekazywanie danych między widokami

- SwiftUI - `@EnvironmentObject`
 - umożliwia przekazywanie obiektu pomiędzy wieloma widokami oraz podwidokami
 - nie trzeba ręcznie przekazywać obiektu do każdego podwidoku.
 - przekazywanie obiektów zgodnie z hierarchią widoków
 - dzielenie danych pomiędzy widokami wykonywane jest jako:
 - wstawienie obiektu do hierarchii widoków za pomocą `.environmentObject(_:)`
 - dostęp do obiektu ze środowiska za pomocą `@EnvironmentObject`
 - do środowiska można przekazać tylko jeden obiekt każdego typu, więc nie można przekazać 2 obiektów (zostanie użyty pierwszy obiekt przekazany z `.environmentObject(_:)`)
 - działa tylko dla typów referencyjnych, tj. `ObservableObject` musi być klasą

Przekazywanie danych między widokami

- SwiftUI - `@EnvironmentObject`

```
class Person: ObservableObject {  
    @Published var name: String  
    @Published var age: Int
```



```
    init() {  
        self.name = "Ania"  
        self.age = 20  
    }
```

```
}
```

Przekazywanie danych między widokami

- SwiftUI - `@EnvironmentObject`

```
struct ContentView: View {  
    @StateObject var person = Person()  
    var body: some View {  
        NavigationView{  
            VStack{  
                Text("Hello \(person.name)")  
                .padding()  
                NavigationLink(destination:  
                    SecondView().environmentObject(person), label:{  
                        Text("Dane szczegółowe")})  
                NavigationLink(destination:  
                    EditView().environmentObject(person), label:{  
                        Text("Edytuj dane")})  
            }  
        }  
    }  
}
```

Przekazywanie danych między widokami

- SwiftUI - `@EnvironmentObject`

```
struct SecondView: View {  
     @EnvironmentObject var person: Person  
    var body: some View {  
        VStack{  
            Text("\(person.name) - \(person.age)")  
        }  
    }  
}
```

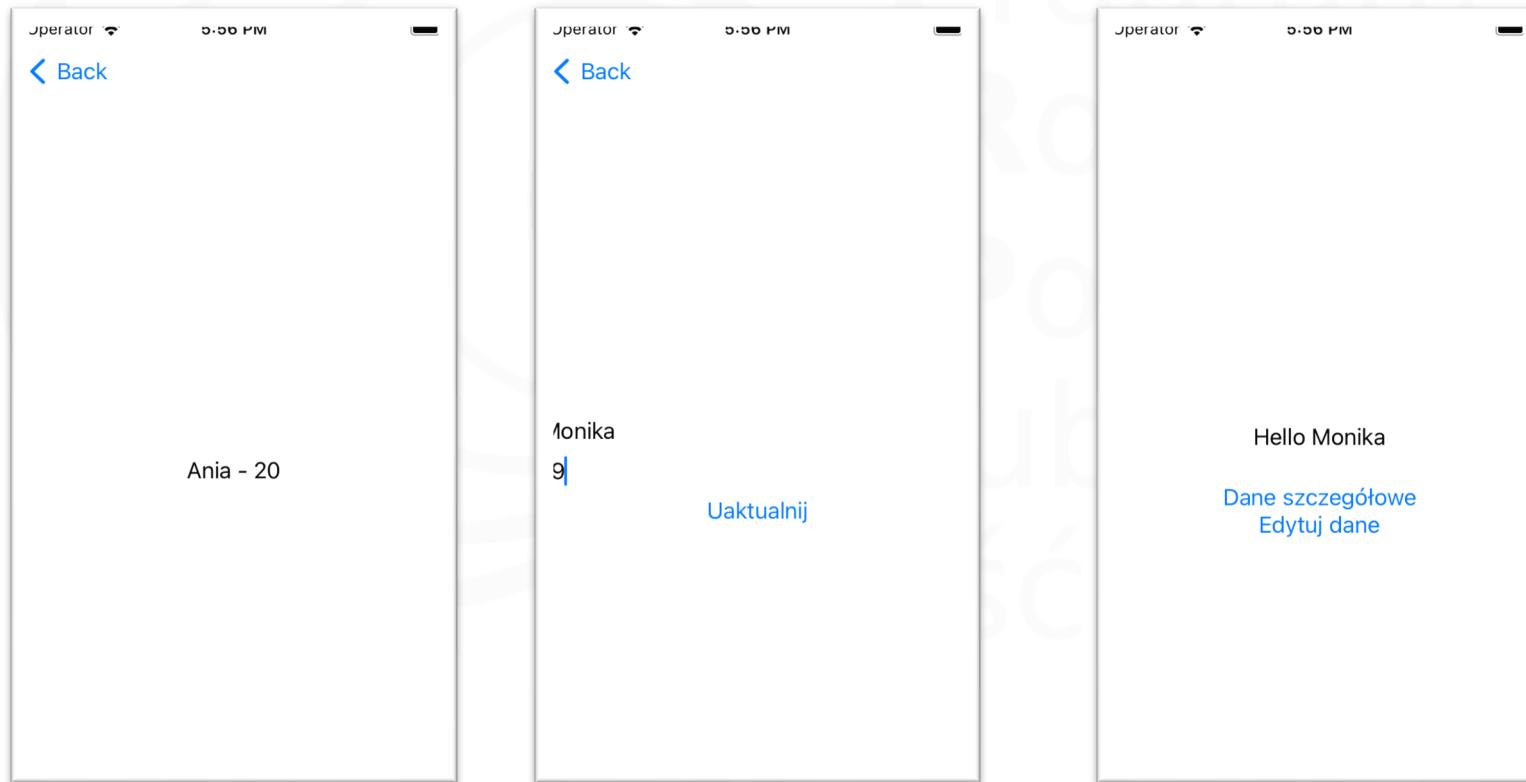
Przekazywanie danych między widokami

- SwiftUI - `@EnvironmentObject`

```
struct EditView: View {  
    @EnvironmentObject var person: Person  
    @State var newName: String = ""  
    @State var newAge: String = ""  
    var body: some View {  
        VStack{  
            TextField("Podaj nowe imię", text: $newName)  
            TextField("Podaj nowy wiek", text: $newAge)  
            Button(action: {  
                person.name = newName  
                person.age = (Int) (newAge) ?? 20  
            }, label: { Text("Uaktualnij") })  
        } } }
```

Przekazywanie danych między widokami

- SwiftUI - `@EnvironmentObject`



POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
INFORMATYKA



Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego



**Fundusze
Europejskie**
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny





Programowanie aplikacji mobilnych na platformę iOS

W8

Zastosowanie systemu kontroli wersji Git

Maria Skublewska-Paszkowska



**Fundusze
Europejskie**
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



Agenda

- Repozytorium kodu
- Repozytorium kodu – Git
- Xcode – tworzenie lokalnego repozytorium
- Xcode – tworzenie remote repozytorium

Repozytorium kodu

- Repozytorium kodu – dokument, który przechowuje całą historię zmian kodu
- Repozytorium kodu jest tworzone w celu korzystania z systemów wersjonowania
- Repozytorium kodu przechowuje kod źródłowy, podczas gdy oprogramowanie systemu wersji archiwizuje ten kod
- Można zarchiwizować wszystkie pliki w repozytorium, zachowując inne wersje lub pliki, nawet jeśli w danej chwili nie są używane
- Repozytoria kodu umożliwiają również nazywanie lub oznaczanie różnych wersji, przechowując zapisy zmian w ramach tego samego projektu

Repozytorium kodu

- Zalety repozytorium kodu:
 - wykrywanie zmian w plikach
 - update jest opisywany
 - historia zmian oraz ich autorzy są odpowiednio podani
 - możliwość znalezienia potencjalnie występujących problemów oraz przywrócenie kodu do poprzedniego stanu
 - możliwość tworzenia specjalnych wersji kodu opisujących konkretne funkcje - przeprowadzenie testów
 - obszerna dokumentacja kodu oraz implementacja konkretnych funkcji
 - wymiana informacji pomiędzy programistami
 - jeśli w projekt zaangażowanych jest kilka osób, zmiana tych samych plików nie spowoduje ich nadpisania
 - możliwość cofania zmian

Repozytorium kodu – Git

- Git – darmowy i rozproszony system kontroli wersji o otwartym kodzie źródłowym
- Jest dystrybuowany: każdy programista ma pełną historię swojego repozytorium kodu lokalnie
- Początkowy klon repozytorium jest wolniejszy, ale kolejne operacje (commit, blame, diff, merge oraz log) są szybsze niż w innych systemach kontroli wersji
- Włodzimierz Gajda "Git"

„ - Mały projekt po takiej wpadce da się jeszcze uratować, ale większy można wyrzucić do kosza. Chyba, że od momentu jego inicjalizacji używamy narzędzia odpowiedzialnego za właściwą synchronizację danych, czyli systemu kontroli wersji - podkreśla autor.

Repozytorium kodu – Git

- Podstawowe pojęcia:
 - *Commit* – zatwierdzenie do repozytorium zmian dokonanych w kodzie
 - *Branch* – odgałęzienie pozwalające na bezkolizyjną i równoczesną pracę programistów – każda osoba pracuje na swojej gałęzi i dopiero po zakończeniu pracy scalane są zmiany z innymi programistami pracującymi przy projekcie
 - *Tip* – najnowszy commit w danym branchu
 - *Checkout* – proces, w którym pobiera się pliki z repozytorium i tworzy kopie lokalną
 - *Merge* – scalanie zmian pochodzących z różnych źródeł – scalanie gałęzi pobocznych (branch) z główną gałęzią projektu (trunk)

Repozytorium kodu – Git

- Podstawowe pojęcia:

- *Conflict* – konflikt, do którego dochodzi w momencie, kiedy kilku programistów próbuje np. dokonać zmian w tym samym miejscu kodu
- *Update* – aktualizacja lokalnego repozytorium
- *Pull* – pobranie/wysłanie zmian z/do innego repozytorium
- *Fork* – kopia repozytorium
- *Working area* – miejsce zawierające zmiany w kodzie (w tym miejscu są widoczne wszystkie, jeszcze nie zatwierdzone zmiany)
- *Staging area* – miejsce, do którego dodawany jest kod, który później zostanie zatwierdzony

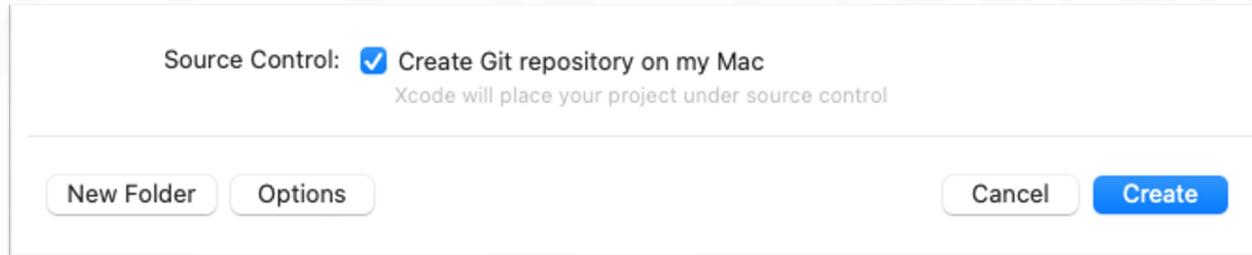
Repozytorium kodu – Git

- Zalety Git:

- rozproszona, odporną na usterki architektura sieci
- większą wydajność podczas pracy przy dużych projektach
- możliwość pracy offline
- wsparcie dla rozgałęzionego procesu tworzenia oprogramowania
- edycja wielu plików jednocześnie
- wsparcie dla protokołów sieciowych: HTTP(S), FTP, rsync, SSH
- scalanie gałęzi (ang. branch)
- popularny – korzysta z niego wielu programistów
- GitHub - usługa służącą do hostowania repozytoriów i udostępniania ich innym członkom zespołu

Xcode – tworzenie lokalnego repozytorium

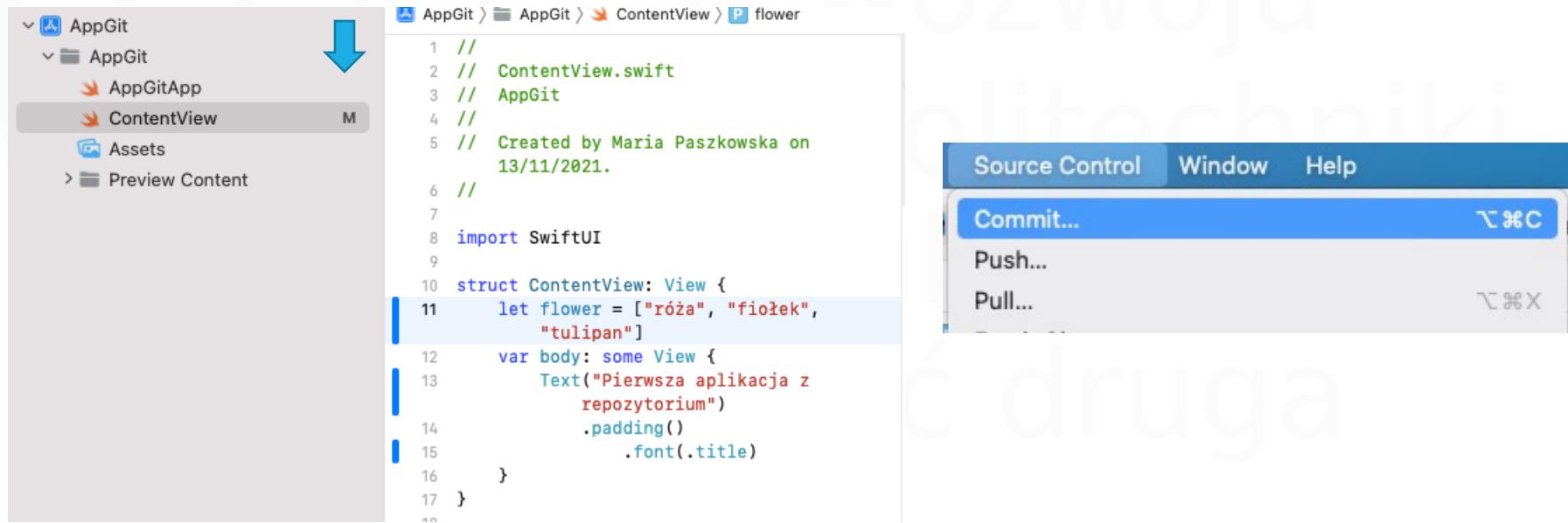
- Tworzone jest w katalogu projektu



- Inicjalizacja nowego repozytorium
- Zatwierdzenie (*Commit*) wszystkich plików utworzonych podczas tworzenia projektu

Xcode – tworzenie lokalnego repozytorium

- Zatwierdzenie zmian: *Source Control ->Commit*
 - M – zmodyfikowany plik
 - A – dodany plik



Xcode – tworzenie lokalnego repozytorium

- Zatwierdzenie zmian

```
1 //  
2 // ContentView.swift  
3 // AppGit  
4 //  
5 // Created by Maria Paszkowska on 13/11/2021.  
6 //  
7  
8 import SwiftUI  
9  
10 struct ContentView: View {  
11     var body: some View {  
12         Text("Hello, world!")  
13             .padding()  
14     }  
15 }  
16  
17 struct ContentView_Previews: PreviewProvider {  
18     static var previews: some View {  
19         ContentView()  
20     }  
21 }
```

```
1 //  
2 // ContentView.swift  
3 // AppGit  
4 //  
5 // Created by Maria Paszkowska on 13/11/2021.  
6 //  
7  
8 import SwiftUI  
9  
10 struct ContentView: View {  
11     let flower = ["róza", "fiołek", "tulipan"]  
12     var body: some View {  
13         Text("Pierwsza aplikacja z repozytorium")  
14             .padding()  
15             .font(.title)  
16     }  
17 }  
18  
19 struct ContentView_Previews: PreviewProvider {  
20     static var previews: some View {  
21         ContentView()  
22     }  
23 }  
24
```

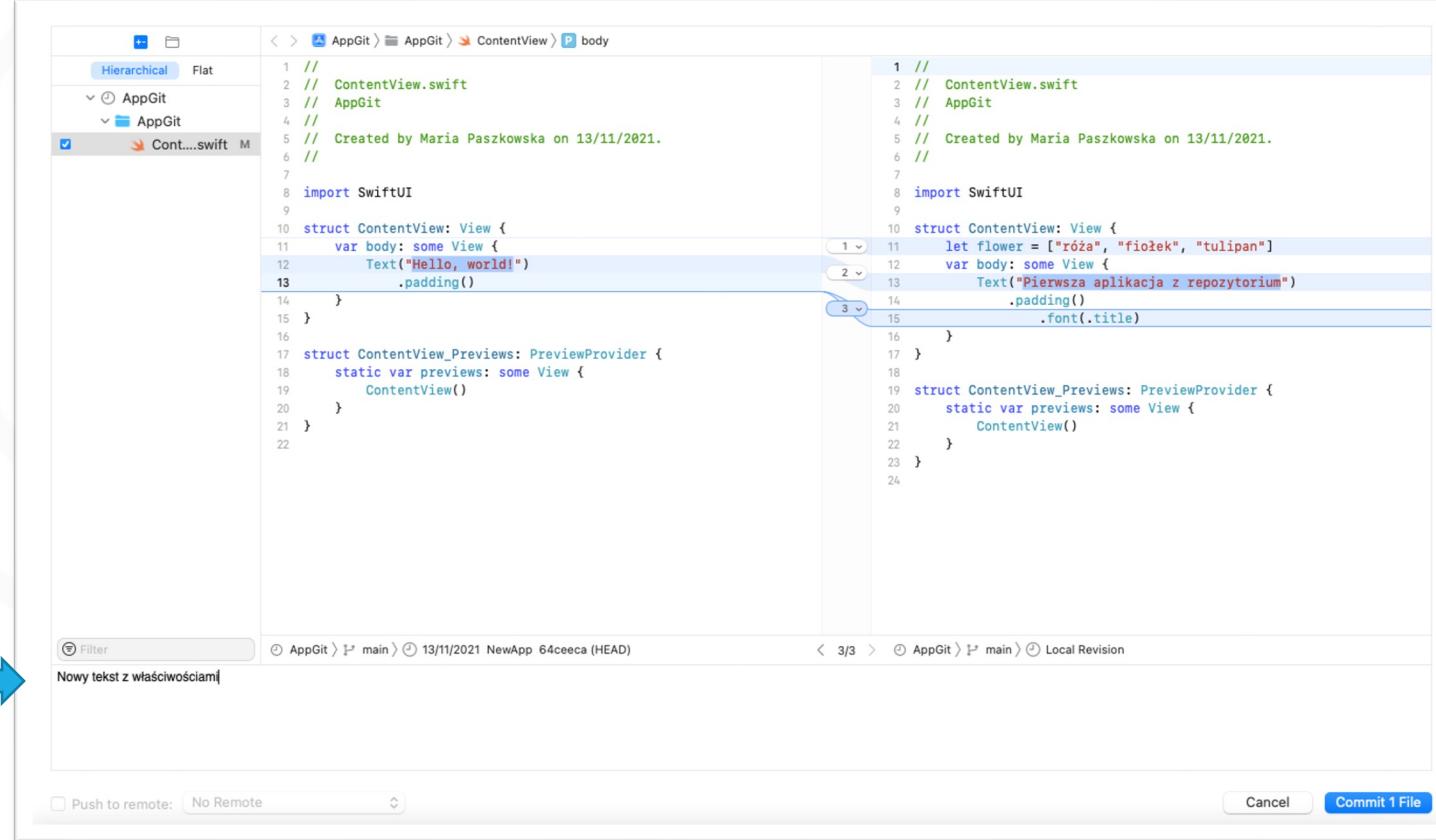
Enter commit message here

Push to remote: No Remote

Cancel Commit 1 File

Xcode – tworzenie lokalnego repozytorium

- Zatwierdzenie zmian



```
1 // ContentView.swift
2 // AppGit
3 // AppGit
4 //
5 // Created by Maria Paszkowska on 13/11/2021.
6 //
7
8 import SwiftUI
9
10 struct ContentView: View {
11     var body: some View {
12         Text("Hello, world!")
13             .padding()
14     }
15 }
16
17 struct ContentView_Previews: PreviewProvider {
18     static var previews: some View {
19         ContentView()
20     }
21 }
```

```
1 // ContentView.swift
2 // AppGit
3 // AppGit
4 //
5 // Created by Maria Paszkowska on 13/11/2021.
6 //
7
8 import SwiftUI
9
10 struct ContentView: View {
11     let flower = ["róża", "fiołek", "tulipan"]
12     var body: some View {
13         Text("Pierwsza aplikacja z repozytorium")
14             .padding()
15             .font(.title)
16     }
17 }
18
19 struct ContentView_Previews: PreviewProvider {
20     static var previews: some View {
21         ContentView()
22     }
23 }
```

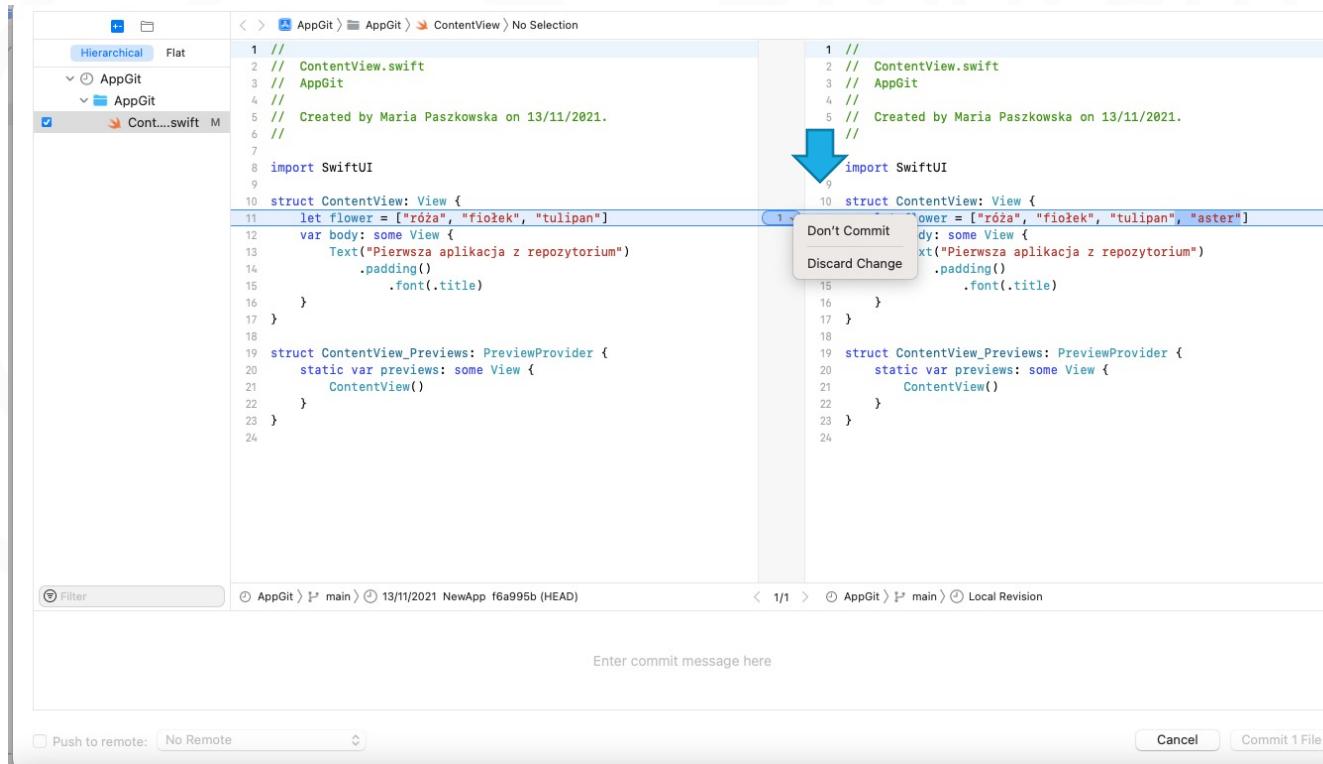
Nowy tekst z właściwościami:

Push to remote: No Remote

Cancel

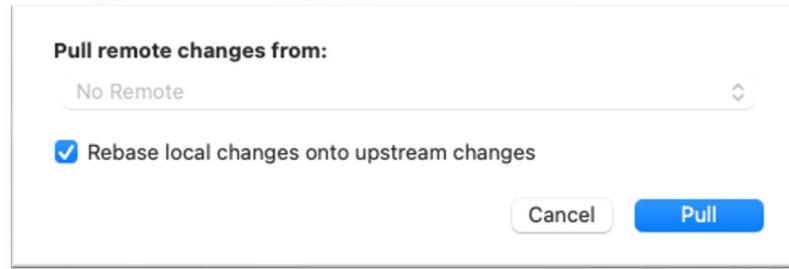
Xcode – tworzenie lokalnego repozytorium

- Zatwierdzenie zmian



Xcode – tworzenie lokalnego repozytorium

- Pobieranie wersji z repozytorium: *Source Control->Pull*

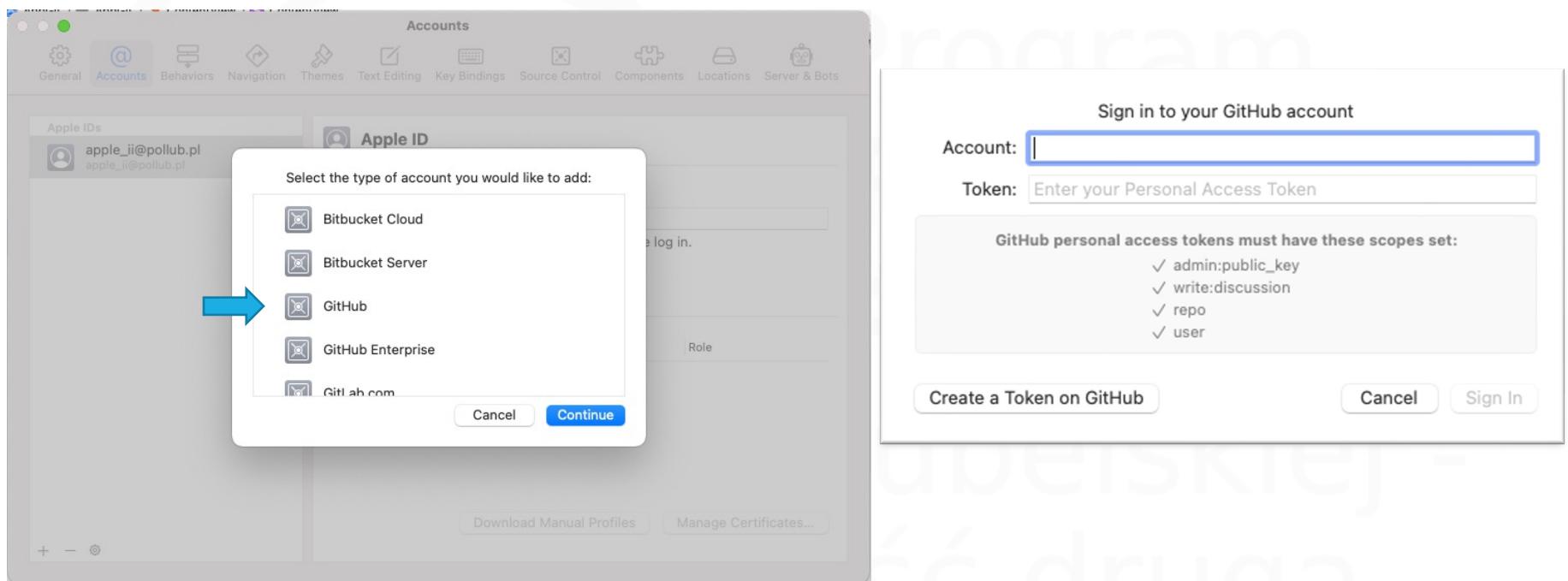


- Pobranie aktualnej wersji z repozytorium
- Utrata niezatwierdzonych zmian w lokalnym projekcie

Xcode – tworzenie remote repozytorium

- Można utworzyć klon istniejącego zdalnego repozytorium Git, aby zaktualizować i zsynchronizować zmiany
- Jeśli repozytorium do sklonowania, używa Bitbucket, GitHub lub GitLab jako hosta i wymaga uwierzytelnienia, należy skonfigurować konto w celu dostępu do repozytorium:
 - Xcode -> Preferencje -> Konta, kliknij przycisk Dodaj (+), wybierz typ konta do dodania i kliknij Kontynuuj

Xcode – tworzenie remote repozytorium



POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
INFORMATYKA



Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego



**Fundusze
Europejskie**
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny





Programowanie aplikacji mobilnych na platformę iOS

W7

Emulatory – zastosowanie i ograniczenia

Maria Skublewska-Paszkowska



**Fundusze
Europejskie**
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



Agenda

- Simulator iOS
- Dostępne operacje
- Ograniczenia

Zintegrowany
Program
Rozwoju
Politechniki
Lubelskiej -
część druga

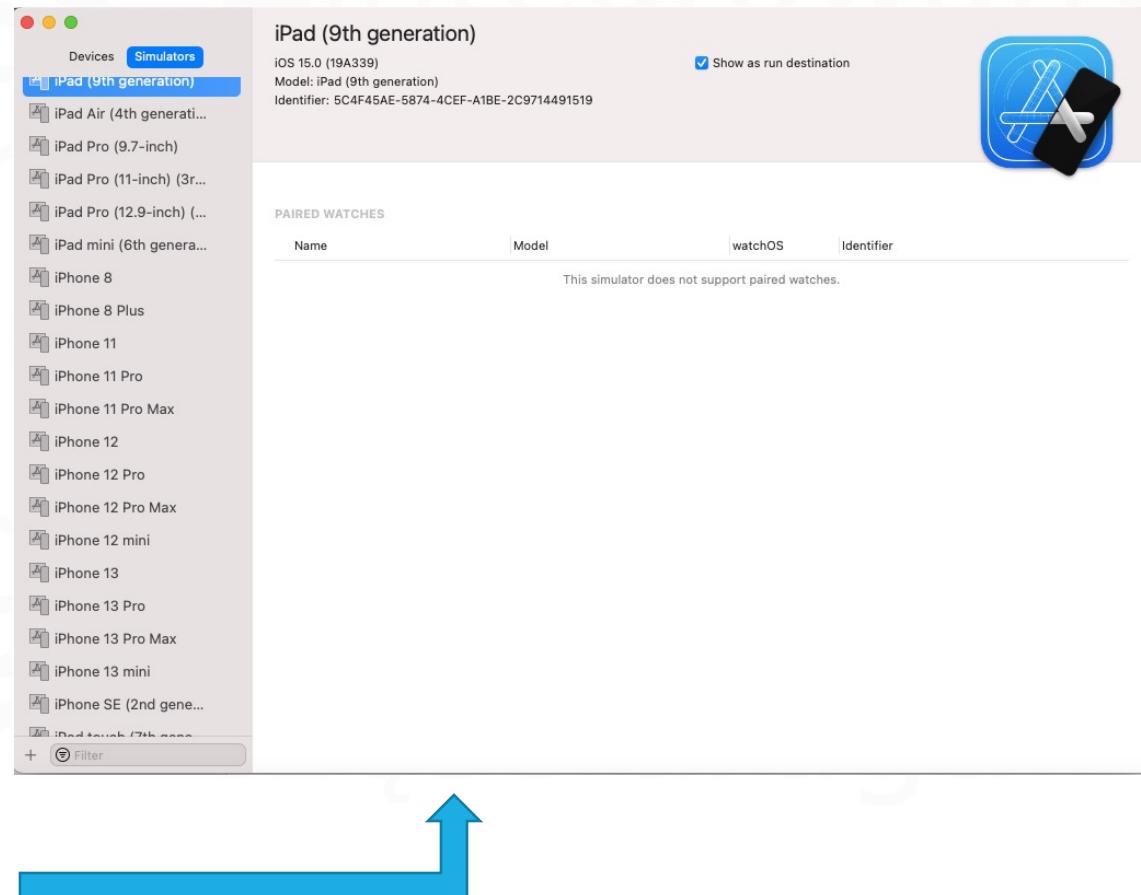
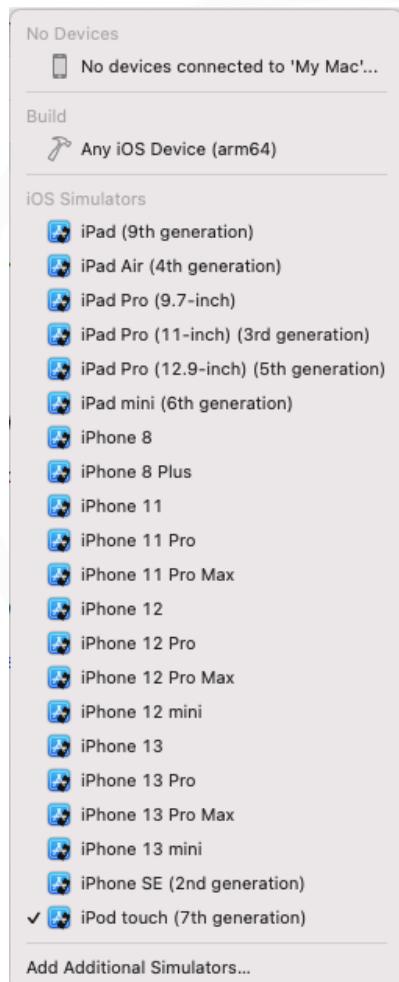
Symulator iOS

- Aplikacja Simulator, wbudowana jest w narzędzie Xcode
- Prezentuje interfejs użytkownika iPhone'a, iPada lub Apple Watch
- Interakcja z symulatorem odbywa się za pomocą klawiatury i myszy do emulowania stuknięć, obracania urządzenia i innych działań użytkownika
- Dostęp do symulatora można uzyskać poprzez Xcode, ale także można go uruchomić jako samodzielna aplikację
 - uruchamiając aplikację
 - Xcode -> Open Developer Tool -> iOS Simulator
 - ctr+click ikona Xcode w Dock -> Developer Tool > iOS Simulator

Symulator iOS

- Symulator stosuje się do:
 - lokalizowanie problemów oraz błędów aplikacji
 - testowanie przed instalacją na urządzeniu
 - zapoznania się ze środowiskiem iOS
- Safari jest stosowana do testowania
- Ustawienie lokalizacji:
 - Debug -> Location-> Custom Location
 - podanie współrzędnych latitude oraz longitude

Symulator iOS



Dostępne operacje

- Wybrane operacje:
 - wykonywanie zrzutów aplikacji
 - obrót symulatora (prawo/lewo): Cmd + strzałka (prawo/lewo)
 - gest wstrząśnięcia: Cmd + Ctrl + Z
 - przejście do ekranu głównego: Cmd + Shift + H
 - aktywowanie Siri: Cmd + Alt + Shift + H
 - ponowne uruchomienie urządzenia
 - symulowanie TouchID oraz FaceID
 - symulowanie ostrzeżenia o niewielkiej ilości dostępnej pamięci
 - symulowanie połączenia telefonicznego
 - wymuszenia użycia Force Touch
 - spreparowanie informacji o położeniu
 - zainicjalizowanie synchronizacji z iCloud

Ograniczenia

- Ograniczenia:
 - dostęp do pamięci komputera, a nie urządzenia mobilnego
 - nie powinien być stosowany do testowania pamięci
 - UI działa szybciej i efektywniej niż na urządzeniu mobilnym
 - sprzętowe:
 - akcelerometr
 - żyroskop
 - kamera
 - czujniki zbliżeniowe
 - wejście mikrofonu
 - brak GPS

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
INFORMATYKA



Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego



**Fundusze
Europejskie**
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny





Programowanie aplikacji mobilnych na platformę iOS

W6

Omówienie kontrolera i widoku

Maria Skublewska-Paszkowska



**Fundusze
Europejskie**
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



Agenda

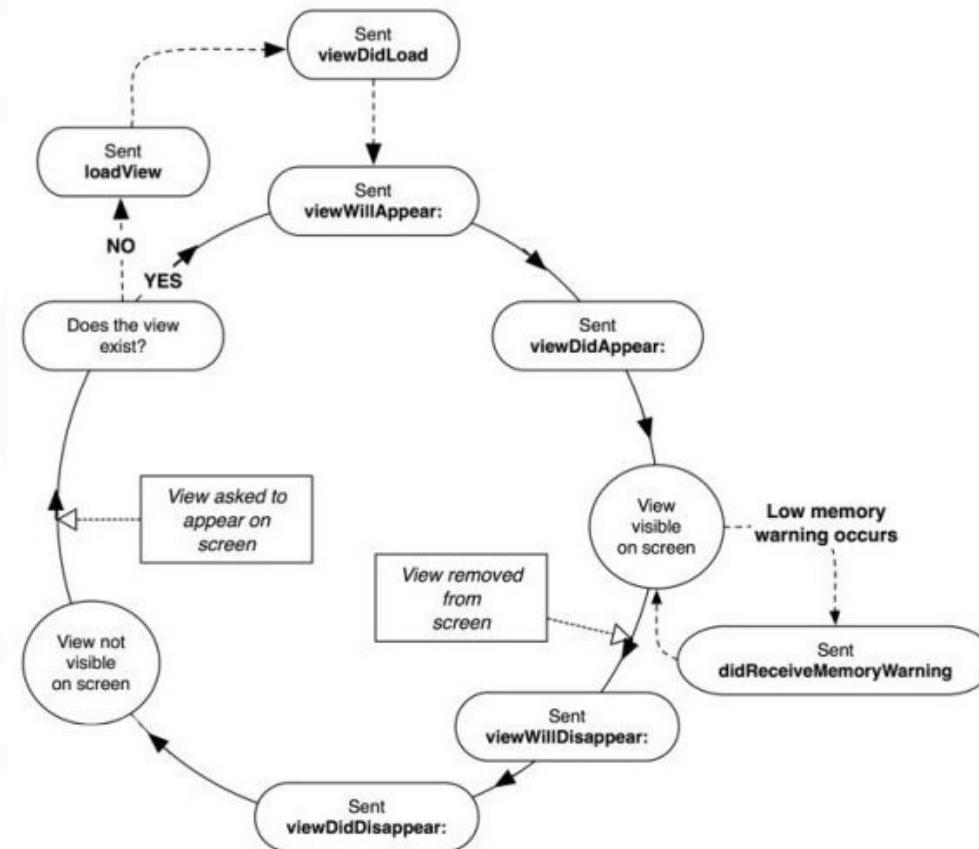
- UIKit – cykl życia
- Widok i kontroler – UIKit
- SwiftUI – cykl życia
- Widok i kontroler – SwiftUI

Zintegrowany
Program
Rozwoju
Politechniki
Lubelskiej -
część druga

UIKit – cykl życia aplikacji

- *UIView* – główna klasa dla wszystkich widoków, definiuje ich wspólne zachowanie, reprezentuje tylko ekran i pewną zawartość dla użytkownika
- *UIControl* – definiuje dodatkowe zachowania, które są specyficzne dla przycisków, przełączników i innych widoków zaprojektowanych dla interakcji użytkownika
- *UIViewController* – obsługuje *UIView*, informuje główny obiekt *UIView*, kiedy ma przejść do ekranu
- Aby widok działał poprawnie, należy wiedzieć, kiedy widok jest tworzony, wczytywany, pojawia się, jest zmieniany, czy kończony

UIKit – cykl życia aplikacji



źródło: <https://candost.blog/view-lifecycle-in-ios/>

UIKit – cykl życia aplikacji

- Metody:

- *loadView*: - należy nadpisać tę metodę, jeśli należy utworzyć niestandardowy widok i ustawić go na właściwość *view*
- *viewDidLoad*: - jest wywoływana tylko raz, po utworzeniu widoku i załadowaniu go do pamięci; metoda jest nadpisywana, aby zainicjować obiekty, których używa kontroler widoku (należy pamiętać o wywołaniu *super*)
- *viewWillAppear(_)*: - jest wywoływana tuż przed pojawieniem się widoku na ekranie; należy pamiętać, że metoda będzie wywoływana za każdym razem, gdy widok pojawi się na ekranie
- *viewDidAppear(_)*: - jest wywoływana zaraz po tym, jak widok jest widoczny dla użytkownika; można tutaj rozpoczęć animacje

UIKit – cykl życia aplikacji

- Metody:

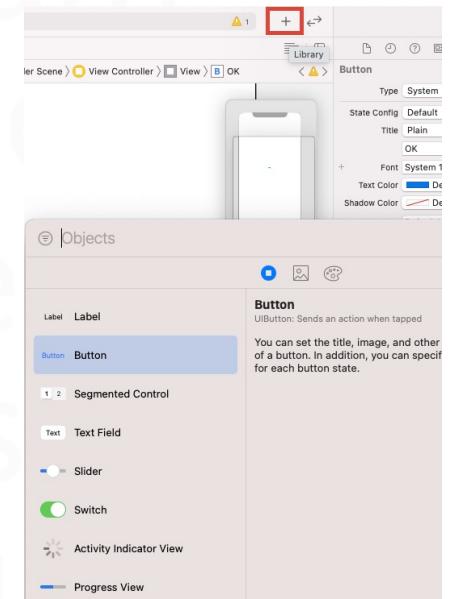
- *viewWillLayoutSubviews*: - pierwsze miejsce, w którym można poznać granice widoku w cyklu życia; jest wywoływana tuż przed metodą *layoutSubviews* w *UIView*; wywoływana jest również, gdy ładowane są podwidoki widoku głównego (np. komórki widoku kolekcji)
- *viewDidLayoutSubviews*: - jest wywoływana zaraz po wywołaniu *layoutSubviews*; widoki podrzędne zostały ustawione i zastosowano rozmiar, położenie i ograniczenia
- *viewWillDisappear(_)*: - jest wywoływana, gdy widok ma zniknąć z ekranu, należy wtedy zapisać dane użytkownika, aby nie stracić niczego ważnego, anulować żądania sieciowe.
- *viewDidDisappear(_)*: - jest wywoywana, gdy widok znika z ekranu, widok jest obecnie usuwany z hierarchii widoków

Widok i kontroler – UIKit

- Widoki i kontrolki to wizualne elementy konstrukcyjne interfejsu użytkownika aplikacji
- Stosowane do rysowania i porządkowania zawartości aplikacji na ekranie
- Widoki mogą zawierać inne widoki – hierarchia ułatwia zarządzanie widokami:
 - superwidok (ang. superview)
 - podwidok (ang. subview)
- Czynności:
 - reakcje na dotknięcia i inne zdarzenia (bezpośrednio lub we współpracy z urządzeniami rozpoznawania gestów)
 - rysowanie
 - obsługa interakcji typu „przeciagnij i upuść”
 - reakcje na zmiany ostrości
 - animacje atrybutu rozmiaru, położenia i wyglądu widoku

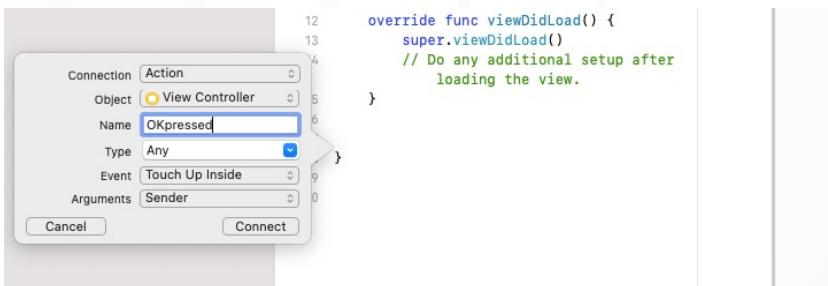
Widok i kontroler – UIKit

- Kontrolki (*UIControls*) implementują elementy, takie jak przyciski i suwaki, których aplikacja może używać do:
 - ułatwiania nawigacji
 - zbierania danych wejściowych od użytkownika
 - manipulowania zawartością danych
- *class UIControl : UIView*
- Kontrolki mogą znajdować się w jednym z kilku stanów, które definiuje typ *UIControl.State*
- Stan kontrolki można zmienić programowo lub za pomocą Storyboard:
 - domyślny
 - podświetlony
 - wybrany
 - wyłączony

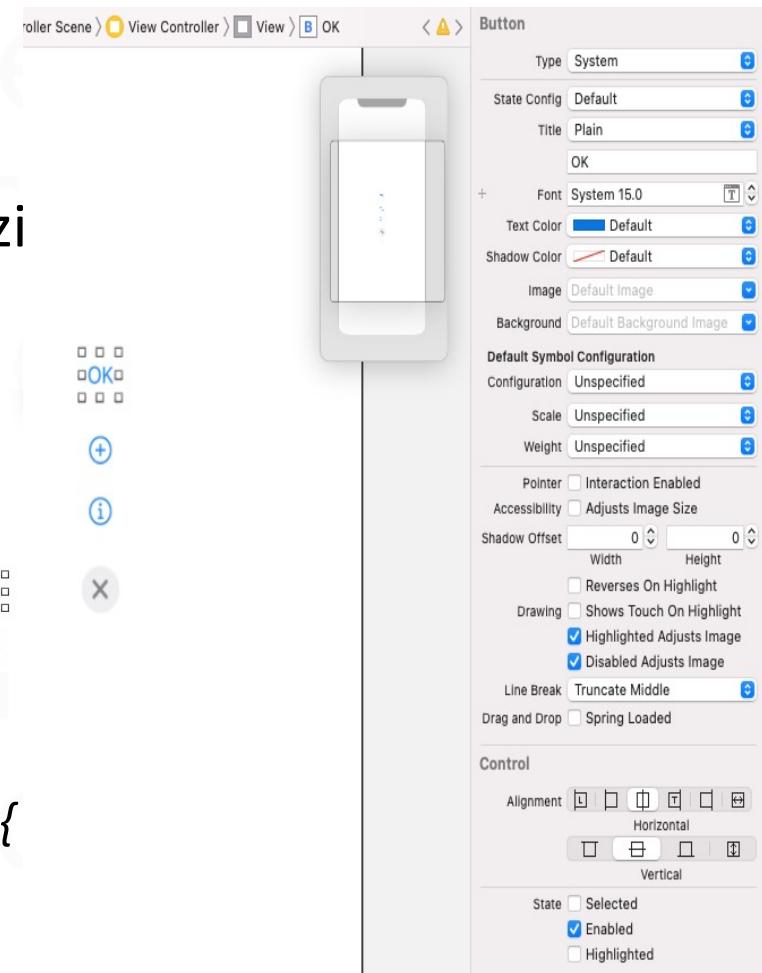


Widok i kontroler – UIKit

- Przycisk (*UIButton*) – wykonuje niestandardowy kod w odpowiedzi na interakcje użytkownika

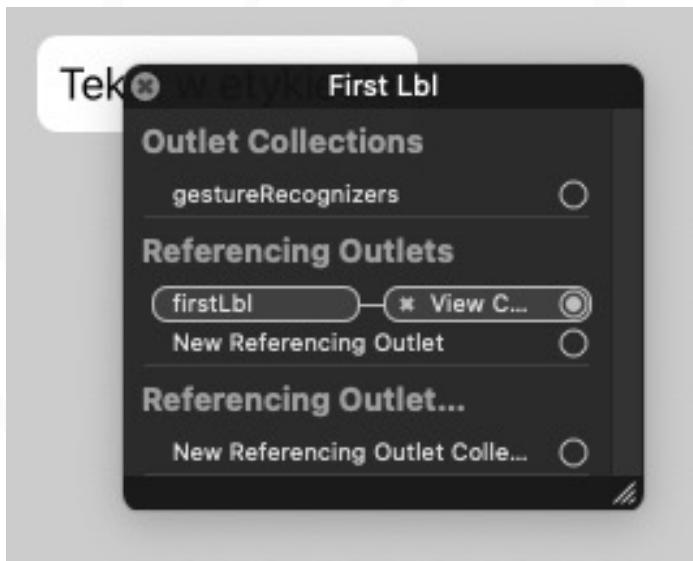


```
@IBAction func OKpressed(_ sender: Any) {
    //obsługa przycisku
}
```

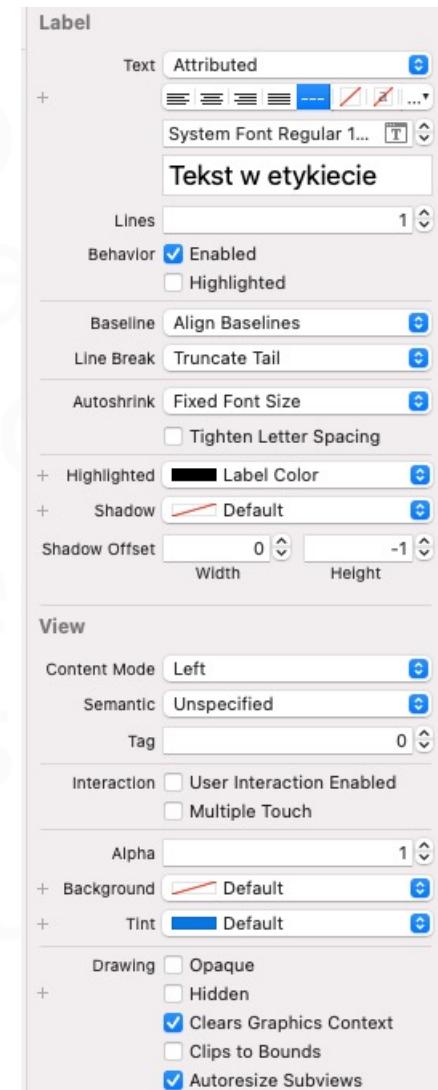


Widok i kontroler – UIKit

- Etykieta (*UILabel*) – tekst wyświetlony w postaci etykiety

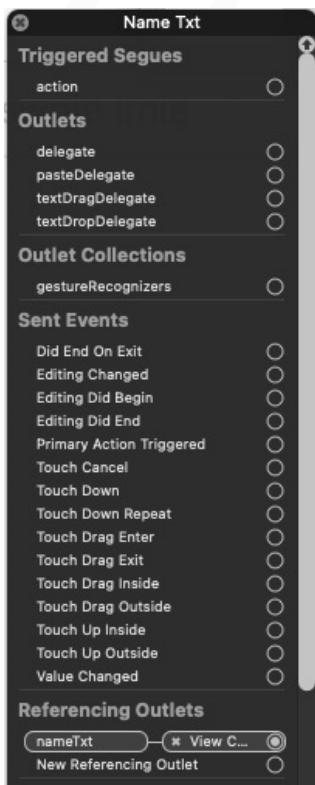


`@IBOutlet weak var firstLbl: UILabel!`

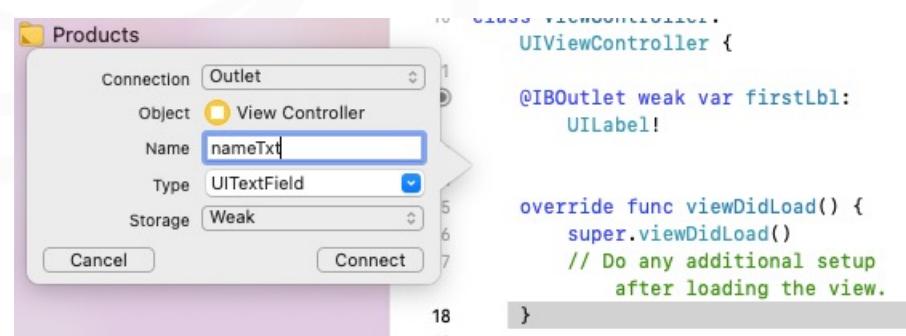


Widok i kontroler – UIKit

- Pole tekstowe (*UITextField*) – wyświetla edytowalny tekst

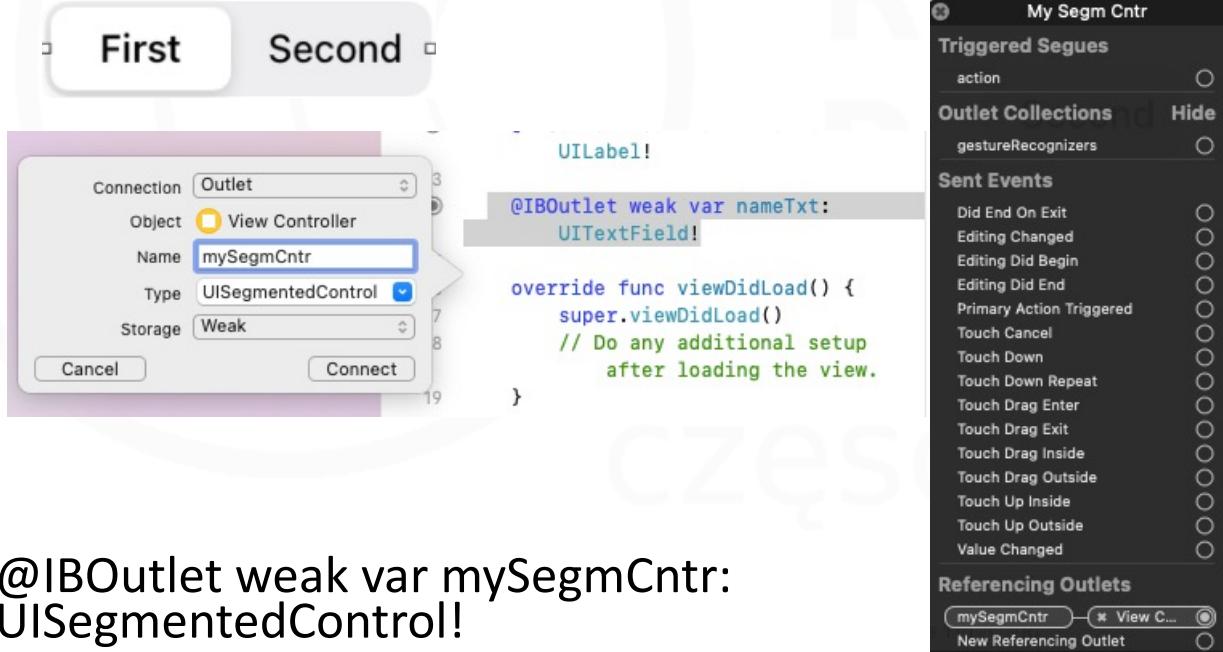


@IBOutlet weak var nameTxt:
UITextField!



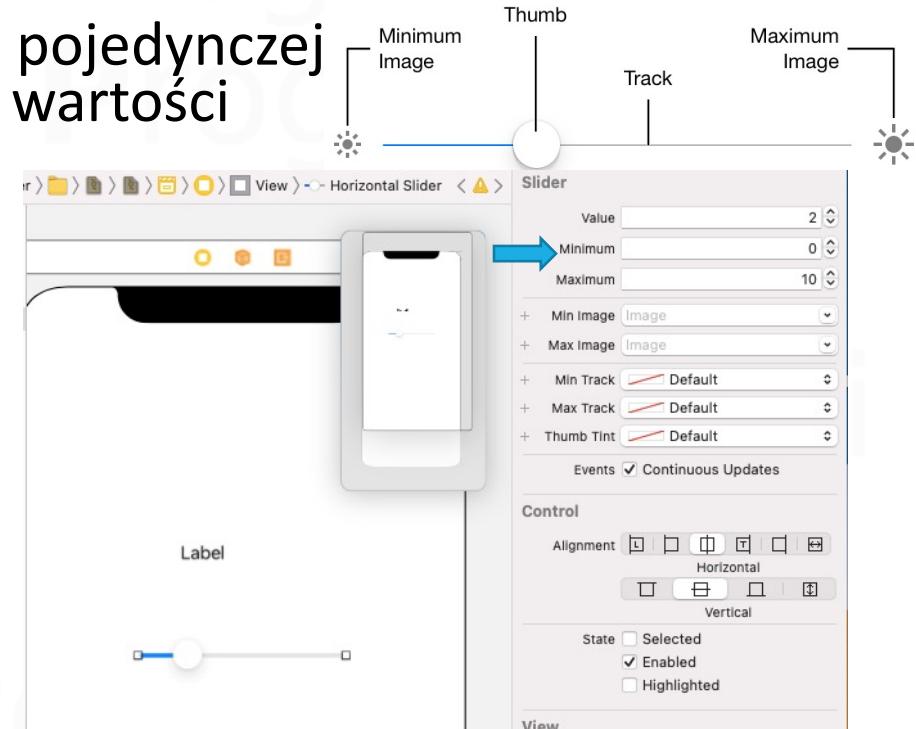
Widok i kontroler – UIKit

- Kontrolka segmentowe (*UISegmentedControl*) – składa się z wielu segmentów, z których każdy działa jako dyskretny przycisk



Widok i kontroler – UIKit

- Slider (*UISlider*) – wybranie pojedynczej wartości z ciągłego zakresu wartości



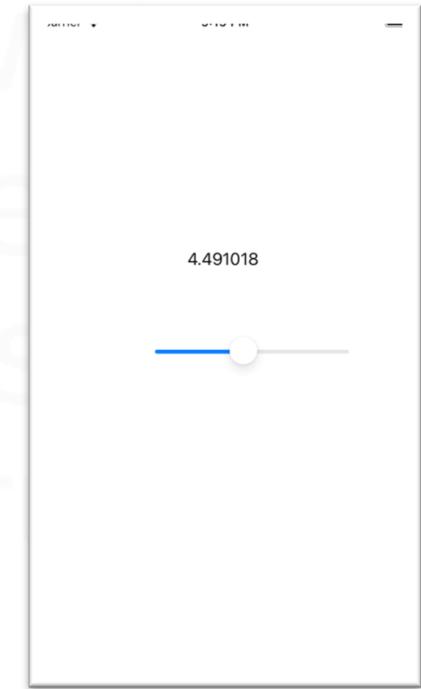
@IBOutlet weak var mySlider: UISlider!

źródło: <https://developer.apple.com/documentation/uikit/uislider>

Widok i kontroler – UIKit

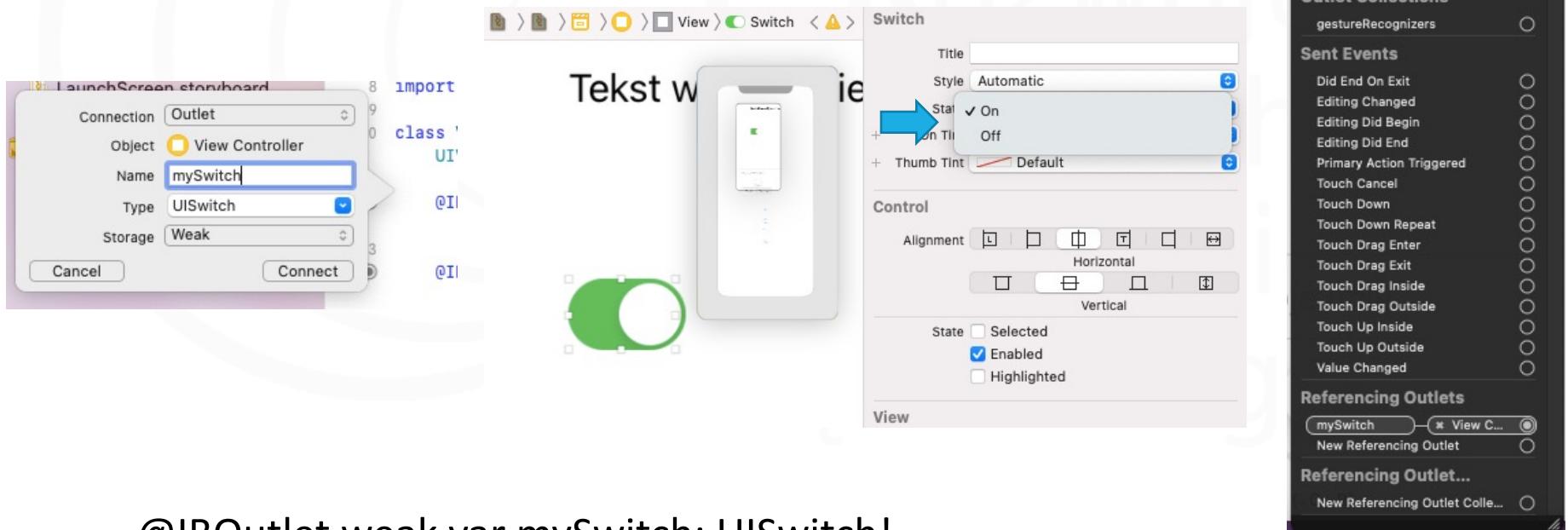
- Slider:
 - powiadomienia o zmianie wartości suwaka – metoda *valueChanged*
 - ustawienie początkowej wartości wyświetlonej w etykiecie

```
@IBAction func valueChanged(_ sender: Any){  
    myLbl.text = String(mySlider.value)  
}
```



Widok i kontroler – UIKit

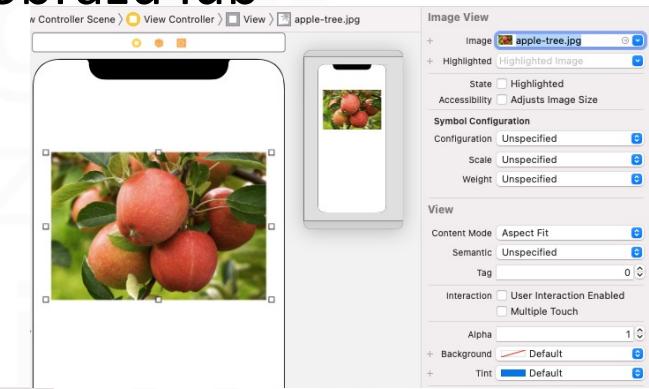
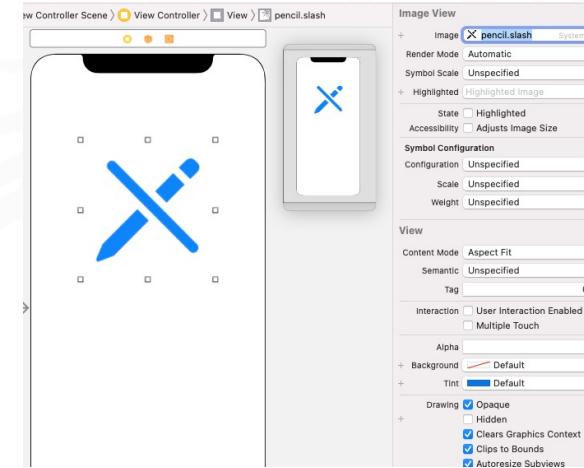
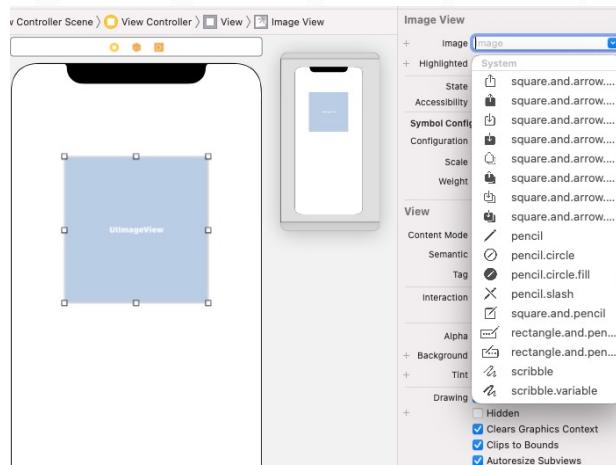
- Switch (*UISwitch*) – wybór binarny, np. włączanie/wyłączanie
 - *var isOn: Bool*
 - *func setOn(Bool, animated: Bool)*



@IBOutlet weak var mySwitch: UISwitch!

Widok i kontroler – UIKit

- **Image (*UIImageView*)** – wyświetlenie obrazu lub sekwencji obrazów (np. JPEG, PNG)
 - *init(named:in:compatibleWith:)*
 - *imageWithContentsOfFile:*

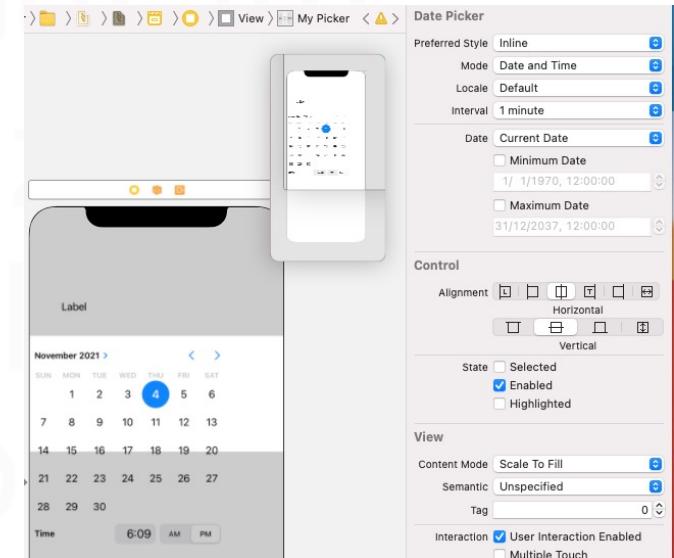
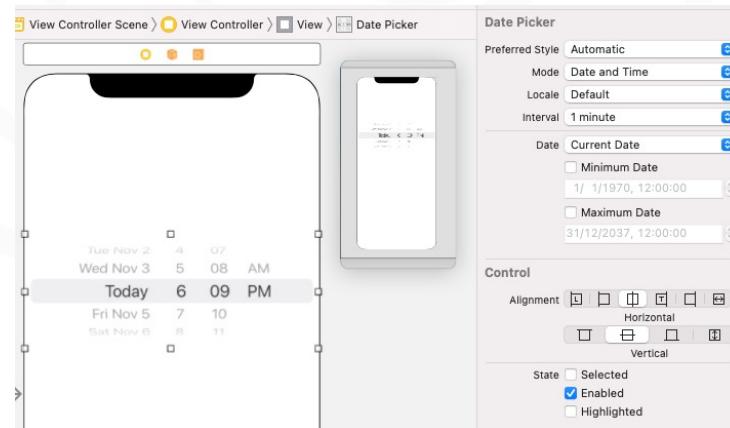


Widok i kontroler – UIKit

- Wybór daty (*UIDatePicker*) – wyświetlenie czasu i/lub daty oraz przedziału czasu
 - minimalna/maksymalna data
 - przedział czasowy

@IBOutlet weak var myLbl: UILabel!

@IBOutlet weak var myPicker: UIDatePicker!

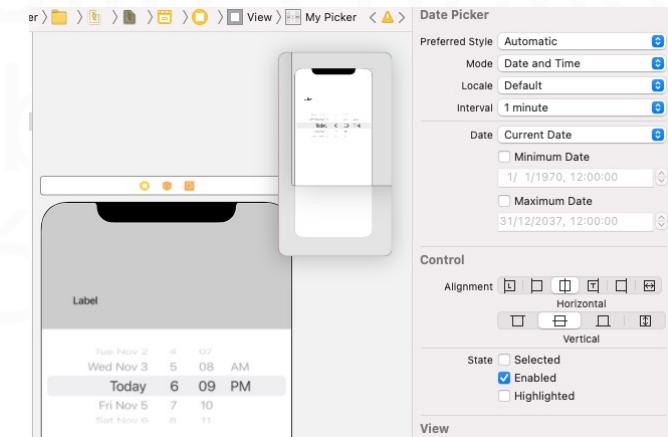
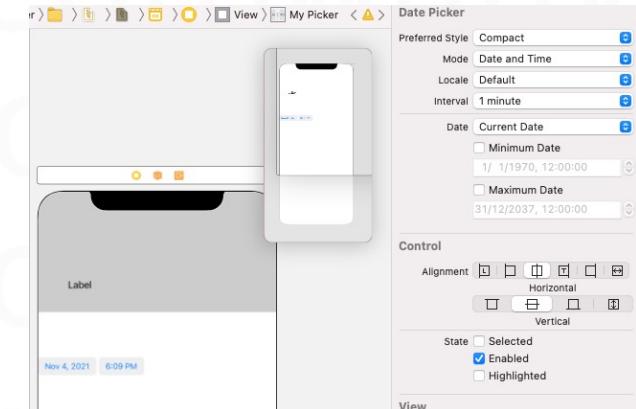


Widok i kontroler – UIKit

- Wybór daty (*UIDatePicker*)

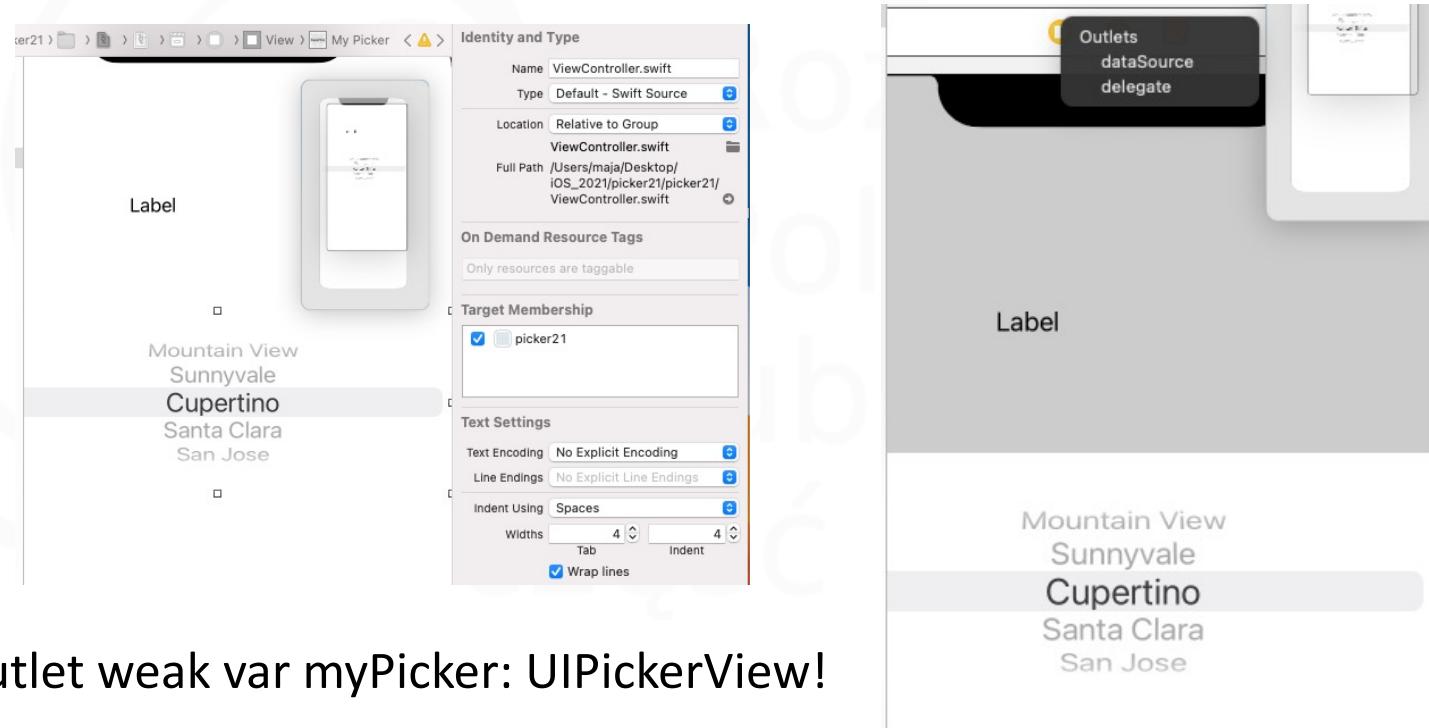
```
override func viewDidLoad() {  
    super.viewDidLoad()  
    myPicker.setDate(Date(), animated: false)  
}
```

```
@IBAction func update(_ sender: Any) {  
    let format = DateFormatter()  
    format.dateFormat = "yyyy-mm-dd"  
    let str = format.string(from: myPicker.date)  
    myLbl.text = str  
    print(str)  
}
```



Widok i kontroler – UIKit

- Lista (*UIPickerView*) – wyświetlenie elementów w postaci koła w celu wybrania jednej wartości



@IBOutlet weak var myPicker: UIPickerView!

Widok i kontroler – UIKit

- Lista (*UIPickerView*)

- dane do wyświetlenia

```
var list: [String] = [String]()
override func viewDidLoad() {
    super.viewDidLoad()
    list = ["komputer", "laptop", "tablet"]
}
```

- class ViewController: UIViewController, *UIPickerViewDelegate*, *UIPickerViewDataSource*

- liczba komponentów

```
func numberOfComponents(in pickerView: UIPickerView)
-> Int {
    return 1
}
```

Widok i kontroler – UIKit

- Lista (*UIPickerView*)

- liczba elementów do wyświetlenia

```
func pickerView(_ pickerView: UIPickerView,  
    numberOfRowsInComponent component: Int) -> Int {  
    return list.count  
}
```

- wyświetlenie elementów

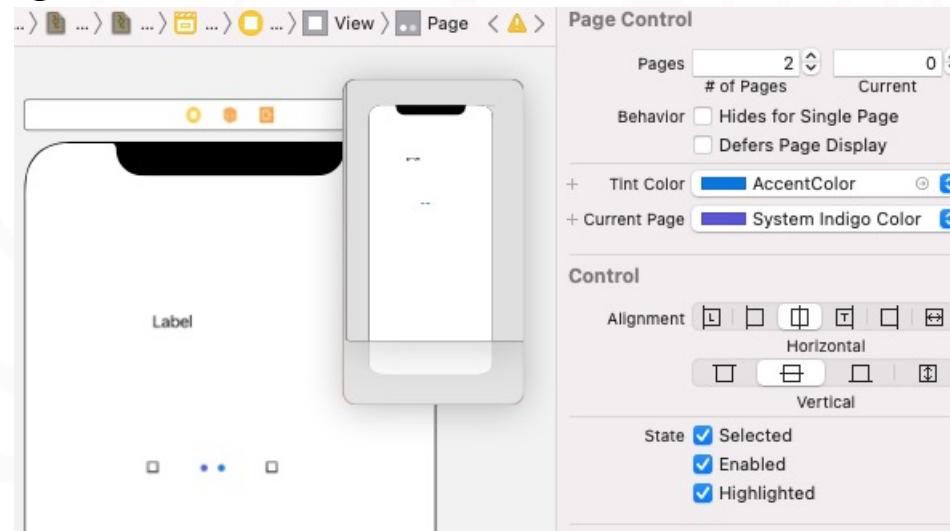
```
func pickerView(_ pickerView: UIPickerView, titleForRow  
    row: Int, forComponent component: Int) -> String? {  
    return list[row]  
}
```

- zaznaczenie elementów

```
func pickerView(_ pickerView: UIPickerView, didSelectRow row:  
    Int, inComponent component: Int) {  
    myLbl.text = list[row]  
}
```

Widok i kontroler – UIKit

- Kontrola stron (*UIPageControl*) - wyświetlenie poziomej serii kropek, z których każda odpowiada stronie w dokumencie aplikacji
 - *UIPageControl : UIControl*

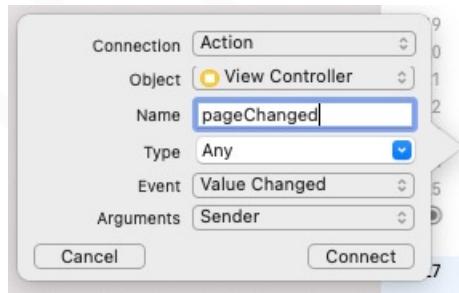


@IBOutlet weak var page: UIPageControl!

Widok i kontroler – UIKit

- Kontrola stron (*UIPageControl*)
 - *valueChanged*
 - *currentPage*

```
var animal: [String] = ["pies",  
"kot", "mysz"]
```



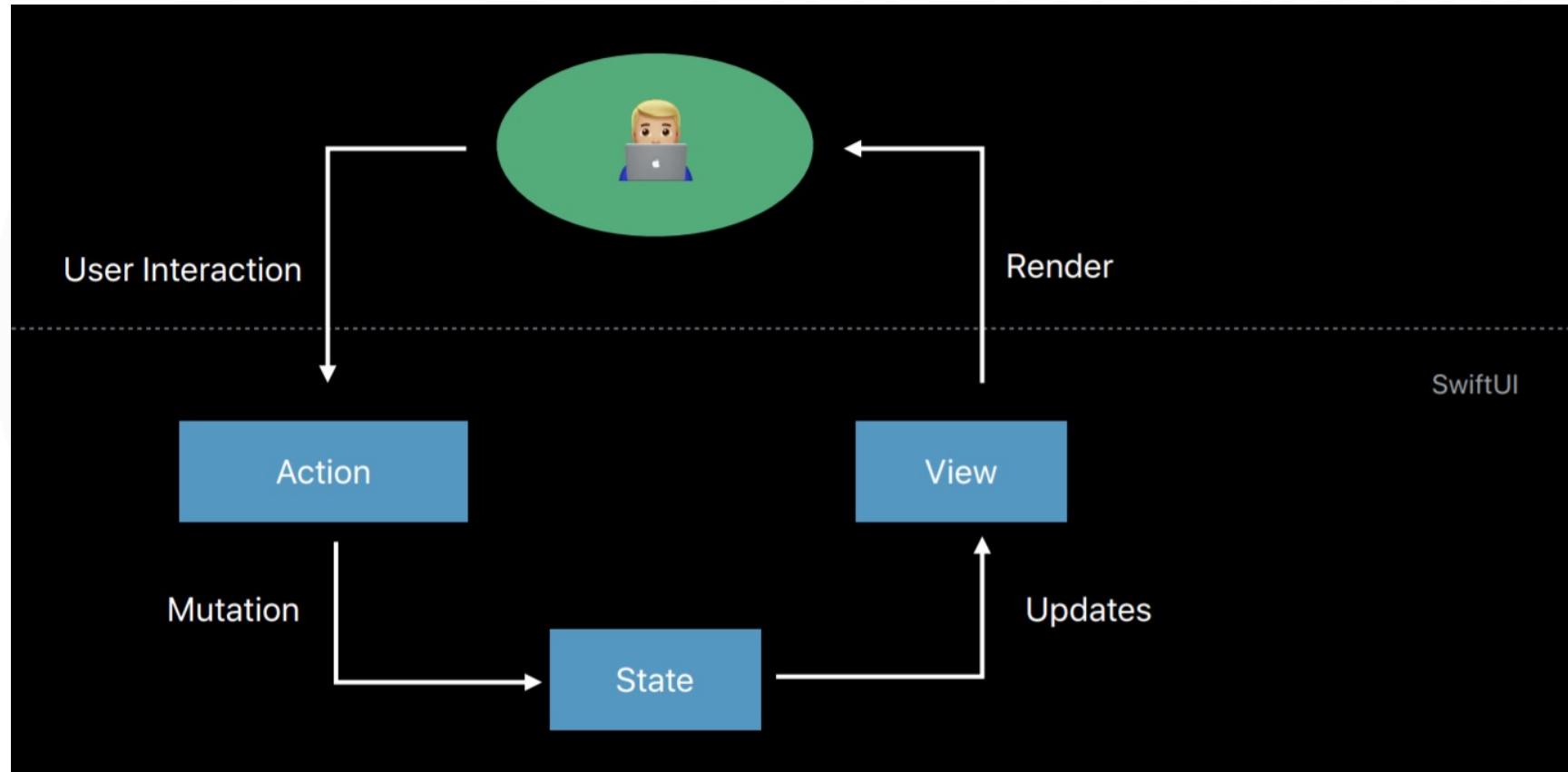
```
override func viewDidLoad() {  
    super.viewDidLoad()  
    page.currentPage = 0  
    page.numberOfPages =  
        animal.count  
    myLbl.text = animal[0]  
}
```

```
@IBAction func pageChanged(_  
    sender: Any) {  
    myLbl.text =  
        animal[page.currentPage]  
}
```

SwiftUI – cykl życia

- Deklaratywny szkielet programistyczny
- Widoki są deklarowane jako typy wartości, a nie jako konkretne odniesienia do tego, co jest wyświetlane na ekranie
- Właściwość *body* opisuje, w jaki sposób widok ma być renderowany
- Protokół *App* zastąpił *app delegate*
- Każdy *@State* jest źródłem prawdy

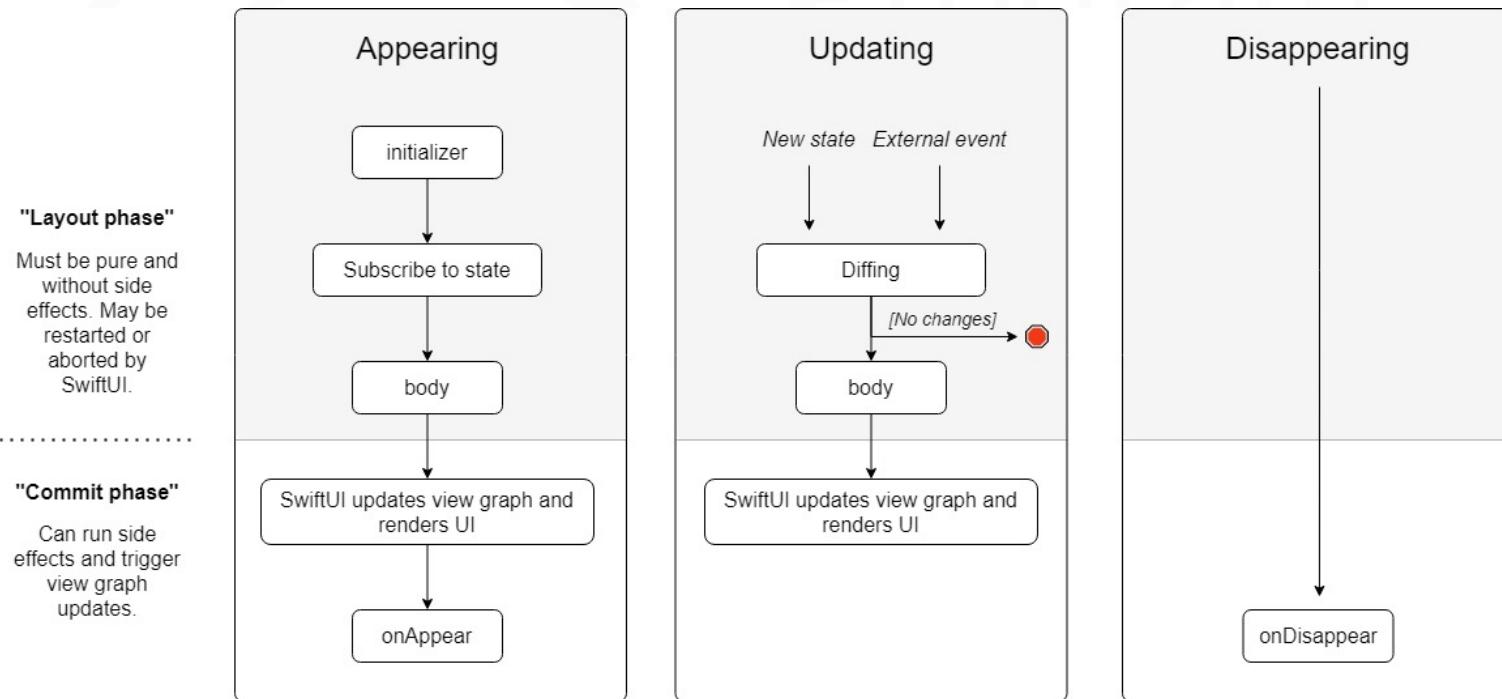
SwiftUI – cykl życia



źródło: <https://nalexn.github.io/clean-architecture-swiftui/>

SwiftUI – cykl życia

- Widok może być manipulowany w jego 3 fazach:



źródło: <https://www.vadimbulavin.com/swiftui-view-lifecycle/>

SwiftUI – cykl życia

- 2 fazy renderowania:
 - układ – layout
 - zatwierdzenie – commit
- Układ:
 - inicjuje hierarchię widoków bez renderowania
 - oblicza ramki
 - łączy widoki ze stanem
 - oblicza różnice do zatwierdzenia na ekranie
- Zatwierdzenie:
 - aktualizuje hierarchię widoków renderowania
 - zatwierdza wszystkie zmiany na ekranie
 - usuwa wszystkie widoki, które nie są już potrzebne

SwiftUI – cykl życia

- Pojawienie się – *Appearing*:
 - widok jest inicjowany i renderowany po raz pierwszy
- Etapy:
 - inicjalizacja widoku – widok nie jest połączony ze stanem, hierarchia widoków jest tworzona
 - połączenie widoku ze stanem
 - widok *body* jest wywoływany po raz pierwszy
 - graf widoku jest aktualizowany
 - wywoływana jest metoda *onAppear()* od widoku rodzica do widoku dziecka

SwiftUI – cykl życia

- Uaktualnienie – *Updating*:
 - jest wykonywana w odpowiedzi na zdarzenie zewnętrzne lub mutację stanu
- Etapy:
 - akcja użytkownika powoduje zmianę stanu lub SwiftUI wykrywa dane emitowane przez wydawcę obserwowane przez `View.onReceive()`
 - widok, który posiada stan zmutowany lub który odebrał zdarzenie zewnętrzne, jest porównywany z poprzednią migawką
 - SwiftUI unieważnia zmienione widoki
 - graf widoku jest aktualizowany, aktualizacje są wykonywane od góry do dołu hierarchii
- Przesuwanie stanu w dół hierarchii widoków zmniejsza liczbę widoków, które mają zostać unieważnione i ponownie renderowane, gdy stan się zmieni

SwiftUI – cykl życia

- Usuwanie – *Disappearing*:
 - usuwanie widoku z hierarchii
- Etapy:
 - metoda *onDisappear()* jest wywoływana po usunięciu widoku z hierarchii
 - wykonywane są metody *onAppear()*, *onDisappear()* od widoku rodzica do widoku dziecka

Widok i kontroler – SwiftUI

- Modyfikatory – stosowane do widoku lub innego modyfikatora widoku, tworząc inną wersję oryginalnej wartości
- Przykładowe modyfikatory:
 - *onAppear* pozwala uzyskać takie same zachowanie jak przy pomocy metody `viewWillAppear`
 - *font()*
 - *background()*
 - *clipShape()*
 - *padding()*
 - niestandardowe

Widok i kontroler – SwiftUI

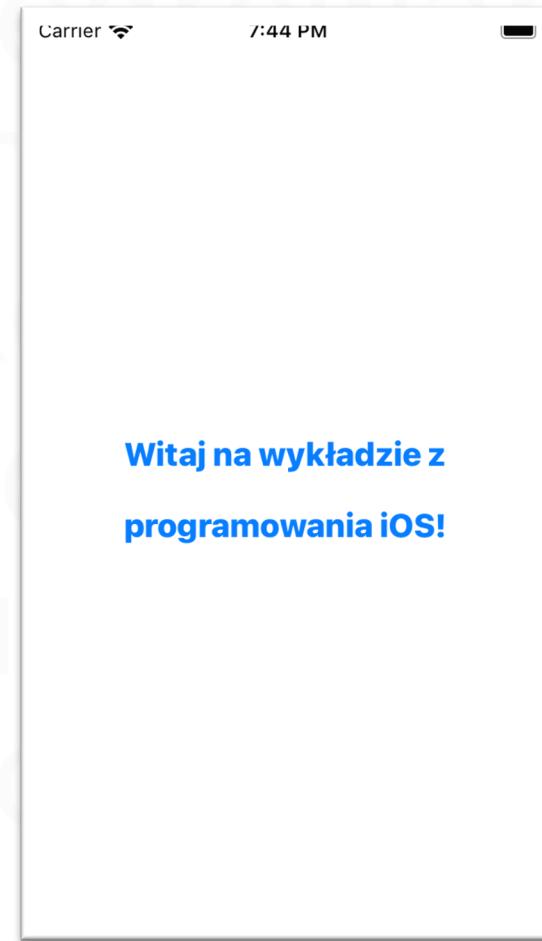
- Tekst (*UIText*) – wyświetla jedną lub wiele linii tekstu nieedytowanego
- Modyfikatory:
 - *.font(.title)*
 - *.font(.system(size: 20, weight:.heavy))*
 - *.foregroundColor()*
 - *.lineSpacing()*
 - *.frame(width: , alignment: .)*



Widok i kontroler – SwiftUI

- Tekst

```
struct ContentView: View {  
    var body: some View {  
        Text("Witaj na wykładzie  
            z programowania iOS!")  
            .padding()  
            .font(.system(size: 20,  
                weight:.heavy))  
            .foreground(Color.blue)  
            .lineSpacing(20.0)  
            .frame(width: 300,  
                alignment: .center)  
    }  
}
```



Widok i kontroler – SwiftUI

- Pole tekstowe (*UITextField*) – wyświetla jedną linię tekstu
 - znaki łamania linii jest traktowane jako spacje
 - *autolayout* – dostosowanie szerokości do tekstu
 - dla ustalonej szerokości:
 - *adjustsFontSizeToFitWidth*
 - *minimumFontSize*
 - *placeholder* – informacyjny tekst, gdy pole jest puste
 - *clearsOnBeingEditing* – automatycznie usuwa tekst w momencie rozpoczęcia edycji (jeśli właściwość jest true)
 - *clearsOnInsertion* – w momencie edycji istniejący tekst jest zaznaczany i nadpisywany nowym
 - *borderStyle: .none; .line; .bezel; .roundedRect*
 - obraz dla tła nie można ustawić dla *.roundedRect*

Widok i kontroler – SwiftUI

- Pole tekstowe (*UITextField*)
 - dostępne widoki:
 - *leftView*
 - *rightView*
 - możliwość wyświetlenia przycisku czyszczenia (*Clear*)
 - tryby widoków:
 - *.never* – widok nie jest wyświetlany
 - *.whileEditing* – pojawia się w momencie edytowania
 - *.unlessEditing* – pojawia się w momencie braku edytowania istniejącego tekstu lub braku tekstu
 - *.always* – przycisk *Clear* pojawia się, jeśli tekst istnieje, widoki (*leftView* lub *rightView*) są zawsze widoczne

Widok i kontroler – SwiftUI

- Pole tekstowe (*UITextField*)
 - jest *first responder*, kiedy jest edytowane (pojawia się klawiatura)
 - pojawienie się klawiatury – ustawienie statusu *becomeFirstResponder*
 - ukrycie klawiatury – utworzenie delegata i wywołanie jego metody *textFieldShouldReturn(_ :)* – po kliknięciu przycisku *Return* pole nie ma statusu *first responder*

```
func textFieldResponder(_ textf: UITextField) -> Bool {  
    textf.resignFirstResponder()  
    return false  
}
```

Widok i kontroler – SwiftUI

- Pole tekstowe (*UITextField*)
 - odczytanie tekstu

```
 VStack{  
     TextField("Podaj imie:", text: $name)  
         .border(Color.green, width: 1)  
     Text(name)  
 }  
 .padding()  
 .font(.title)  
 }
```

Podaj imie:

Ania

Ania

Widok i kontroler – SwiftUI

- Pole tekstowe (*UITextField*)
 - sprawdzenie danych

```
 VStack{  
     TextField("Podaj swój email:",  
               text: $email)  
     .onSubmit{  
         correctEmail = checkEmail(email:  
                                     email)  
     }  
     ....  
     if (correctEmail) {  
         Text("Poprawny email")  
     } else {  
         Text("Błędny email")  
     }  
 }  
 ....
```

jk@gmail.com

Błędny email

```
func checkEmail(email : String) ->  
Bool{  
    let emailFormat = "[A-Z0-9a-  
z. %+-]+@[A-Za-z0-9.-]+\\".[A-Za-  
z]{2,64}"  
    let predicte =  
        NSPredicate(format: "SELF  
MATCHES %@", emailFormat)  
    return predicte.evaluate(with:  
                           email)  
}
```

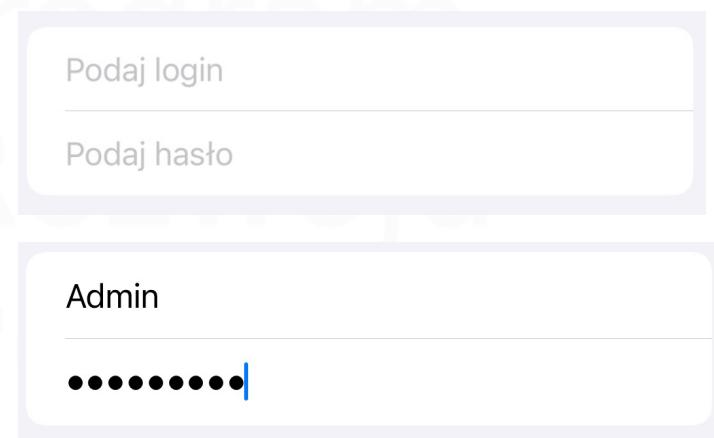
jk@gmail.com

Poprawny email

Widok i kontroler – SwiftUI

- Formularz
 - połączenie kilku pól tekstowych

```
struct ContentView: View {  
    @State var name : String = ""  
    @State var passwd : String = ""  
  
var body: some View {  
    Form {  
        TextField(text: $name, prompt: Text("Podaj login")) {  
            Text("Imie")  
        }  
        SecureField(text: $passwd, prompt: Text("Podaj hasło")){  
            Text("Hasło")  
        }  
    }  
}
```



Widok i kontroler – SwiftUI

- Przycisk (*UIButton*) – zawiera:
 - *tytuł* – razem z nim można zdefiniować kolor tekstu, kolor cienia tekstu
 - *obrazek* – wyświetlany razem z tytułem, jeśli obrazek mieści się w jego granicach
 - *obraz tła* – zawsze jest dopasowywany do rozmiarów przycisku
 - stany przycisku:
 - *.normal*
 - *.highlited*
 - *.selected*
 - *.disabled*
 - metody dla stanów:
 - *setTitle(_ : for :)*
 - *setTitleColor(_ : for :)*
 - ...

Widok i kontroler – SwiftUI

- Przycisk (*UIButton*)

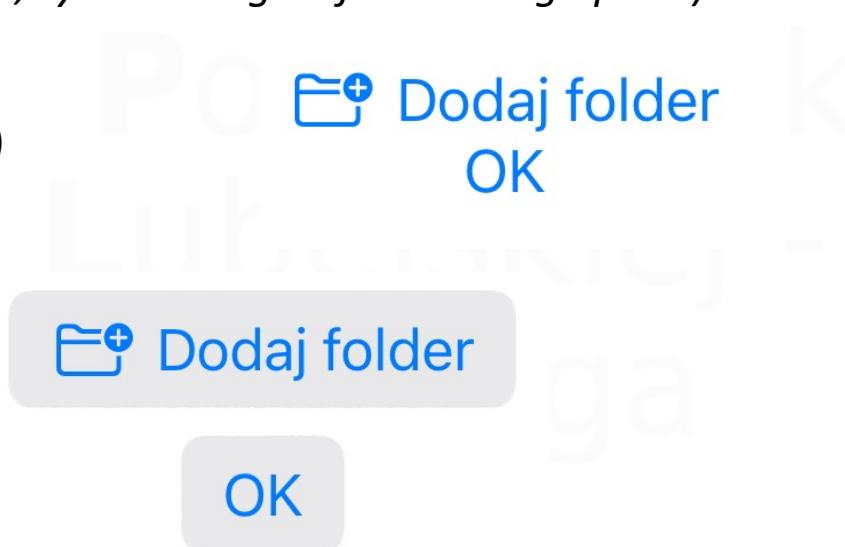
- akcja

```
 VStack{
```

```
     Button(action: add) {
         Label("Dodaj folder", systemImage: "folder.badge.plus")
     }
```

```
     Button("OK", action: {
         print("Wciśnięto")
     })
 }
```

- *.buttonStyle(.bordered)*



Dodaj folder
OK

Dodaj folder

OK

Widok i kontroler – SwiftUI

- Obraz (*UIImage*) – wyświetla obraz zapisany w katalogu *Asset* albo w *app bundle*

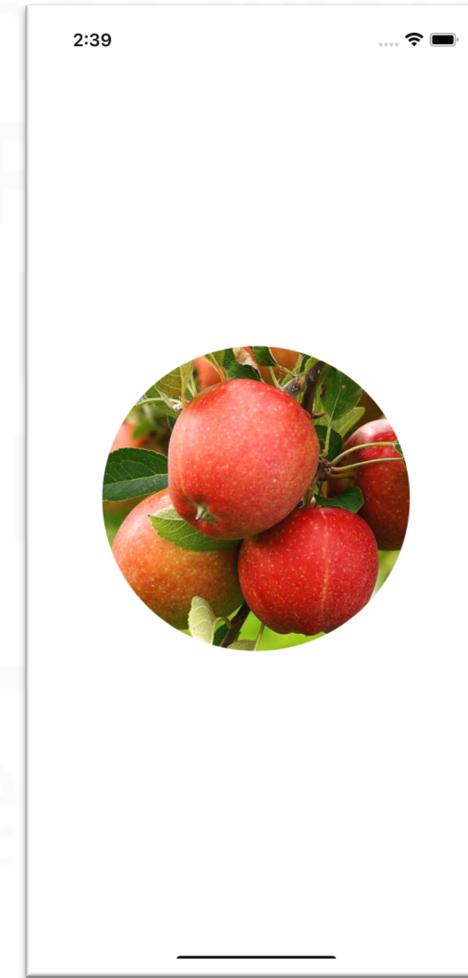
```
Image("apple-tree")
    .resizable()
    .scaledToFit()
    .frame(width: 220, height: 220, alignment: .center)
}
```



Widok i kontroler – SwiftUI

- Obraz (*UIImage*)

```
Image("apple-tree")
    .resizable()
    .scaledToFit()
    .clipShape(Circle())
}
```



Widok i kontroler – SwiftUI

- Wybór daty (*UIDatePicker*) – wyświetla datę oraz czas korzystając z widoku kalendarza lub liczbowo
 - po zmianie ustawień, zdarzenie *Value Changed* jest wywoływanie
 - style (*preferredDatePickerStyle*):
 - *.inline* – pełny interfejs jest pokazywany przez cały czas
 - *.compact* – interfejs jest wyświetlany w postaci linii tekstu, dopiero po kliknięciu pokazuje się cały interfejs
 - *.time* – wyświetlenie czasu w postaci etykiet czasowych z obsługą klawiatury oraz segmentem AM/PM
 - *.date* – wyświetlenie daty w postaci kalendarza
 - *.dateAndTime* – wyświetlenie daty i czasu w postaci kalendarza
 - *.countDownTimer* – wyświetla godziny i minuty do odliczania

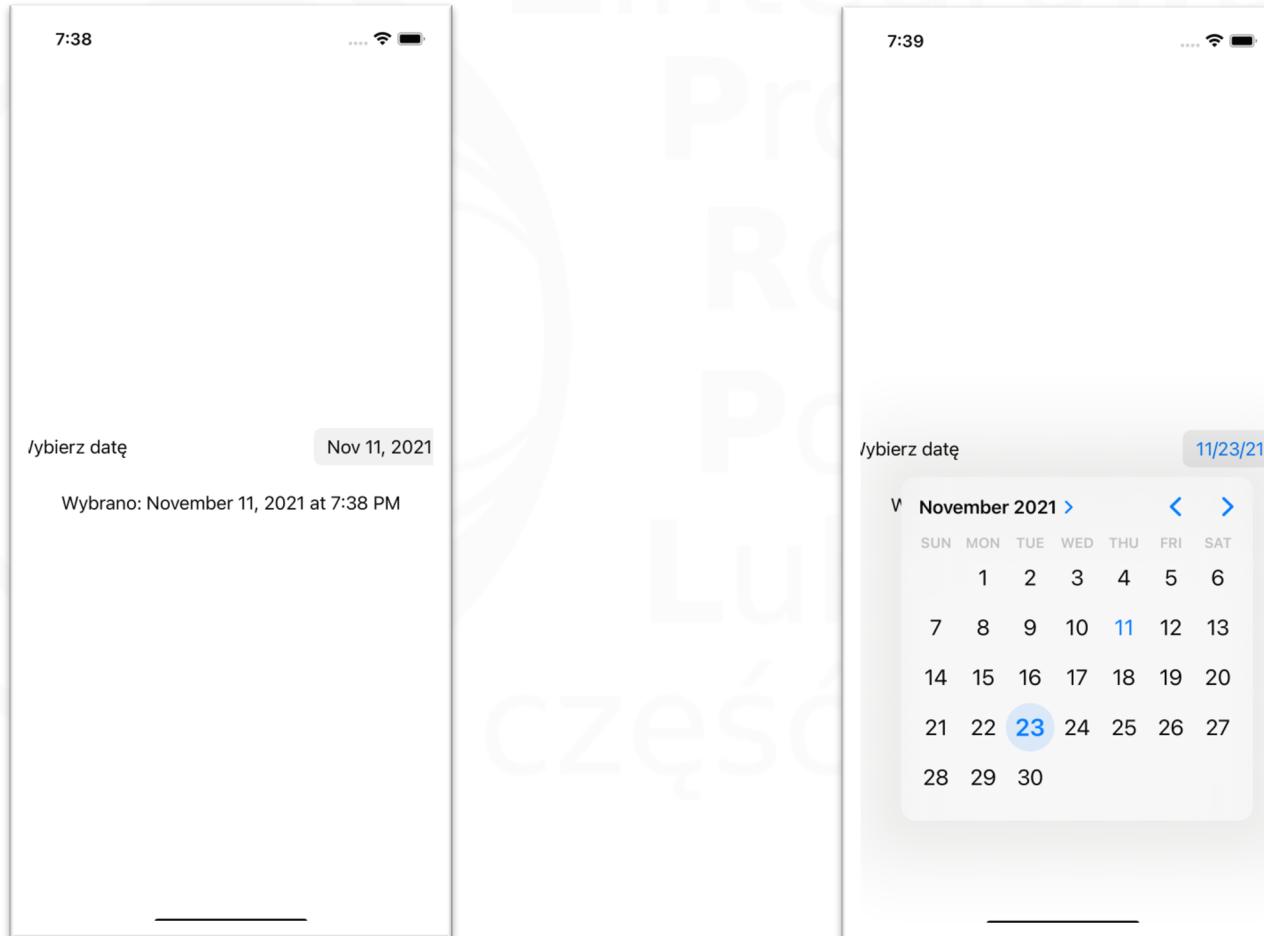
Widok i kontroler – SwiftUI

- Wybór daty
(UIDatePicker)

```
struct ContentView: View {  
    @State var myDate : Date =  
        Date()  
    let dateFormatter :  
        DateFormatter = {  
            let format =  
                DateFormatter()  
            format.dateStyle = .long  
            format.timeStyle = .short  
            return format  
        }()  
}
```

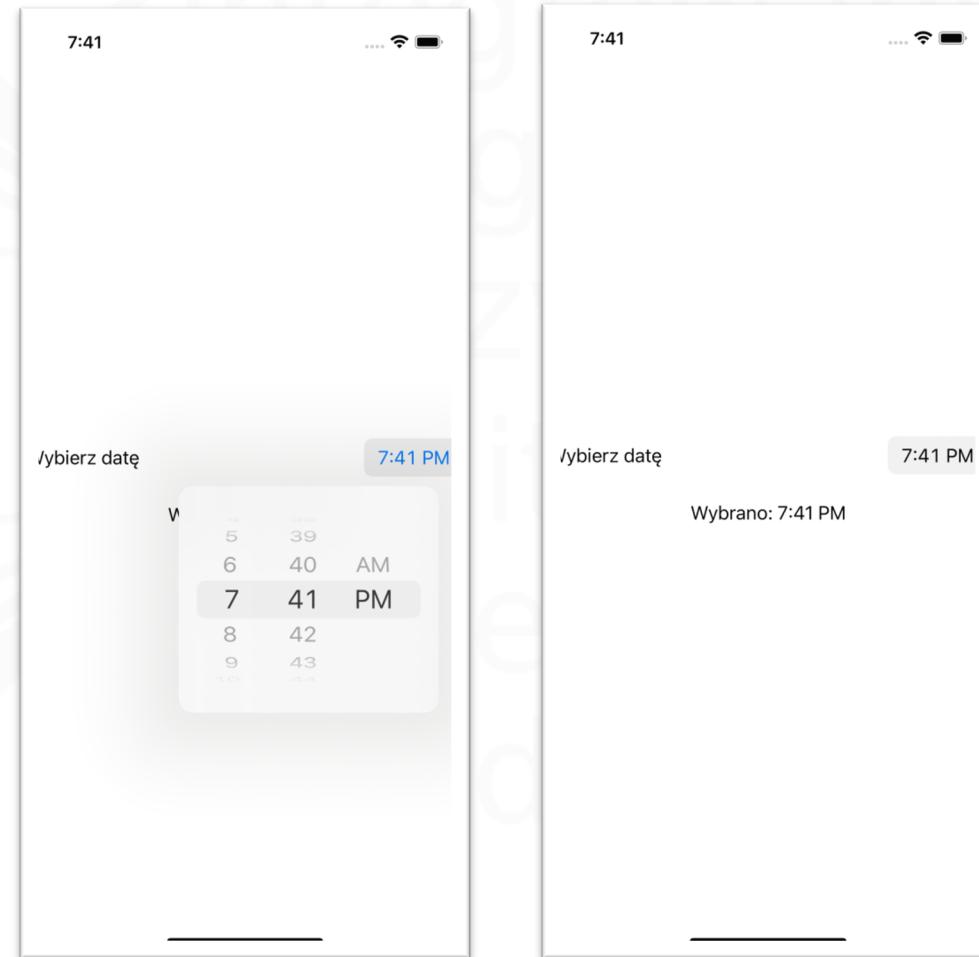
```
var body: some View {  
    VStack{  
        DatePicker("Wybierz datę",  
            selection: $myDate,  
            displayedComponents:  
                .date)  
        Text("Wybrano: \(myDate,  
            formatter: dateFormatter)")  
        .padding()  
    }  
}
```

Widok i kontroler – SwiftUI



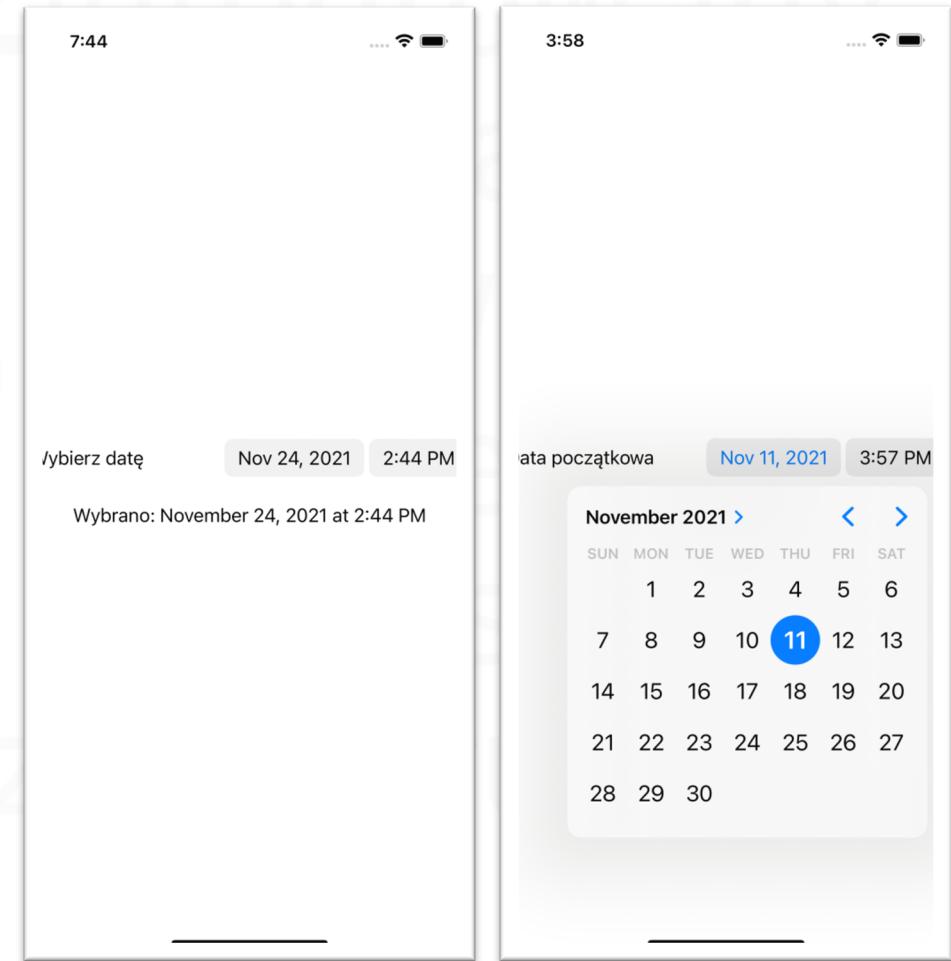
Widok i kontroler – SwiftUI

*displayedComponents:
.hourAndMinute)*



Widok i kontroler – SwiftUI

*displayedComponents:
[.date, .hourAndMinute)]*



Widok i kontroler – SwiftUI

- Lista (*UIPickerView*) – wyświetlenie listy elementów
 - domyślna wysokość 162 dla szerokości *.compact* oraz 261 dla pozostałych ustawieniach
 - *UIPickerViewDataSource*
 - *numberOfComponents(in:)*
 - *pickerView(_ : numberOfRowsInComponent:)*
 - *pickerView(_ : titleForRow: forComponent:)*
 - *pickerView(_ : attributedTitleForRow: forComponent:)*
 - *pickerView_ : viewForRow: forComponent: reusing:)*
 - *UIPickerViewDelegate*
 - *pickerView(_ : rowHeightForComponent:)*
 - *pickerView(_ : widthForComponent:)*

Widok i kontroler – SwiftUI

- Lista (*UIPickerView*)
 - *UIPickerViewDelegate*
 - *pickerView(_: didSelectRow: inComponent:)*
 - odświeżenie listy elementów
 - *reloadComponent(_:)*
 - *reloadAllComponents(_:)*
 - *numberOfComponents*
 - *numberOfRows(inComponent:)*
 - *View(forRow: forComponent:)*
 - numeracja wierszy od 0

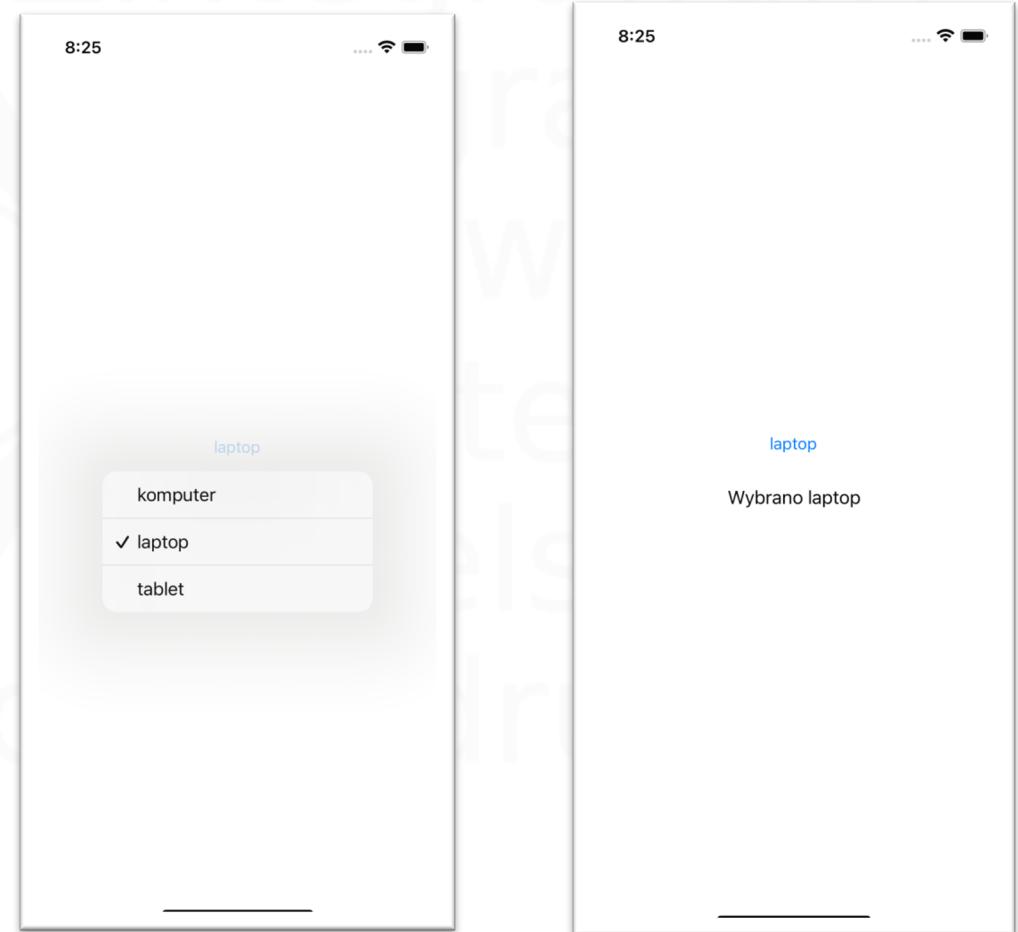
Widok i kontroler – SwiftUI

- Lista (*UIPickerView*) – wyświetlenie i obsługa wyboru

```
let list = ["komputer", "laptop", "tablet"]
@State var selectedItem : String = "komputer"
var body: some View {
    VStack{
        Picker("Wybierz element", selection: $selectedItem){
            ForEach(list, id: \.self){
                Text($0)
            }
        }
        Text("Wybrano \(selectedItem)")
        .padding()
    }
}
```

Widok i kontroler – SwiftUI

- Lista (*UIPickerView*)



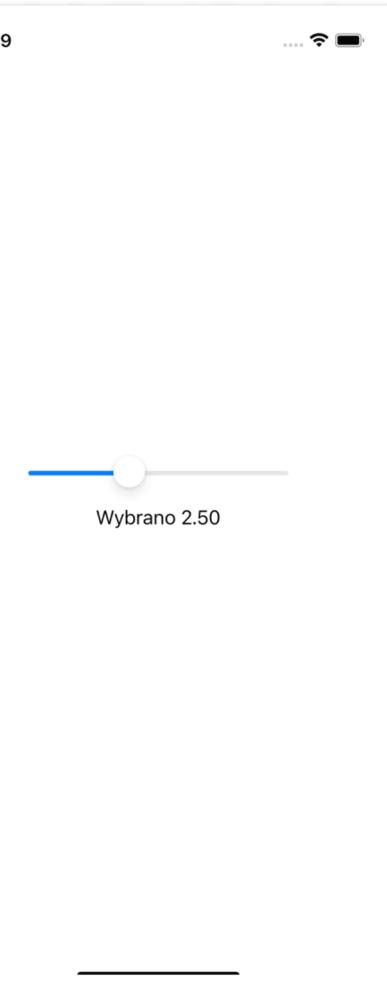
Widok i kontroler – SwiftUI

- Suwak (*UISlider*) – zmiana wartości od minimalnej do maksymalnej w sposób ciągły
 - *minimumValue*
 - *maximumValue*
 - domyślne ustawienia: 0-1
 - podczas zmiany wywoływana jest metoda *Value Changed*
 - nie ma wbudowanej metody reakcji na tapnięcie
 - należy zastosować metody rozpoznawania gestów *UITapGestureRecognizer*

Widok i kontroler – SwiftUI

- Suwak (*UISlider*)

```
Struct ContentView: View {  
    @State var chosenVal : Double = 1.5  
    var body: some View {  
        VStack{  
            Slider(value: $chosenVal, in: 1...5, step: 0.5)  
                .frame(width: 225, height: 10, alignment:  
.center)  
            Text(String(format: "Wybrano %.2f", chosenVal))  
                .padding()  
        }  
    }  
}
```



Widok i kontroler – SwiftUI

- Segment (*UISegmentedControl*) – wiersz elementów (segmentów) do wyboru
 - od iOS 13 zaznaczony element jest oznaczony zaokrąglonym prostokątem
 - właściwość zaznaczenia:
 - *isMomentary (false)* – zaznaczenie w momencie wyboru, które utrzymuje się
 - *isMomentary (true)* – zaznaczenie w momencie wyboru, które znika, co powoduje, że zaznaczenie nie jest widoczne
 - zaznaczony element można zlokalizować właściwością *SelectedSegmentIndex*
 - zaznaczony element wywołuje zdarzenie *Value Changed*
 - brak zaznaczonego elementu: *UISegmentControl.noSegment*
 - włączenie / wyłączenie segmentu: *setEnabled(_ : forSegmentAt:)*
 - posiada albo tytuł, albo obraz
 - *setTitle(_ : forSegmentAt:)*, *titleForSegment(at:)*
 - *setImage(_ : forSegmentAt:)*, *imageForSegment(at:)*

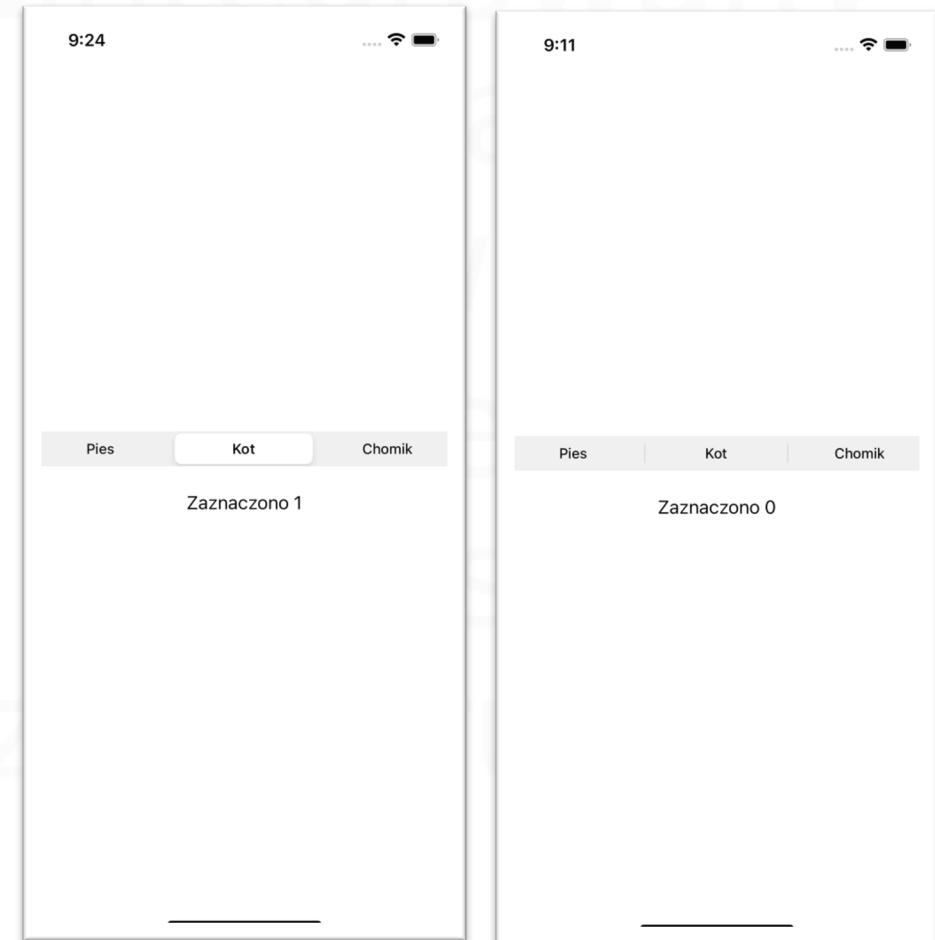
Widok i kontroler – SwiftUI

- Segment (*UISegmentedControl*)

```
Struct ContentView: View {  
    @State var selectedItem : Int = 0  
    let list = ["Pies", "Kot", "Chomik"]  
    var body: some View {  
        VStack{  
            Picker("wybierz zwierzę", selection: $selectedItem) {  
                ForEach(0 ..< list.count) { item in  
                    Text(list[item])  
                }  
            }.pickerStyle(SegmentedPickerStyle())  
            Text("Zaznaczono \(selectedItem)")  
            .padding()  
        }  
    }  
}
```

Widok i kontroler – SwiftUI

- Segment
(*UISegmentedControl*)



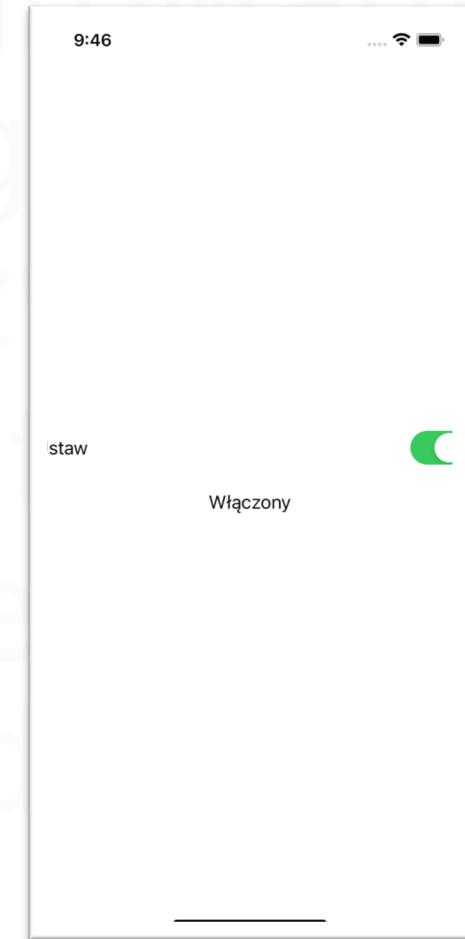
Widok i kontroler – SwiftUI

- Przełącznik(*UISwitch*) – przełącznik o dwóch stanach
 - posiada właściwość `isOn` (true lub false)
 - zmiana wywołuje metodę *Value Changed*
 - zmiana zaznaczenia: `setOn(_ : animated:)`
 - nie można zmienić jego rozmiaru
 - nie można zmienić koloru przejścia w stan *Off*
 - ustawienia:
 - `onTintColor` – kolor śledzenia zmiany ustawienia na *On*
 - `thumbTintColor` – kolor suwanego przycisku

Widok i kontroler – SwiftUI

- Przełącznik(*UISwitch*)

```
struct ContentView: View {  
    @State var isSelected :Bool = true  
    var body: some View {  
        VStack{  
            Toggle("Ustaw", isOn: $isSelected)  
            if(isSelected){  
                Text("Włączony")  
                    .padding()  
            }  
            else {  
                Text("Wyłączony")  
                    .padding()  
            }  
        }  
    }  
}
```



Widok i kontroler – SwiftUI

- Kontrola stron (*UIPageControl*) – wiersz kropek, reprezentujących kolejne strony
 - często stosowany z innymi kontrolkami (np. *UIScrollView*)
 - liczba stron (kropek): *numberOfPages*
 - bieżąca strona: *currentPage*
 - od iOS 14 nie musi być wystarczająco szeroki do wyświetlenia wszystkich kropek, wystąpi automatyczne przesunięcie
 - dla właściwości *hidesForSinglePage* ustawionej na *true*, w przypadku jednej kropki, będzie ona niewidoczna
 - właściwości:
 - *pageIndicatorTintColor* – kolor kropek
 - *currentPageIndicatorTintColor* – kolor bieżącej kropki
 - *setIndicatorImage(_ :forPage:)* – ustawienie obrazu dla konkretnej kropki

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
INFORMATYKA



Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego



**Fundusze
Europejskie**
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



POLITECHNIKA LUBELSKA

WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI

INFORMATYKA



Zintegrowany
Program
Rozwoju
Politechniki
Lubelskiej -
część druga

Programowanie aplikacji mobilnych na platformę iOS

W5
Środowisko Xcode, Storyboard

Maria Skublewska-Paszkowska



**Fundusze
Europejskie**
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



Agenda

- Xcode
- Soryboard

Zintegrowany
Program
Rozwoju
Politechniki
Lubelskiej -
część druga

Xcode

- Zintegrowane środowisko programistyczne (IDE) firmy Apple dla systemu macOS, używane do tworzenia oprogramowania dla systemów macOS, iOS, iPadOS, watchOS i tvOS.
- Dostępny w Mac App Store bezpłatnie dla zarejestrowanych użytkowników
- Wspiera wytwarzanie oprogramowania dla języków:
 - Objective-C
 - Swift
 - C
 - C++
 - Java,
 - Python
 - inne



Xcode

- Wytwarzanie oprogramowania i jego testowanie
- Tworzenie graficznego interfejsu użytkownika
- Debuggowanie oprogramowania
- Testowanie oprogramowania – platforma testowa do testowania funkcjonalnego, wydajnościowego i interfejsu użytkownika
- Zintegrowana dokumentacja
- Udostępnianie oprogramowania za pomocą narzędzi opartych na chmurze, a wyniki są wyświetlane w nawigatorze raportów
- Wyświetlanie i edycja pracy oraz wyświetlanie wyników komplikacji w App Store Connect
- Dystrybuowanie komplikacji automatycznie do testerów za pomocą TestFlight

Xcode

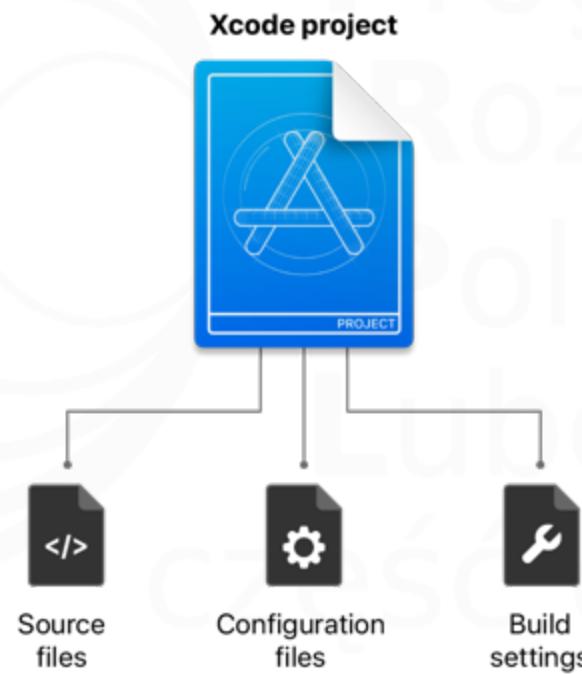
- Minimalne wymagania:

Xcode Version	Minimum OS Required	SDK	Architecture	Deployment Targets	Simulator	Swift
Xcode 13.1	macOS Big Sur 11.3	iOS 15 macOS 12 tvOS 15 watchOS 8 DriverKit 21.0.1	x86_64 armv7 armv7s armv7k arm64 arm64e arm64_32	iOS 9-15 iPadOS 13-15 macOS 10.9-12 tvOS 9-15 watchOS 2-8 DriverKit 19-21.0.1	iOS 10.3.1-15 tvOS 10.2-15 watchOS 3.2-8	Swift 4 Swift 4.2 Swift 5.5
Xcode 13	macOS Big Sur 11.3	iOS 15 macOS 11.3 tvOS 15 watchOS 8 DriverKit 20.4	x86_64 armv7 armv7s armv7k arm64 arm64e arm64_32	iOS 9-15 iPadOS 13-15 macOS 10.9-11.3 tvOS 9-15 watchOS 2-8 DriverKit 19-20.4	iOS 10.3.1-15 tvOS 10.2-15 watchOS 3.2-8	Swift 4 Swift 4.2 Swift 5.5

źródło: <https://developer.apple.com/support/xcode/>

Xcode

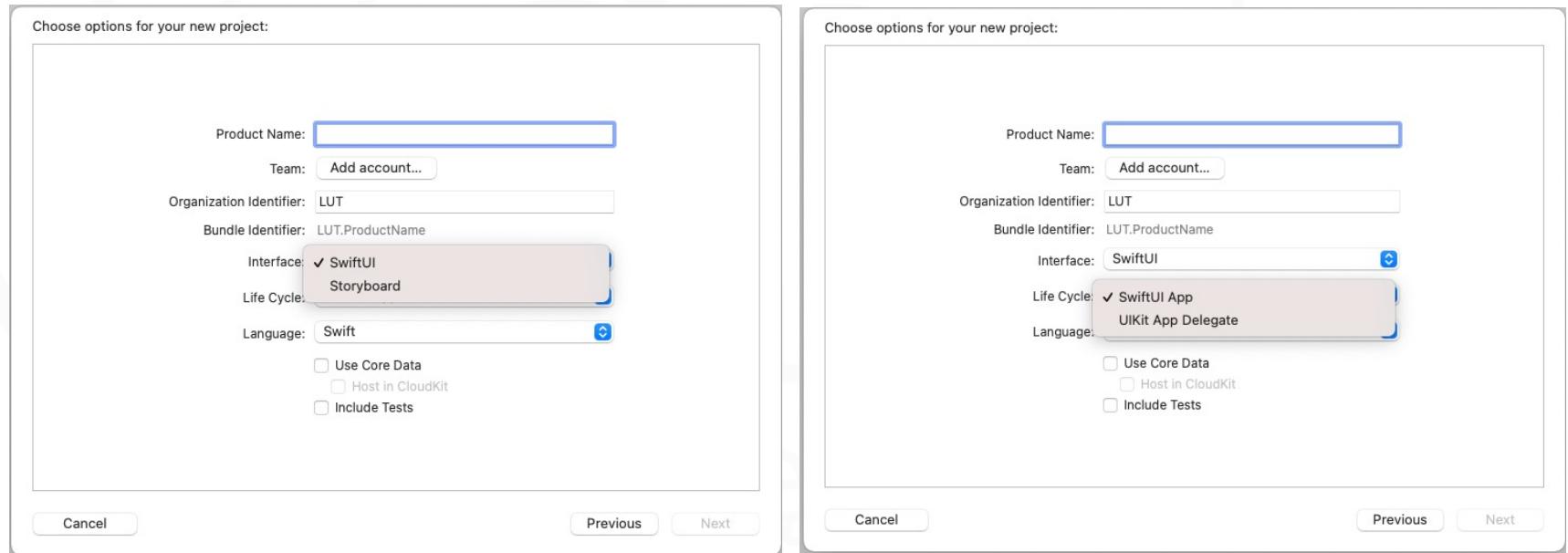
- Zapewnia szablony dla różnych typów aplikacji dla platformy Apple



źródło:<https://developer.apple.com/tutorials/app-dev-training/creating-a-storyboard-app>

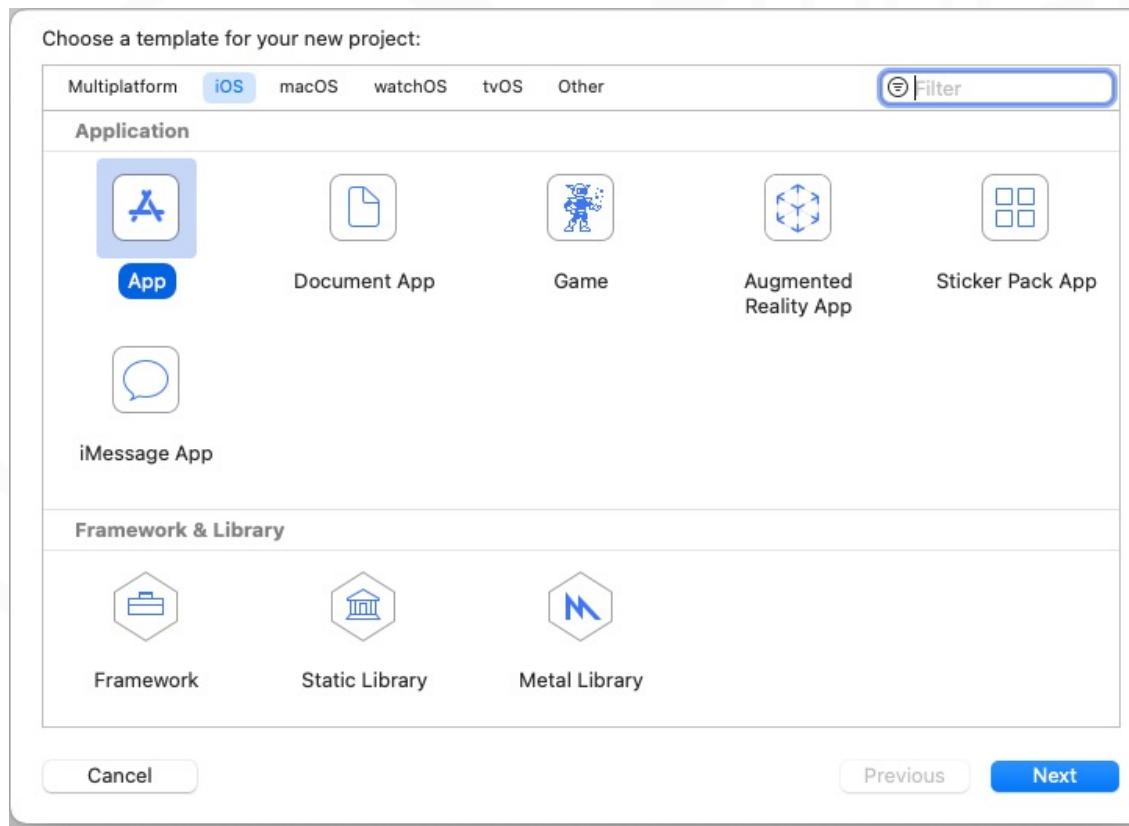
Xcode

- Tworzenie nowego projektu



Xcode

- Szablony



Xcode

- Nowy projekt oparty jest na pakiecie, dostarczany z plikiem o nazwie *<projekt>-Info.plist*
- W czasie komplikacji plik jest używany do generowania pliku *Info.plist*, który jest następniełączany do wynikowego pakietu

Key	Type	Value
Localization native development region	String	\$(DEVELOPMENT_LANGUAGE)
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	\$(PRODUCT_BUNDLE_PACKAGE_TYPE)
Bundle version string (short)	String	1.0
Bundle version	String	1
Application requires iPhone environment	Boolean	YES
Application Scene Manifest	Dictionary	(1 item)
Application supports indirect input events	Boolean	YES
Launch Screen	Dictionary	(0 items)
Required device capabilities	Array	(1 item)
Supported interface orientations	Array	(3 items)
Supported interface orientations (iPad)	Array	(4 items)

Xcode



źródło: <https://developer.apple.com/documentation/xcode/creating-an-xcode-project-for-an-app>

Xcode

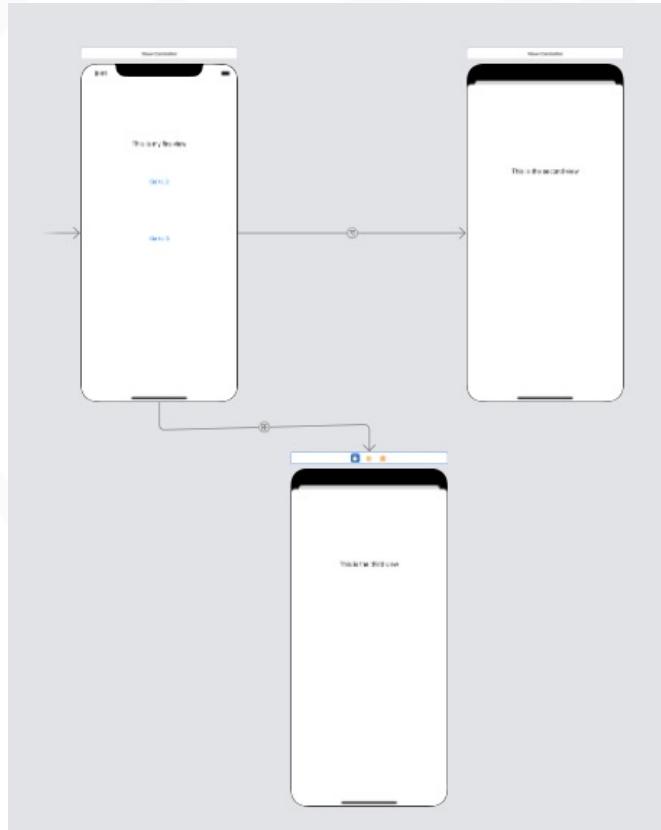
- Swift playground („a place where people can play”) – projekt służący do nauki języka
 - tworzenie małych programów zwane „playgrounds”, które natychmiast pokażą wyniki napisanego kodu
 - prototypowanie algorytmów, czy interfejsu użytkownika
 - tworzenie komponentów i ich dokładnego przetestowania przed umieszczeniem ich w aplikacji mobilnej
 - wartości różnego typu są wyświetlane w odmienny sposób (np. wykres, lista wartości, zmienna)
 - interaktywne lekcje uczą kluczowych koncepcji kodowania
 - firmy publikują grounds, które można używać do sterowania robotami, dronami i innymi akcesoriami sprzętowymi za pomocą Bluetooth
 - możliwość udostępnienia kodu

Storyboard

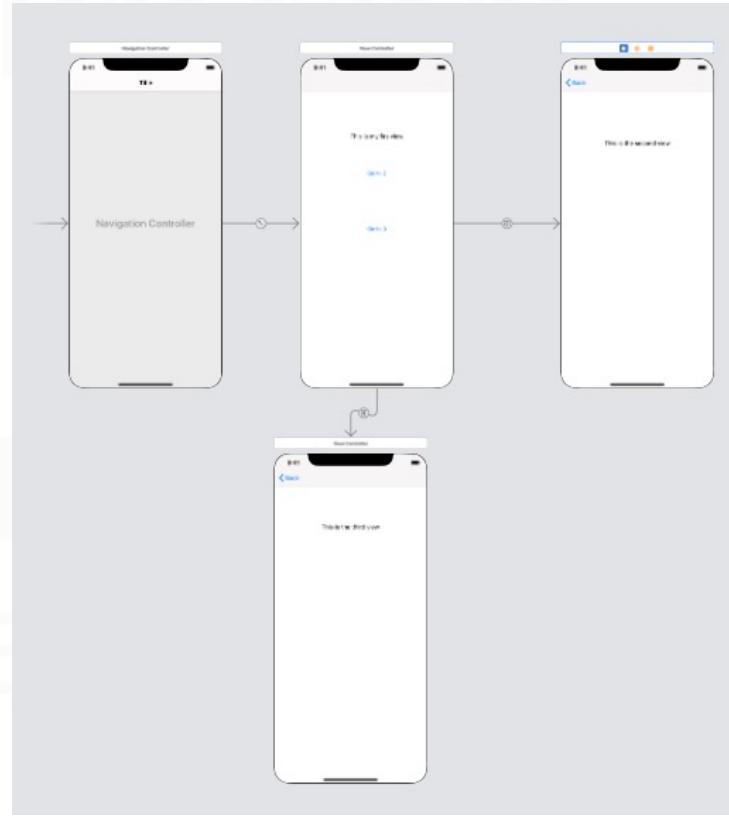
- Tworzenie graficznego interfejsu użytkownika dla UIKit
- Narzędzie Interface Builder:
 - sceny - obszar zawartości ekranu (pojedyncze na iPhone i iPodzie touch lub wiele na ekranie iPada i Macu)
 - przejście między scenami (ang. segues) – przejście od jednej sceny do kolejnej
 - kontrolki – elementy interfejsu graficznego użytkownika
- Plik interfejsu użytkownika ma rozszerzenie .storyboard lub .xib
- Storyboard to zestaw kontrolerów widoku i przejść między tymi kontrolerami
- Domyślne pliki interfejsu użytkownika są dostarczane przez Xcode podczas tworzenia nowych projektów z jego wbudowanych szablonów

Storyboard

UIKit

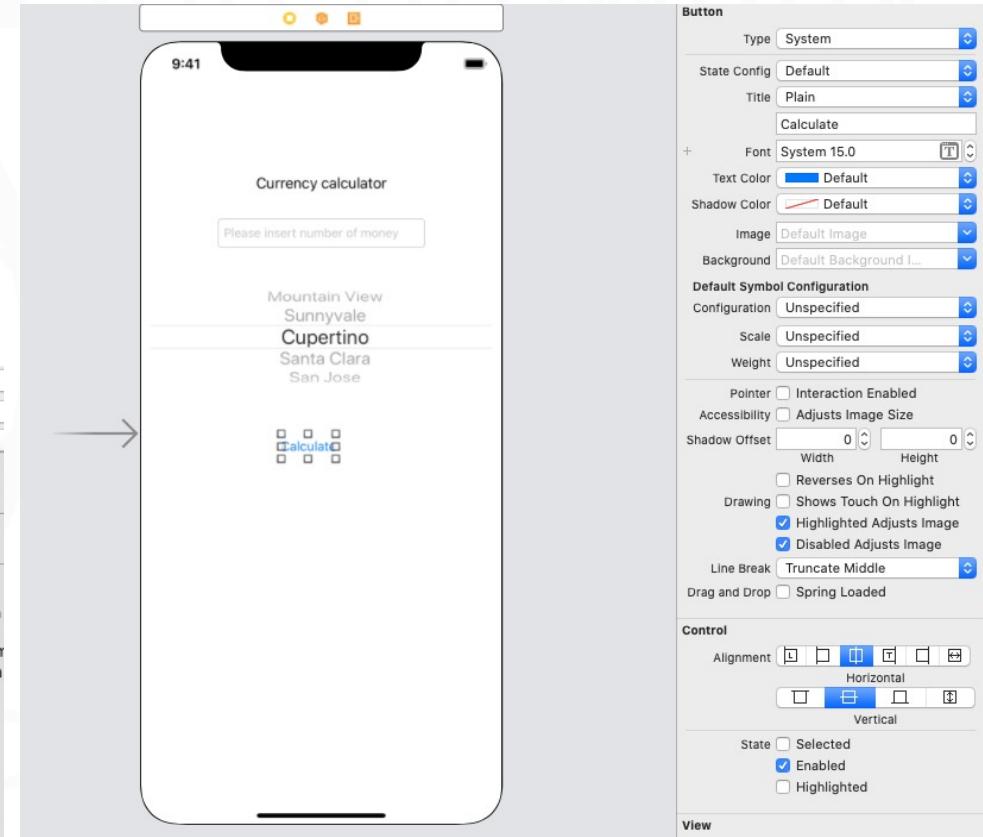
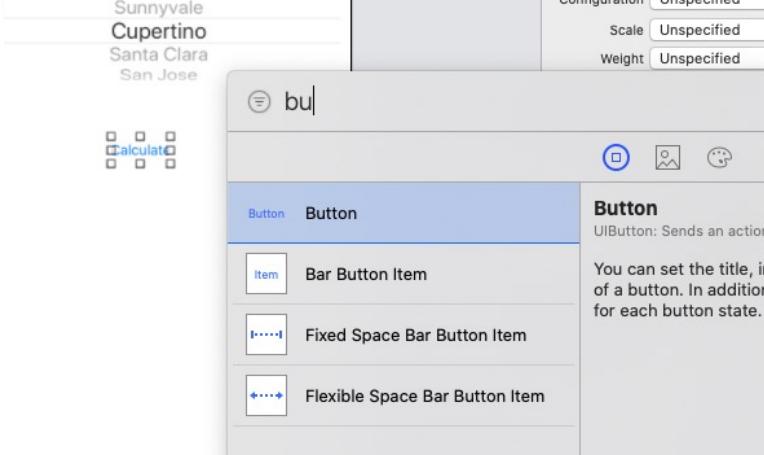


Nawigacja



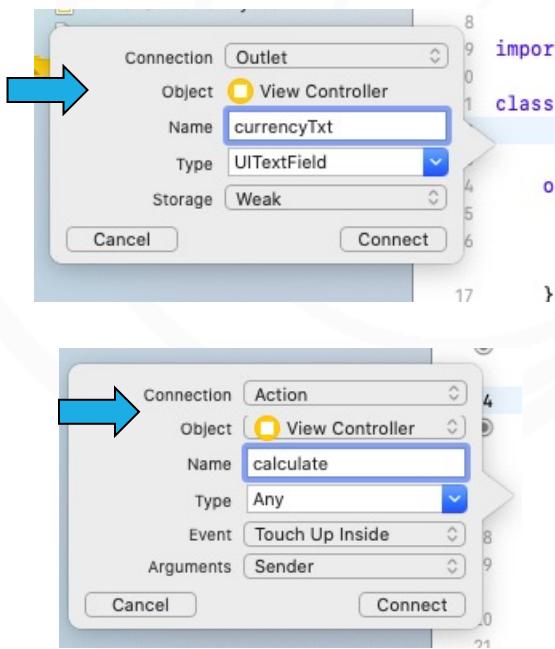
Storyboard

- UIKit – tworzenie widoku



Storyboard

- UIKit – tworzenie powiązań
ViewController.swift



```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var currencyTxt: UITextField!
    @IBOutlet weak var currencyPicker: UIPickerView!
    @IBOutlet weak var resultLabel: UILabel!

    @IBAction func calculate(_ sender: Any) {
    }
}
```

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
INFORMATYKA



Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego



**Fundusze
Europejskie**
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny





Programowanie aplikacji mobilnych na platformę iOS

W4

**Wytyczne dotyczące interfejsu użytkownika Apple
iOS**

Maria Skublewska-Paszkowska



**Fundusze
Europejskie**
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



Agenda

- Motywy projektowania interfejsu użytkownika
- Zasady projektowania interfejsu użytkownika
- Interfejs użytkownika – UIKit
- Działanie interfejsu użytkownika
- Rozmieszczenie elementów
- UIKit vs SwiftUI

Motywy projektowania interfejsu użytkownika

- Aby aplikacja odniosła sukces – spełnienie wysokich wydajności jakości i funkcjonalności
- Trzy motywy wyróżniające aplikacje iOS:
 - przejrzystość (ang. clarity):
 - ✓ tekst jest czytelny w każdym rozmiarze
 - ✓ ikony są precyzyjne i czytelne
 - ✓ ozdoby są subtelne i odpowiednie
 - ✓ położony nacisk na funkcjonalność
 - ✓ odstępy, kolor, czcionki, grafika i elementy interfejsu subtelnie podkreślają ważne treści i zapewniają interaktywność

Motywy projektowania interfejsu użytkownika

- Trzy motywy wyróżniające aplikacje iOS:
 - atencja (ang. deference)
 - ✓ płynny ruch i wyraźny interfejs pomagają zrozumieć treści oraz zapewnić poprawną interakcję z użytkownikiem
 - ✓ treść zazwyczaj wypełnia całą przestrzeń ekranu
 - ✓ minimalne użycie ramek, gradientów i cieni powoduje, że interfejs jest jasny i klarowny
 - ✓ prezentowana treść jest najważniejsza
 - głębokość (ang. depth)
 - ✓ wyraźne warstwy wizualne i realistyczny ruch przekazują hierarchię i ułatwiają zrozumienie
 - ✓ dotyk i łatwość odnajdywania umożliwiają dostęp do funkcjonalności i dodatkowej zawartości bez utraty kontekstu
 - ✓ przejścia między widokami zapewniają poczucie głębi podczas przeglądania treści

Zasady projektowania interfejsu użytkownika

- Projektując interfejs należy kierować się zasadami:
 - estetyczna integralność (ang. aesthetic integrity)
 - spójność (ang. consistency)
 - bezpośrednia manipulacja (ang. direct manipulation)
 - wsparcie (ang. feedback)
 - metafory (ang. metaphors)
 - kontrola użytkownika (ang. user control)
- Estetyczna integralność:
 - wygląd i zachowanie aplikacji integrują się z funkcjonalnością
 - utrzymanie koncentracji użytkownika dzięki subtelnej, dyskretnej grafice, standardowym kontrolkom i przewidywalnym zachowaniom
 - „wciągająca” aplikacja (np. gra) może zapewnić zabawę i emocje, jednocześnie zachęcając do odkrywania

Zasady projektowania interfejsu użytkownika

- Spójność:
 - implementowane są znane standardy i paradigmaty, korzystając z dostarczonych przez system elementów interfejsu, dobrze znanych ikon, standardowych stylów tekstu i jednolitej terminologii
 - aplikacja zawiera funkcje i zachowania w sposób, jakiego oczekują użytkownicy
- Bezpośrednia manipulacja:
 - manipulacja treścią angażyuje użytkowników i ułatwia jej zrozumienie
 - manipulacja podczas np. obracania urządzeniem, stosowania gestów
 - natychmiastowa obserwacja rezultatów działania użytkownika

Zasady projektowania interfejsu użytkownika

- Wsparcie:
 - informacja zwrotna potwierdza działania i pokazuje rezultaty
 - wbudowane aplikacje iOS zapewniają wyczuwalną informację zwrotną w odpowiedzi na każde działanie użytkownika
 - elementy interaktywne są na krótko podświetlane po dotknięciu, wskaźniki postępu informują o stanie długotrwałych operacji, a animacja i dźwięk pomagają wyjaśnić działania
- Metafory:
 - szybsza nauka, gdy wirtualne obiekty i działania aplikacji są metaforami znajomych doświadczeń, ze świata rzeczywistego lub cyfrowego
 - użytkownicy fizycznie wchodzą w interakcję z ekranem: przesuwają widoki, aby odsłonić zawartość poniżej ich, przeciągają i przesuwają zawartość, przełączają przełączniki, przesuwają suwaki i przewijają wartości selektora, przeglądają strony książek

Zasady projektowania interfejsu użytkownika

- Kontrola użytkownika:
 - w systemie iOS to użytkownicy, nie aplikacje, mają kontrolę
 - aplikacja może sugerować kierunek działania lub ostrzegać przed niebezpiecznymi konsekwencjami, ale nie ma procesu decyzyjnego.
 - równowaga między umożliwianiem użytkownikom kontroli, a unikaniem niepożądanych wyników
 - aplikacja może sprawić, że użytkownicy wiedzą, że mają kontrolę, zachowując znajome i przewidywalne elementy interaktywne, potwierdzając destrukcyjne działania i ułatwiając anulowanie operacji, nawet gdy są już w toku

Interfejs użytkownika - UIKit

- Większość interfejsów jest budowana z komponentów UIKit
- Aplikacje mają spójny wygląd na urządzeniach Apple
- Elementy aktualizowane są podczas aktualizacji systemu iOS
- Elementy interfejsu UIKit należą do kategorii:
 - paski (ang. bars) – informują, gdzie użytkownik się znajduje, zapewniają nawigację i akcje (np. przyciski)
 - widoki (ang. views) – wyświetlają tekst, grafikę, animacje i elementy interaktywne, zapewniają zachowania, jak przewijanie, wstawianie, usuwanie i rozmieszczanie
 - kontrolki (ang. controls) – inicują działania i przekazują informacje (np. przyciski, przełączniki, pola tekstowe, wskaźniki postępu)

Działanie interfejsu użytkownika

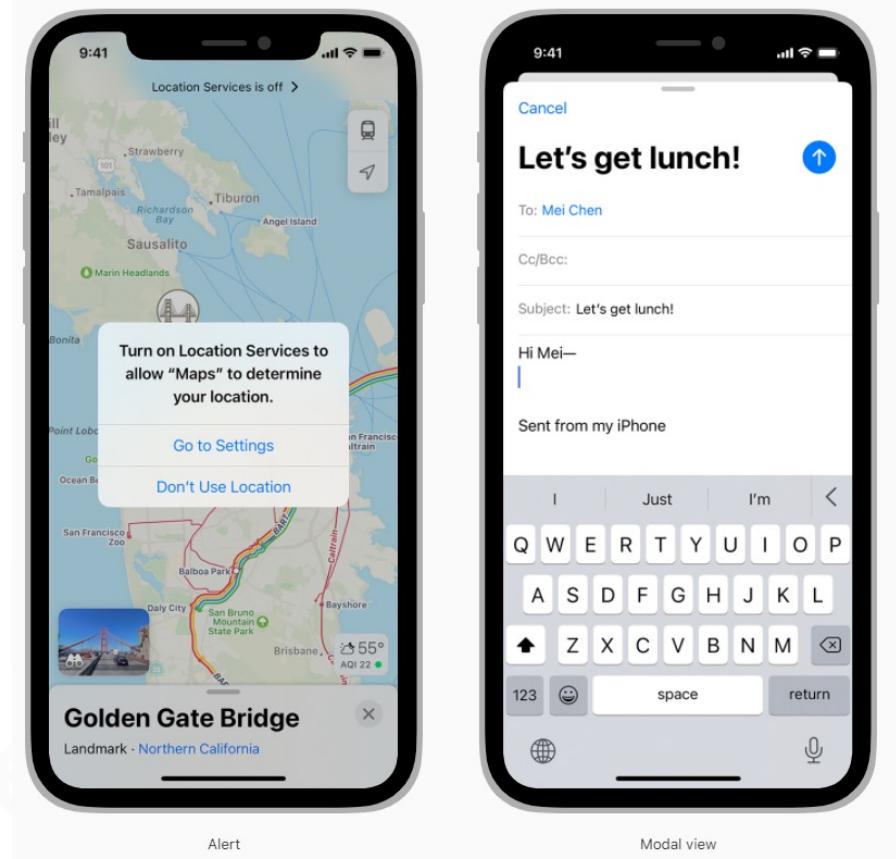
- Uruchomienie aplikacji (ang. launching)
 - szybkie i bezproblemowe
 - ekran startowy i pierwszy ekran aplikacji (początkowa zawartość)
 - uruchomienie aplikacji w odpowiedniej orientacji
 - unikaj proszenia użytkownika o informacje dotyczące konfiguracji (ustawienia dla większości użytkowników, prośba przy pierwszym uruchomieniu)
 - przywrócenie ostatniego stanu aplikacji po ponownym uruchomieniu aplikacji, aby działanie mogło być kontynuowane
 - brak zachęty do ponownego uruchamiania (wrażenie zawodnej aplikacji)
 - brak reklam
 - unikaj częstego pytania o ocenę aplikacji

Działanie interfejsu użytkownika

- Onboarding
 - witanie nowych użytkowników i powracających
- Ładowanie (ang. loading)
 - brak pustego lub statycznego ekranu (wrażenie, że aplikacja jest zablokowana)
 - oznaczenie ładowania danych (np. pasek postępu)
 - jak najszybsze wyświetlenie zawartości
 - podczas ładowania wyświetlenie dodatkowych informacji (np. wskazówek dotyczących rozgrywki, zabawnych sekwencji wideo lub ciekowej grafiki)
 - dostosowane ekranu ładowania, aby pasował do stylu aplikacji

Działanie interfejsu użytkownika

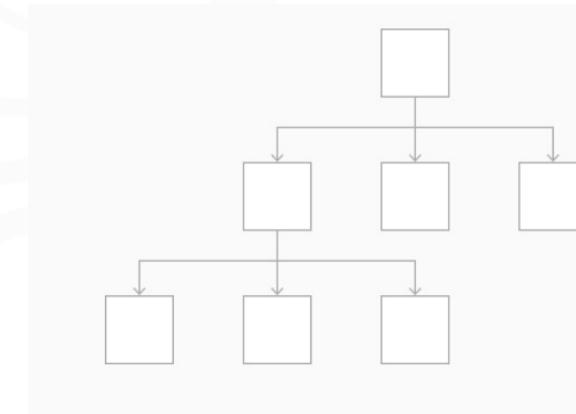
- Modalność (ang. modality)
 - przedstawienie zawartości w trybie tymczasowym, który wymaga wyraźnego działania użytkownika
 - samodzielne zadania dla użytkowników
 - wyświetlenie krytycznych informacji
 - style: alerts, activity views, share sheets, action sheets
 - proste, krótkie zadania modalne
 - dołączaj przycisk, który kończy widok modalny



źródło: <https://developer.apple.com/design/human-interface-guidelines/ios/app-architecture/modality/>

Działanie interfejsu użytkownika

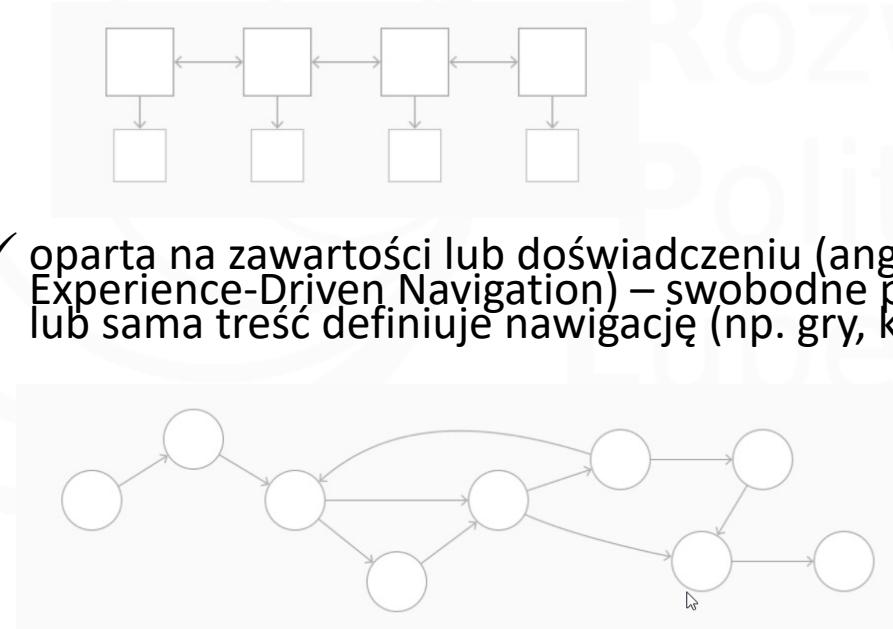
- Nawigacja (ang. navigation)
 - ma wspierać strukturę i cel aplikacji bez zwracania na siebie uwagi
 - powinna być naturalna i znajoma, nie powinna dominować w interfejsie ani odciągać uwagi od treści
 - w iOS istnieją trzy główne style nawigacji:
 - ✓ hierarchiczna (ang. Hierarchical Navigation) – jeden wybór na każdym ekranie (np. ustawienia, czy poczta)



źródło:
<https://developer.apple.com/design/human-interface-guidelines/ios/app-architecture/navigation/>

Działanie interfejsu użytkownika

- Nawigacja (ang. navigation)
 - w iOS istnieją trzy główne style nawigacji:
 - ✓ płaska (ang. Flat Navigation) – przełączanie między wieloma kategoriami
 - ✓ oparta na zawartości lub doświadczeniu (ang. Content-Driven or Experience-Driven Navigation) – swobodne poruszanie się po treści lub sama treść definiuje nawigację (np. gry, książki)

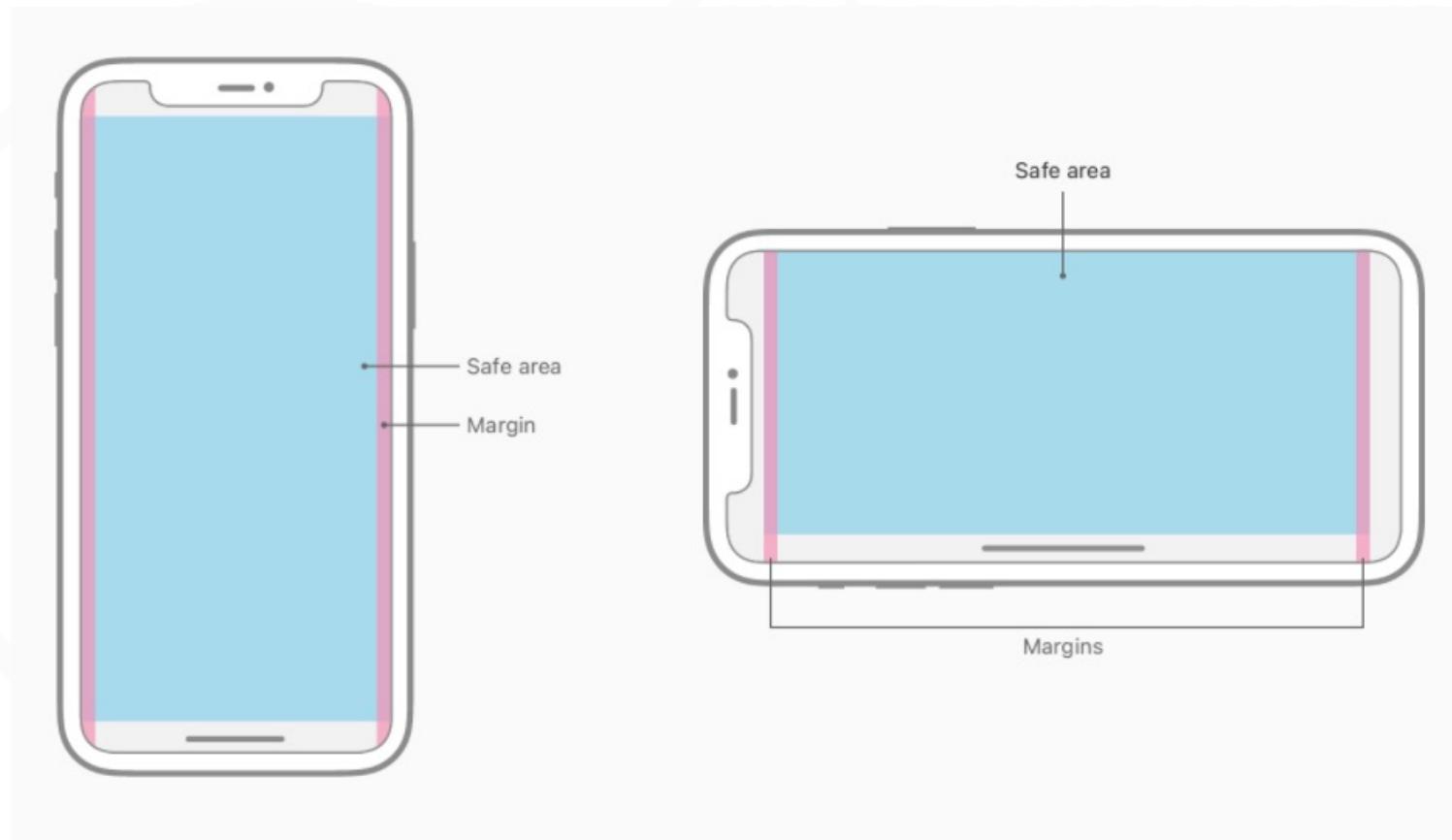


źródło:
<https://developer.apple.com/design/human-interface-guidelines/ios/app-architecture/navigation/>

Rozmieszczenie elementów

- Układ (ang. layout) definiuje prostokątny region, który pomaga pozycjonować, wyrównywać i rozmieszczać zawartość na ekranie
- Stosuje standardowe marginesy wokół treści, które ograniczają szerokość tekstu w celu uzyskania optymalnej czytelności
- Bezpieczny obszar definiuje obszar w widoku, który nie jest objęty paskiem nawigacji, paskiem kart, paskiem narzędzi ani innymi widokami, które może zapewnić kontroler widoku

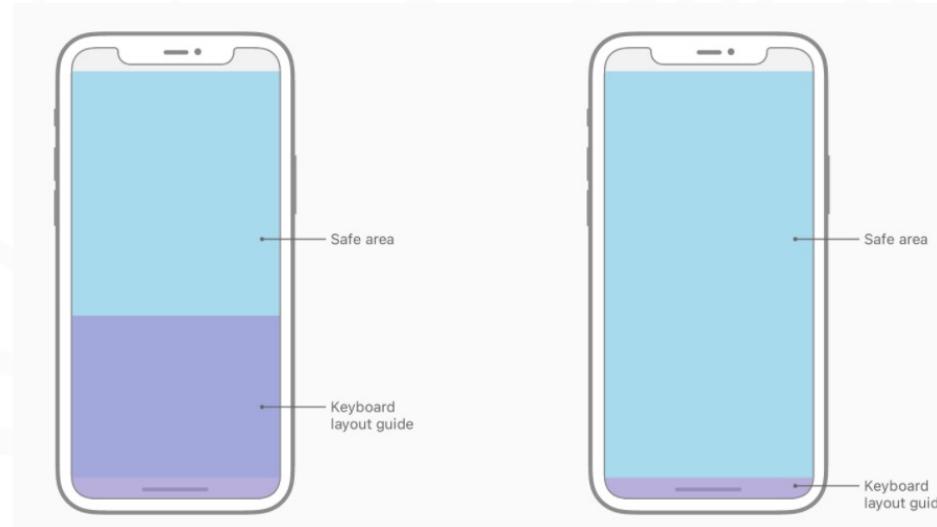
Rozmieszczenie elementów



źródło: <https://developer.apple.com/design/human-interface-guidelines/ios/visual-design/adaptivity-and-layout/>

Rozmieszczenie elementów

- System iOS 15 i nowsze definiuje rozmieszczenie klawiatury, który reprezentuje miejsce, które klawiatura aktualnie zajmuje, i uwzględnia wstawki obszaru bezpiecznego



źródło: <https://developer.apple.com/design/human-interface-guidelines/ios/visual-design/adaptivity-and-layout/>

Rozmieszczenie elementów

- Rozmieszczenie elementów:
 - podstawowa zawartość jest przejrzysta i w odpowiednim rozmiarze
 - wizualna waga dla rozróżnienia ważnych i mniej ważnych elementów
 - stosowanie wyrównywania tekstu – hierarchia i organizacja danych
 - obsługa pionowej i poziomej orientacji w aplikacji
 - obsłużenie zmiany rozmiaru czcionki, po wybraniu odpowiednich opcji z ustawień

UIKit vs SwiftUI

UIKit

- Imperatywny szkielet programowania
- Wywołanie funkcji po kliknięciu np. przycisku
- Śledzenie stanu aktualizacji interfejsu użytkownika
- Brak aktualizacji użytkownika -> operowanie na błędnych danych

SwiftUI

- Deklaratywny szkielet programowania
- Dane mogą być automatycznie powiązane z elementami interfejsu użytkownika
- Przejmuje zadanie wyświetlenia aktualnych danych w interfejsie użytkownika

UIKit vs SwiftUI

UIKit

- Widok dziedziczy po klasie *UIView*, która ma wiele właściwości i metod
- Klasa widoku dziedziczy po *UIView*, istnieje wiele funkcji, które można potrzebować (np. kolor tła, warstwa do renderowania zawartości)

SwiftUI

- Widoki są strukturą, a nie klasą
- Widoki można prawie dowolnie tworzyć
- Struktura widoku jest zgodna tylko z protokołem widoku
- Brak dodatkowych wartości dziedziczonych z klas nadzędnych

UIKit vs SwiftUI

UIKit

- Interface Builder
- Aplikacja nie będzie działała poprawnie, jeśli nie będzie @IBOutlet przypisanego do konkretnej zmiennej
- AutoLayout
- Brak Live Preview
- MVC

SwiftUI

- Live Preview
- Nie potrzebuje Interface Buider
- Brak AutoLayout
- Do ułożenia elementów stosowane są : HStack, VStack, ZStack, Groups, Lists
- MVVM

UIKit vs SwiftUI

- Paradygmat imperatywny:
 - sekwencja poleceń zmieniających krok po kroku stan maszyny, aż do uzyskania oczekiwanej wyniku
 - używa instrukcji zmieniających stan programu
 - np. wywołanie akcji po naciśnięciu przycisku, wywołanie akcji po nasłuchaniu zdarzenia
- Paradygmat deklaratywny:
 - opis warunków końcowych, które powinny być spełnione, żeby możliwe było uzyskanie rozwiązania
 - „co chcemy otrzymać”, a nie „jakie kroki należy wykonać”
 - brak sekwencji instrukcji modyfikujących stan maszyny krok po kroku

UIKit vs SwiftUI

- Paradygmat imperatywny – UIKit
 - modyfikacja tekstu wyświetlanego w etykiecie po edycji tekstu w polu tekstowym

```
@objc func textFieldDidChange(_ textField:  
UITextField) {  
  
    guard let newText = textField.text else  
    {  
  
        return  
  
    }  
  
   .textLabel?.text = "Your name is  
        \\\n(newText)"  
  
}
```

UIKit vs SwiftUI

- Paradymat deklaratywny – SwiftUI

```
struct ContentView: View {  
    @State private var name = ""  
  
    var body: some View {  
        Form {  
            TextField("Enter your name", text:  
                     $name)  
            Text("Your name is \(name)")  
        }  
    }  
}
```

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
INFORMATYKA



Zintegrowany
Program
Rozwoju
Politechniki
Lubelskiej -
część druga

Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego



Fundusze
Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



POLITECHNIKA LUBELSKA

WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI

INFORMATYKA



Zintegrowany
Program
Rozwoju
Politechniki
Lubelskiej -
część druga

Programowanie aplikacji mobilnych na platformę iOS

W3
Wzorzec Model-Widok-Kontroler

Maria Skublewska-Paszkowska



**Fundusze
Europejskie**
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



Agenda

- Wzorce projektowe
- Model-Widok-Kontroler
- Model-Widok-WidokModel

Zintegrowany
Program
Rozwoju
Politechniki
Lubelskiej -
część druga

Wzorce projektowe

- Wzorce projektowe (ang. design patterns)
 - uniwersalne, sprawdzone w praktyce rozwiązanie często pojawiających się, powtarzalnych problemów projektowych
 - przedstawia powiązania i zależności pomiędzy klasami oraz obiektami i ułatwia tworzenie, modyfikację oraz utrzymanie kodu źródłowego
 - są opisem rozwiązań, a nie ich implementacją
 - sformalizowane najlepsze praktyki, które programista może wykorzystać do rozwiązywania typowych problemów podczas projektowania aplikacji lub systemu
 - stosowane są w projektach wykorzystujących programowanie obiektowe

Wzorce projektowe

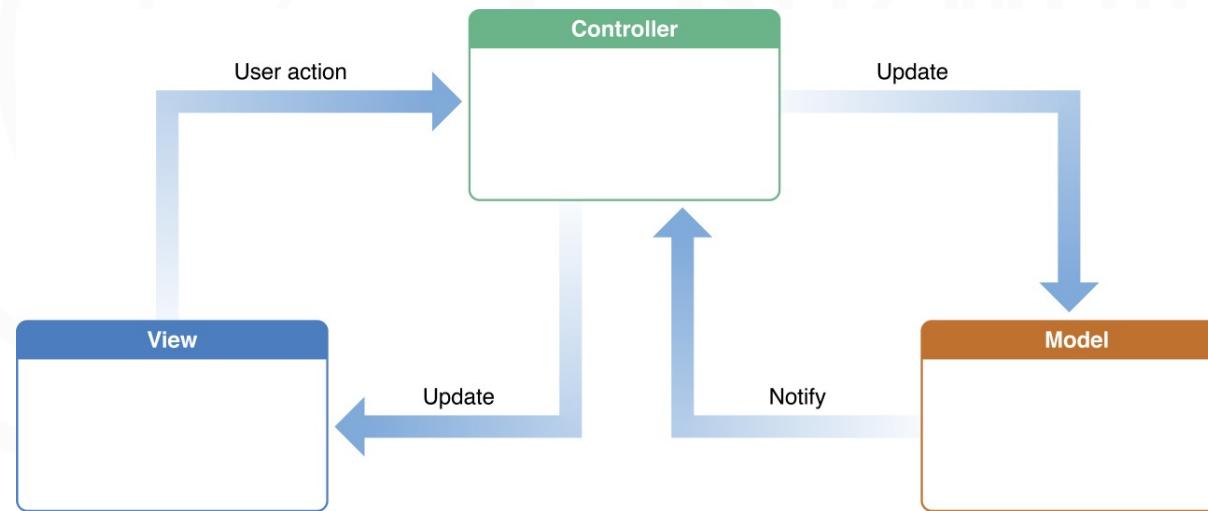
- Zalety stosowania wzorców projektowych:
 - zapewniają sprawdzone rozwiązania
 - ujednolicenie kodu – zapewniają typowe rozwiązania, które zostały przetestowane pod kątem błędów (mniej błędów podczas projektowania architektury aplikacji)
 - wspólne nazewnictwo – dany wzorzec projektowy jest skojarzony z konkretnym rozwiązaniem
 - wyjaśniają rozwiązanie danego problemu
 - omawiane są na przykładach implementacji w danym języku programowania
 - bazują na doświadczeniach nabytych w przeszłości

Wzorce projektowe

- Wzorce projektowe dla języka Swift:
 - kreacyjne – zajmują się mechanizmami tworzenia obiektów, opierając się na hermetyzacji oraz ukrywania sposobu tworzenia i łączenia instancji tych klas (np. Factory, Abstract, Singleton, Builder)
 - strukturalne – ułatwiają proces projektowania aplikacji poprzez znalezienie łatwego sposobu na zaimplementowanie relacji między elementami, definiując jak łączyć obiekty i klasy w większe struktury przy zachowaniu elastyczności i wydajności tych struktur (np. Fasada, Adapter, MVC)
 - behawioralne – identyfikują wspólne wzorce komunikacji między obiektami, zwiększając w ten sposób ich elastyczność, poprzez algorytmy i przypisywaniem odpowiedzialności pomiędzy obiektami (np. Template Method, Observer i Memento)
- Wzorce projektowe Swift (iOS) są takie same jak w innych językach i są używane w standardowych przypadkach użycia

Model-Widok-Kontroler

- Model-View-Controller (MVC)
 - jeden z najczęściej stosowanych wzorców projektowych
 - zalecany przez firmę Apple do tworzenia struktury aplikacji systemu iOS



źródło:<https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html>

Model-Widok-Kontroler

- Model-View-Controller (MVC)

- model – niezależny od interfejsu użytkownika, zarządza danymi, logiką i regułami aplikacji
- widok – opisuje, jak wyświetlić określoną część modelu w interfejsie użytkownika
- kontroler – akceptuje dane wejściowe od użytkownika i reaguje na jego działania, zarządzając aktualizacjami modeli i odświeżając widoki, łączący pozostałe warstwy

Model-Widok-Kontroler

- Model-View-Controller (MVC) – model:
 - hermetyzuje dane i definiuje logikę aplikacji
 - komunikacja między danymi w obrębie modelu jest zdefiniowana
 - obiekt może reprezentować figurę, postać lub kontakt w książce adresowej
 - istnieją relacje między obiekty:
 - jeden do jednego
 - jeden do wielu
 - większość danych to stałe elementy programu (takie jak pliki, czy baza danych) i powinny należeć do warstwy modelowej po uruchomieniu aplikacji

Model-Widok-Kontroler

- Model-View-Controller (MVC) – model:
 - nie powinno być bezpośredniego połączenia między warstwami modelu i widoku
 - nie powinno być bezpośredniego połączenia między interfejsem użytkownika, a modelem
 - jedyna komunikacja, która powinna być wykonywana, odbywa się za pośrednictwem warstwy kontrolera
 - rezultatem działań wykonanych przez użytkownika w warstwie widoku (np. tworzenie lub modyfikacja danych) jest dodanie lub modyfikacja obiektu modelu
 - jeśli obiekt modelu zostanie zmodyfikowany, obiekt kontrolera zmieni właściwy obiekt widoku

Model-Widok-Kontroler

- Model-View-Controller (MVC) – widok:
 - ma kontakt z użytkownikiem – reaguje na akcje wykonywane przez użytkownika
 - głównym celem tych obiektów jest wyświetlanie danych z modelu i zarządzanie nim
 - obiekty są odseparowane od warstwy modelowej i zapewniają spójność między aplikacjami
 - struktury UIKit i AppKit zapewniają zestaw klas widoków
 - Interface Builder oferuje wiele obiektów widoku w bibliotece
 - obiekty są również informowane o zmianach, jakie nastąpiły w danych modelu za pośrednictwem obiektów kontrolera

Model-Widok-Kontroler

- Model-View-Controller (MVC) – kontroler:
 - działa jako połączenie między jednym lub większą liczbą obiektów widoku aplikacji, a jednym lub większą liczbą obiektów modelu
 - obiekty tworzą link, przez który obiekty widoku są informowane o zmianach w obiektach modelu i odwrotnie
 - mogą również konfigurować, koordynować koordynację zadań dla aplikacji oraz zarządzać cyklem życia innych obiektów
 - obiekt interpretuje zadania wykonywane przez obiekty widoku i tworzy nowe dane lub modyfikuje je z warstwy modelu
 - gdy obiekty modelu zmienią się, obiekty kontrolera poinformują obiekty widoku o nowych danych lub o danych do modyfikacji

Model-Widok-Kontroler

- Model w aplikacji iOS:
 - kod dot. sieci (Network)
 - Persistence code: bazy danych, Core Data, dane na urządzeniu
 - Parsing code: JSON encoding/decoding
 - klasy do zarządzania i klasy abstrakcyjne
 - dane źródłowe i delegaty
 - rozszerzenia
 - walidacja danych
 - *XYZModel.swift*

Model-Widok-Kontroler

- Widok w aplikacji iOS:
 - klasy dziedziczące po *UIView* (podstawowe i złożone widoki)
 - klasy należące do UIKit/AppKit
 - Core Animation
 - Core Graphics
 - *XYZView.swift*
- Kontroler w aplikacji iOS:
 - klasy dziedziczące po *UIViewController*
 - stosuje elementy w warstwie modelu do zdefiniowania przepływu informacji w aplikacji
 - decyduje w jaki sposób ma zachować się aplikacja:
 - ✓ jak często należy odświeżyć aplikację
 - ✓ dostęp do danych/sieci
 - ✓ czy aplikacja ma działać w tle
 - *XYZController.swift*

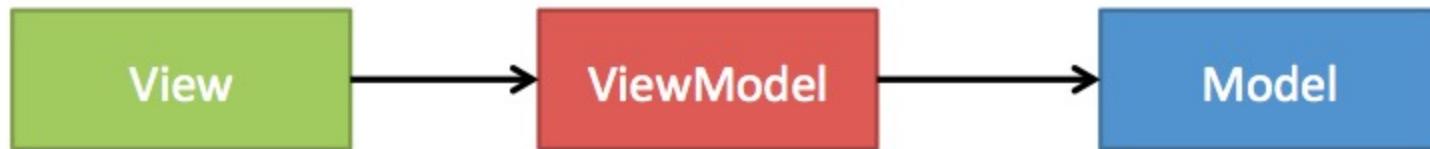
Model-Widok-Kontroler

- Problem implementacji w aplikacji iOS:
 - niektóry kod nie należy ani do warstwy widoku ani warstwy modelu – co zrobić? Dodać go do kontrolera?
 - przykład: model, czy kontroler powinien zarządzać formatowaniem danych?
 - model nie powinien zarządzać widokiem
 - widok powinien tylko wyświetlać dane, a nie nimi zarządzać
 - problem: przeciążone kontrolery
 - Czy MVC jest najlepszym rozwiązaniem?

Model-Widok-WidokModel

- Model-View-ViewModel (MVVM):
 - model
 - widok
 - widok modelu – tłumaczy dane z modelu na wartości, które mogą być wyświetcone w widokach
 - kontroler

ViewController + View (w Storyboard, XIB lub kodzie), który działa jako widok MVVM



źródło: <https://www.raywenderlich.com/4161005-mvvm-with-combine-tutorial-for-ios>

Model-Widok-WidokModel

- Model-View-ViewModel (MVVM):
 - model – reprezentuje te same dane, jak w przypadku modelu MVC
 - widok – jest reprezentowany przez UIView lub UIViewController razem z plikami .xib i .storyboard
 - widok modelu – ukrywa asynchroniczny kod sieciowy, kod przygotowania danych do prezentacji wizualnej i kod nasłuchujący zmian modelu; kod jest ukryty za zdefiniowanym interfejsem API, dopasowanym do tego konkretnego widoku
- Jedną z korzyści MVVM jest testowanie
 - ViewModel jest często obiektem NSObject (lub np. strukturą) i nie jest połączony z kodem UIKit
 - można go łatwiej przetestować w testach jednostkowych bez wpływu na kod interfejsu użytkownika

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
INFORMATYKA



Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego



**Fundusze
Europejskie**
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny

