POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Automatyka i Robotyka (W4)
SPECJALNOŚĆ: Technologie Informacyjne w Systemach Automatyki (ART)

PROJEKT INŻYNIERSKI

Fotolicznik - aplikacja mobilna dla systemu iOS z przetwarzaniem obrazu

Photometer - mobile image processing application for iOS

AUTOR:
Wojciech Frątczak

PROWADZĄCY PRACĘ:
Dr inż. Piotr Ciskowski

OCENA PRACY:

WROCŁAW 2016

Spis treści:

1. Wstęp

W dzisiejszych czasach należy zwrócić uwagę na szczególnie dynamiczny rozwój technologii mobilnych. Jesteśmy świadkami swoistej rewolucji mobilnej. Tablety, smartfony oraz inteligentne zegarki towarzyszą nam na każdym kroku. Wiąże się to z wieloma czynnikami. Jednym z nich jest to, że korzystanie z urządzeń mobilnych jest szybsze i łatwiejsze niż z tradycyjnych komputerów stacjonarnych lub laptopów. Następną zaletą korzystania z takich urządzeń jest możliwość wręcz błyskawicznego połączenia z Internetem w niemal każdym miejscu, oczywiście w miarę dostępu do niego za pomocą sieci komórkowej lub Wi-Fi.

Co więcej, postęp technologiczny pozwala wykorzystywać do budowy smartfonów coraz to lepsze podzespoły i obecne urządzenia posiadają wielokrotnie większe możliwości od pierwszych urządzeń mobilnych oraz porównywalne do komputerów stacjonarnych sprzed kilku lat. Wspomniana wyżej przeze mnie popularyzacja mobilnych technologii spowodowała stworzenie ogromnej ilości aplikacji, które pomagają nam w codziennym życiu. Publikowane są one w sklepach dostępnych dla odpowiednich systemów operacyjnych. Aplikacje służą do komunikacji z znajomymi, robienia zakupów, zarządzania czasem. Prawdopodobnie dla większości, nawet najbardziej abstrakcyjnych pomysłów powstały już mniej lub bardziej udane próby implementacji aplikacji do tego służących.

Aplikacje tego typu zaczęły również powstawać w celu wsparcia zarządzania czasem, budżetem domowym, lub innymi zasobami. Jednym z takich przypadków jest zarządzanie stanami liczników wody, prądu bądź gazu, które są obecne w każdym mieszkaniu. Oprócz standardowych funkcjonalności, takich jak wprowadzanie stanu licznika, przetrzymywanie i zarządzanie nim w bazie danych, bardzo ciekawym rozwiązaniem okazuje się wykorzystanie urządzenia jako czytnika.

2. Analiza projektu

W ramach tej pracy inżynierskiej powstanie aplikacja mobilna na system iOS zarządzająca stanami liczników posiadanych przez użytkownika. Zaimplementowane zostanie przetwarzanie obrazów umożliwiające rozpoznanie konkretnego licznika a następnie rozpoznanie jego stanu. Użytkownik będzie miał również możliwość przechowywania zapisanych wartości oraz ich wizualizacji na przestrzeni czasu.

2.1. Warstwa aplikacji

Aplikacja zostanie napisana na platformę iOS. System ten jest jednym z dwóch obecnie najpopularniejszych systemów dostępnych na urządzenia mobilne obok systemu Android. Aplikacja będzie natywna, dedykowana tylko i wyłącznie na konkretny system.

2.1.1. System

System iOS jest systemem operacyjnym firmy Apple dla urządzeń mobilnych iPhone, iPod touch oraz iPad. Pierwszy system, na którym bazuje obecny, został przedstawiony przez firmę z Cupertino 6. marca 2008 roku pod nazwą iPhone OS. Następnie w miarę ewolucji i udoskonalania systemu przyjął on nazwę iOS. Bazuje on na systemie operacyjnym Mac OS X 10.5 oraz jądrze Darwin.

Każdy członek programu deweloperskiego może korzystać z iOS SDK w celu stworzenia swojej aplikacji. Proces publikowania tworzenia i publikowania jest ściśle kontrolowany przez firmę Apple. Do publikacji potrzebujemy posiadać odpowiednie konto deweloperskie aktywowane po uiszczeniu odpowiedniej rocznej opłaty. Deweloperzy mogą korzystać z zintegrowanego środowiska programistycznego Xcode. Dostępny jest darmowo wraz z system OS X i umożliwia tworzenie oprogramowania na systemy: OS X, watchOS, iOS oraz tvOS. Xcode posiada możliwość

kompilacji kodów źródłowych języków: C, C++, Objective-C++, Objective-C oraz Swift. Program posiada również wiele ciekawych usprawnień i urządzeń ułatwiających wytwarzania oprogramowania.

System iOS składa się z czterech warstw kolejno odpowiadających za różne elementy działania, od interakcji z sprzętem po interfejs użytkownika:

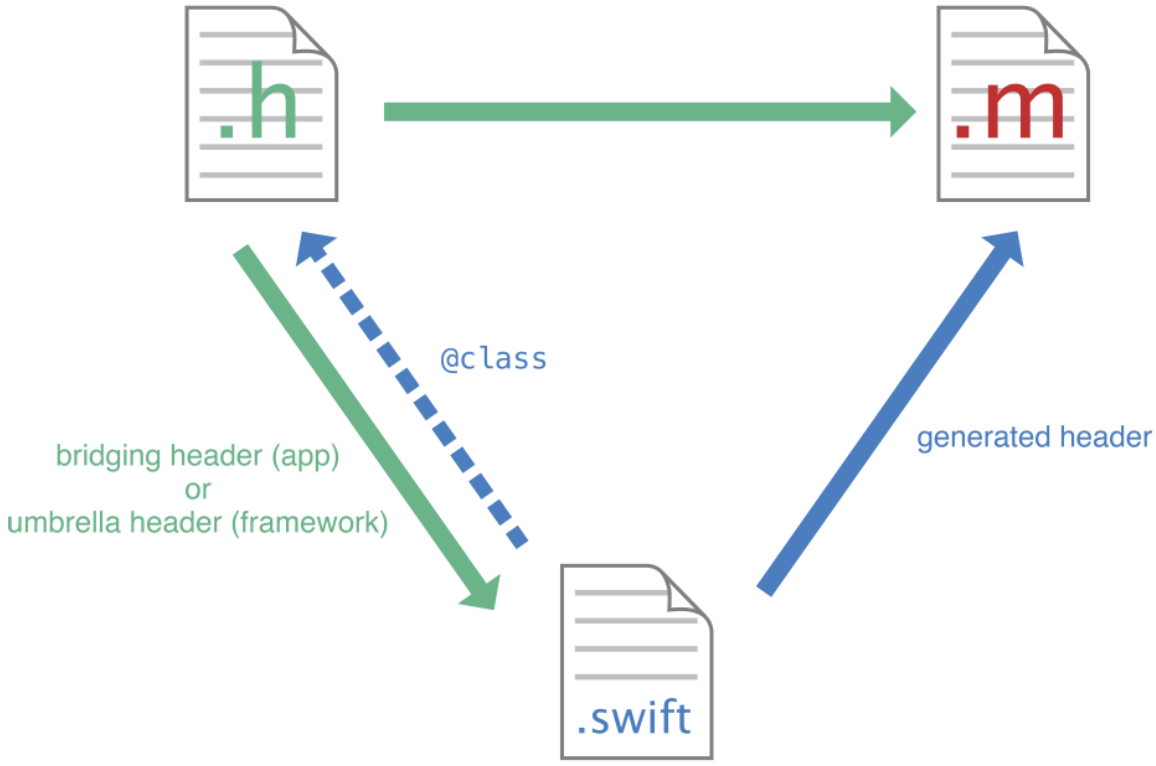
- Core OS – komunikacja z portami wejścia i wyjścia urządzenia, zabezpieczenia aplikacji
- Core Services – obsługa usług systemowych, w tym wielowątkowość, bazy danych, lokalizacja
- Media – obsługa multimediów użytych w aplikacji, w tym generowanie dźwięków, grafik, animacji i wideo
- Cocoa Touch – interfejs graficzny aplikacji, w tym wielozadaniowość, notyfikacje push, obsługa gestów

2.1.2. Języki programowania

Aplikacja zostanie napisana w swego rodzaju hybrydzie języków programowania. Zdecydowanie największa część projektu zostanie napisana w języku Swift. Jest to nowoczesny język programowania stworzony przez firmę Apple i pokazany pierwszy raz światu podczas Worldwide Developers Conference 2 czerwca 2014 roku. Od tego momentu dość dynamicznie wypiera z rynku język Objective-C wcześniej używany do tworzenia oprogramowania na system iOS. Przez twórców jest reklamowany jako „Potężny język, który jednocześnie jest łatwy do nauki”[1]. W mojej aplikacji wykorzystam język Swift w wersji 3.0.

Z uwagi na to, że aplikacja będzie wykorzystywać bibliotekę OpenCV napisaną w języku C++, dla potrzeb integracji języka Swift oraz C++ w jednym projekcie, część kodu zostanie również napisana w języku Objective-C.

W ten sposób powstaje hybryda języków Objective-C oraz C++ nazywana Objective-C++. Pozwoliło to mi na wykorzystywanie biblioteki OpenCV wewnątrz klasy napisanej w języku Objective-C. Następnie integracja języka Objective-C z językiem Swift jest banalanie prosta przy pomocy pliku Bridging Header automatycznie stworzonego przez środowisko Xcode. W ten sposób otrzymujemy aplikację napisaną w języku Swift wykorzystującą kod napisany w języku C++.



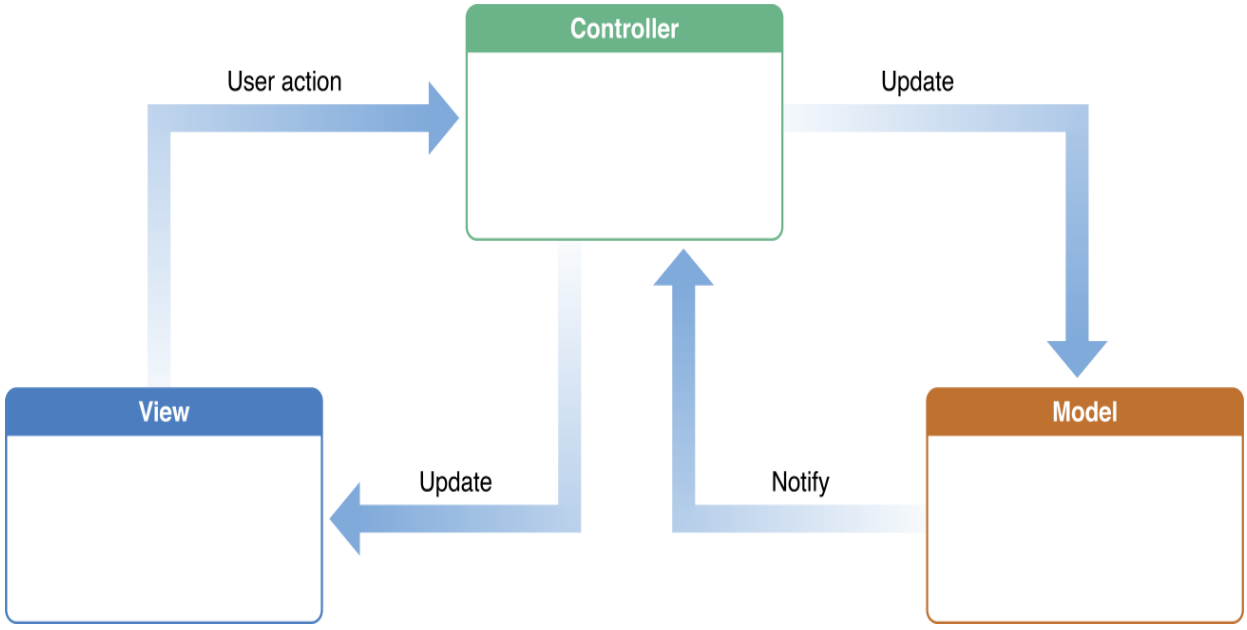
Rysunek 1 Model importowania plików w obu językach programowania[6]

2.1.3. Architektura aplikacji

Projekt zostanie stworzony wykorzystując architekturę Model View Controller – w skrócie MVC. Jest to wzorzec architektoniczny służący organizowania struktury aplikacji. Definiuje on role obiektów a aplikacji.

Wyodrębnia on trzy warstwy posiadające odpowiednio różne odpowiedzialności:

- Model – warstwa modelowa i logika aplikacji
- Widok – interfejs użytkownika
- Kontroler – komunikacja pomiędzy widokiem oraz modelem



Rysunek 2 Model komunikacji między warstwami w architekturze MVC [10]

Z korzystania z owej architektury płynie wiele benefitów i jest rekomendowana przez firmę Apple do tworzenia aplikacji.

2.2. Rozpoznawanie licznika

Aplikacja będzie miała zaimplementowaną funkcjonalność rozpoznawania licznika na podstawie dodanych do projektu wzorcowych zdjęć wszystkich liczników dostępnych w mieszkaniu użytkownika. Do przechwycenia obraz licznika wykorzystany zostanie aparat urządzenia.

Jest to możliwe dzięki bibliotece AVFoundation udostępnionej dla programistów. Pozwala ona na pracę w czasie rzeczywistym z elementami audiowizualnymi smartfona, takimi jak mikrofon lub kamera.

Do przetwarzania obrazu przechwyconego z kamery posłużyłem się biblioteką OpenCV. Jest to potężna biblioteka funkcji wykorzystywanych do obróbki obrazu. OpenCV jest projektem multiplatformowym o charakterze open source.

Do rozpoznawania licznika zostanie wykorzystana metoda porównywania histogramów. Użytkownik będzie miał możliwość dodawania nieskończonej liczby liczników oraz do każdego z nich wzorcowego zdjęcia. Napisany przeze mnie program porówna przechwycony obraz za pomocą kamery do każdego zdjęcia wzorcowego. Obraz z najlepszym współczynnikiem dopasowania zostanie uznany za rozpoznany licznik. Wewnątrz aplikacji operujemy na obiektach klasy UIImage. Jest to klasa dostarczona przez framework UIKit do zarządzania danymi obrazu. Biblioteka OpenCV dostarcza nam odpowiednie metody do przetworzenia obiektu klasy UIImage na obiekt klasy cv::Mat, który z łatwością możemy przetworzyć przy pomocy OpenCV. Do porównania histogramów dwóch obrazów wykorzystałem metodę compareHist. Przyjmuje ona trzy argumenty:

- H1 – pierwszy histogram
- H2 – drugi histogram
- Metoda – metoda, którą chcemy porównać oba histogramy.

Do porównania histogramów możemy skorzystać z 4 metod[5]:

- Metoda korelacji (correlation)

$$d(H_1, H_2) = \frac{\sum_I (H_1(I) - \bar{H_1})(H_2(I) - \bar{H_2})}{\sqrt{\sum_I (H_1(I) - \bar{H_1})^2 \sum_I (H_2(I) - \bar{H_2})^2}}$$

gdzie:

$$\bar{H_k} = \frac{1}{N} \sum_J H_k(J)$$

oraz N jest całkowitą liczbą przedziałów klasowych.

- Chi-kwadrat (Chi-square)

$$d(H_1, H_2) = \sum_I \frac{(H_1(I) - H_2(I))^2}{H_1(I)}$$

- Metoda przecięcia (intersection)

$$d(H_1, H_2) = \sum_I \min(H_1(I), H_2(I))$$

- Metoda odległości Bhattacharyya (Bhattacharyya distance) – OpenCV faktycznie oblicza odległość Hellingera, która jest powiązana z współczynnikiem Bhattacharyya

$$d(H_1, H_2) = \sqrt{1 - \frac{1}{\sqrt{\bar{H_1}\bar{H_2}N^2}} \sum_I \sqrt{H_1(I) \cdot H_2(I)}}$$

W swoim projekcie wykorzystam wszystkie wyżej wymienione metody. Obraz wzorcowy, który będzie miał najlepszy współczynnik w wszystkich (bądź w większości) metodach zostanie uznany za rozpoznany obraz.

2.3. Rozpoznawanie stanu licznika

Rozpoznawanie stanu licznika zostanie przeprowadzone przy pomocy biblioteki SwiftOCR. Jest to dedykowany framework na system iOS. SwiftOCR jest szybką i prostą w obsłudze biblioteką OCR (Optical Character Recognition) napisaną w języku Swift. Wykorzystuje ona również sieci neuronowe. Biblioteka jest zoptymalizowana w ten sposób, by rozpoznawać krótkie, jednoliniowe cody

alfanumeryczne, więc dobrze sprawdza się do rozpoznawania stanu licznika. SwiftOCR wypada niezwykle dobrze na tle konkurencyjnego Tesseract. Autor użytej przeze mnie biblioteki dokonał testu obu bibliotek na 50 ciężkich do rozpoznania kodów alfanumerycznych. Wynik był niezwykle korzystny dla SwiftOCR.

	SwiftOCR	Tesseract
Prędkość działania	0,08 s	0,63 s
Precyzja	97,7%	45,2%
Obciążenie procesora	~30%	~90%
Wykorzystanie pamięci	45 MB	73 MB

Tabela 1 Porównanie wydajności bibliotek SwiftOCR oraz Tesseract[7]

Jak widzimy, SwiftOCR jest naturalnie najlepszym wyborem dedykowanym na system iOS, który wspomaga rozpoznawanie krótkich ciągów znaków.

3. Opis projektu

Aplikacja ma pełnić rolę pewnego rodzaju menadżera do zarządzania stanami liczników posiadanych przez użytkownika w mieszkaniu. Ma pozwolić na ułatwienie w regularnym spisywaniu stanu liczników oraz magazynowania ich. Wizualizacja zmagazynowanych danych pozwala na analizę i optymalizację zużycia zasobów takich jak: prąd, woda czy gaz. Aplikacja składa się z czterech zakładek, dzięki którym łatwo możemy nawigować pomiędzy widokami odpowiedzialnymi za różne funkcjonalności aplikacji.

3.1. Lista liczników

Omawiana część aplikacji odpowiedzialna jest za wyświetlenie wszystkich liczników posiadanych przez użytkownika. Lista liczników

zaczerpnięta jest z lokalnej bazy danych. Oczywiście istnieje możliwość edycji każdego licznika. Możemy odpowiednio usunąć lub dodać nowy licznik do listy. Jest to możliwe za pomocą przycisków u góry ekranu. Jeśli chcemy dodać nowy licznik, konieczne jest podanie jego nazwy. Kliknięcie któregokolwiek licznika z listy przenosi nas do widoku szczegółów. Wyświetla się tam jego nazwa oraz wzorcowe zdjęcie licznika, które oczywiście możemy dodać (jeśli uprzednio nie zostało dodane) lub zmienić. Dodane tam wzorcowe zdjęcie zostanie użyte przy rozpoznawaniu licznika ze zdjęcia zrobionego kamerą urządzenia. Jest to prosta a zarazem bardzo funkcjonalna część projektu. Pozwala na przeglądanie wszystkich liczników i sprawne zarządzanie nimi.

3.2. Dodawanie pomiaru

Jest to część odpowiedzialna za najważniejszą funkcjonalność aplikacji. Pozwala ona na rozpoznanie stanu licznika i dodanie go do historii pomiarów konkretnego licznika. Zaczynamy od rozpoznanie licznika, do którego chcemy dodać pomiar. Na wstępie zostaje nam przedstawiony obraz z kamery z suwakiem odpowiadającym za przybliżenie obrazu. Po naciśnięciu przycisku „Take photo” przechwytywany jest obraz i przetworzony w celu rozpoznania konkretnego licznika. Po upływie krótkiej chwili zostaje wyświetlony komunikat z nazwą rozpoznanego licznika. Mamy możliwość potwierdzenia, lub zrobienia zdjęcia ponownie, jeśli aplikacja rozpoznała licznik błędnie. Następnie zostajemy przeniesieni do identycznego widoku, lecz posiada on już inną odpowiedzialność. Przechwytuje on obraz w celu rozpoznania już stanu licznika. Na obraz z kamery nałożony jest wąski przyciemniony pasek, który należy skierować na obszar stanu licznika. Po naciśnięciu przycisku „Take photo” aplikacja zachowuje się identycznie. Jeśli rozpoznanie zostało przeprowadzone pomyślnie, stan licznika zostaje zapisany do bazy danych przypisany do konkretnego licznika. Omawiana część aplikacji jest bardzo intuicyjna. Nie wymaga od użytkownika dużej interakcji, a jedynie dwukrotnego wykonania zdjęcia, co skutkuje na końcu dodaniem nowego rekordu do bazy danych.

3.3. Wizualizacja wyników

Trzecia część aplikacji odpowiada za wizualizację wyników. Użytkownik może wybrać jeden z listy dostępnych liczników i zobaczyć wizualizację wszystkich wyników na przestrzeni czasu. Dodatkowo, istnieje możliwość wyboru okresu czasu, którego ma dotyczyć wizualizacja pomiarów. Jest to bardzo użyteczna funkcjonalność pozwalająca na analizę zużycia odpowiednich zasobów.

3.4. Informacje o projekcie

Ostatnia część aplikacji odpowiedzialna jest za przedstawienie informacji o aplikacji i jego autorze. Dodatkowo użytkownik posiada możliwość skontaktowania się z autorem aplikacji.

4. Implementacja projektu

4.1. Implementacja części bazodanowej

4.1.1. Technologia

W aplikacji została zaimplementowana baza danych Realm. Jest to alternatywa dla powszechnie używanych baz danych Core Data, czy SQLite. Realm jest projektem typu open source. Wokół projektu powstała bardzo duża społeczność biorąca udział w jego rozwijaniu. Obecnie baza danych Realm posiada kilka milionów użytkowników. Można ją uruchomić bezpośrednio na urządzeniu, dla którego tworzymy aplikację. Realm powiada wsparcie dla bardzo wielu platform oraz języków, w tym również dla systemu iOS w językach Swift oraz Objective-C. Do głównych zalet wykorzystanej technologii należą:

- Wsparcie działania aplikacji bez dostępu do Internetu
- Szybkie tworzenie zapytań do bazy danych

- Dostęp do danych z wielu wątków
- Wsparcie dla wielu platform
- Możliwość szyfrowania danych
- Wsparcie dla programowania reaktywnego

4.1.2. Implementacja obiektów bazy danych

W aplikacji stworzyłem dwa rodzaje obiektów zapisywalne do bazy danych. Są to klasy Meter oraz MeterValue, Reprezentują one odpowiednio liczniki dodane do aplikacji oraz odczytane wartości liczników. Wewnątrz operujemy właśnie na obiektach ww. klas. W celu kompatybilności klas modelowych z bazą danych Realm muszą one dziedziczyć po klasie o nazwie Object. Jest to klasa dostarczająca możliwość zapisu oraz wczytywania obiektów z bazy danych Realm obiektów przy minimalnej ilości kodu. Każde dodane pole wewnątrz klasy dziedziczącej po klasie Object zostanie odnotowana w bazie danych bez dodatkowej implementacji.

4.1.3. Zapis i odczyt obiektów

W celu zapisu (bądź wczytania) obiektów do bazy danych należy stworzyć instancję klasy Realm. Sam zapis obiektów jest bardzo prosty i sprowadza się do napisania tylko kilku linii kodu. Poniższy kod przedstawia inicjalizację nowego obiektu klasy Meter oraz jego zapis do bazy danych.

```
let newMeter = Meter()
newMeter.name = meterName!
let realm = try! Realm()
try! realm.write {
    realm.add(newMeter)
}
```

Listing 1 Stworzenie nowego obiektu klasy Meter oraz dodanie go do bazy danych

W celu odczytania obiektów z bazy danych wykonujemy równie prostą operację. Poniższy kod przedstawia prosty odczyt wszystkich obiektów klasy Meter z bazy danych.

```
private func loadMeters() {
    let realm = try! Realm()
    meters = Array(realm.objects(Meter.self))
    tableView.reloadData()
}
```

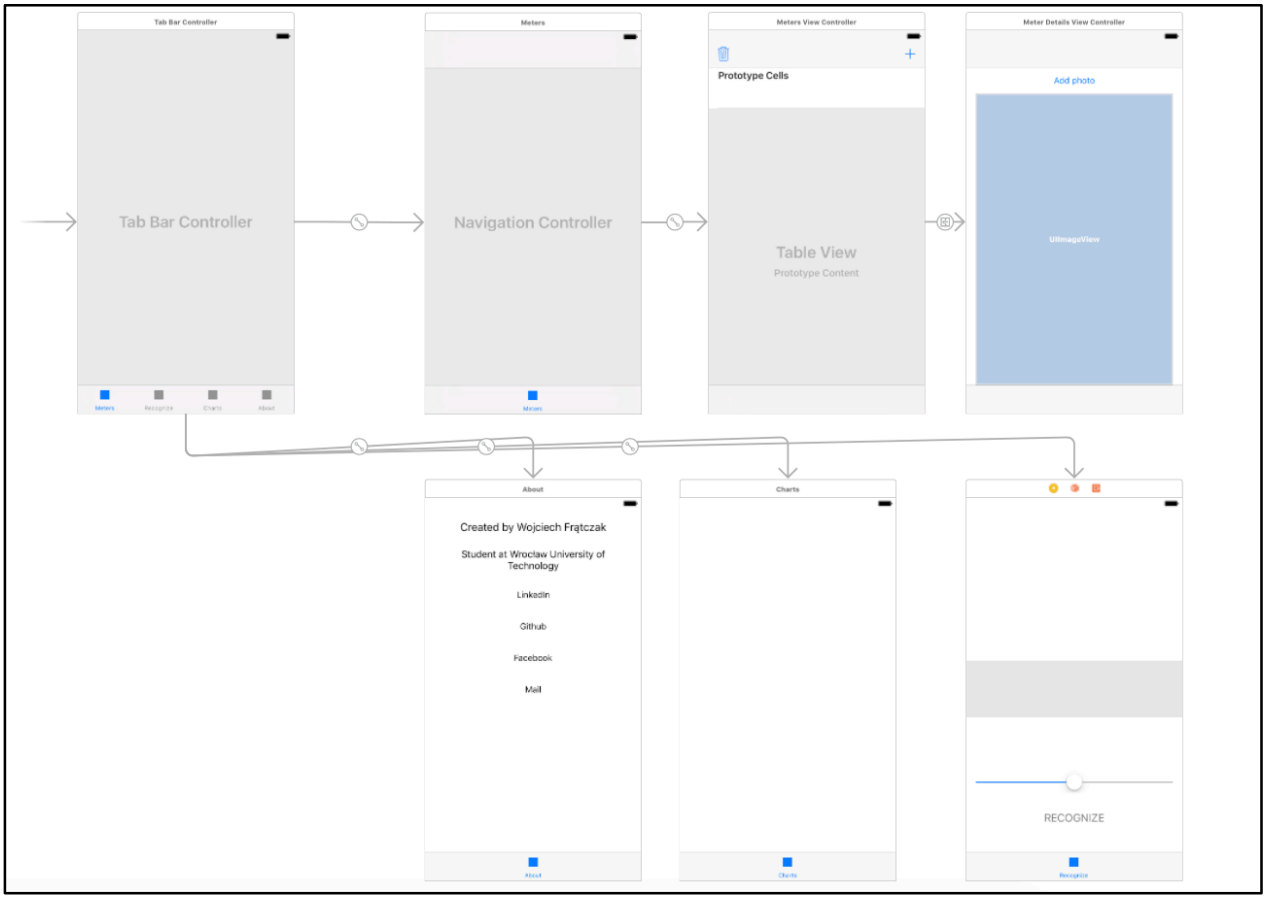
Listing 2 Wczytanie z bazy danych wszystkich obiektów klasy Meter

W rezultacie otrzymujemy tablicę obiektów w wyniku wywołania zaledwie jednej linijki kodu.

4.2. Implementacja interfejsu użytkownika

Do implementacji interfejsu użytkownika zostało wykorzystane narzędzie Interface Builder dostępne w ramach środowiska programistycznego Xcode. Jest to edytor, który umożliwia łatwe projektowanie interfejsu użytkownika bez wymogu pisania kodu. Technologia Cocoa Touch jest zbudowana na podstawie wzorca projektowego Model View Controller, zatem bardzo łatwe jest projektowanie warstwy interfejsu użytkownika nie ingerując w warstwy kontrolerów i modeli. Po zdefiniowaniu przynależności zaprojektowanego widoku z odpowiadającą klasą, system iOS dynamicznie tworzy połączenie między interfejsem użytkownika, a napisanym kodem. Przy pomocy plików

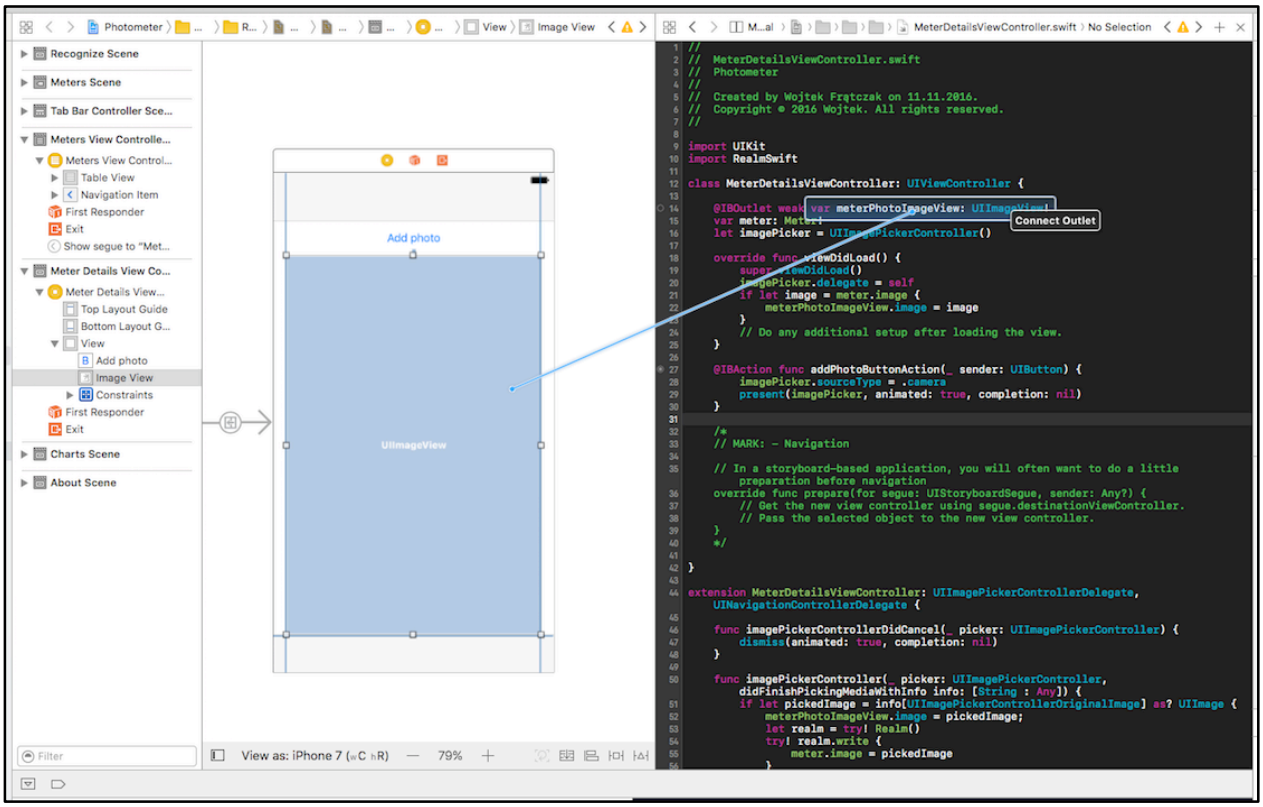
o rozszerzeniu storyboard możemy zaprojektować poszczególne widoki oraz połączenia między nimi. W ten sposób otrzymujemy podgląd przepływu aplikacji między poszczególnymi widokami. Na poniższym rysunku został przedstawiony podgląd pliku Main.storyboard ukazujący wszystkie widoki zaimplementowane w aplikacji razem z połączeniami między nimi.



Rysunek 3 Struktura widoków w pliku Main.storyboard

Przy budowaniu konkretnego widoku wykorzystujemy narzędzie Assistant, które pozwala nam na sprawne połączenie elementów interfejsu takich jak UILabel, UIButton i inne z kodem napisanym dla tego widoku. Poniżej przestawiam przykładowy sposób połączenia elementu z odpowiadającym polem obiektu klasy UIViewController. Analogiczną czynność wykonujemy dla połączenia akcji elementów odpowiadających

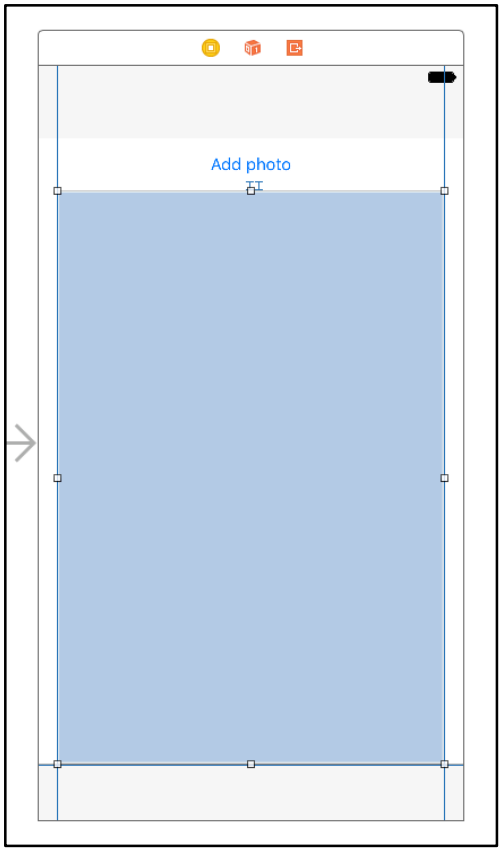
za interakcję z użytkownikiem z kodem wykonywanym podczas danej interakcji, np. dla obiektów klasy UIButton.



Rysunek 4 Budowanie widoków przy pomocy narzędzia Interface Builder

Z lewej strony obrazu widzimy strukturę zbudowanych widoków z podświetloną nazwą widoku, którym się obecnie zajmujemy oraz sam zbudowany widok. Na prawo od niego mamy dostęp do kodu napisanego lub częściowo wygenerowanego przez środowisko Xcode. Przy pomocy prawego przycisku myszy możemy sprawnie i szybko połączyć elementy interfejsu z kodem. Na rysunku zaprezentowane jest to przy pomocy niebieskiej linii.

Poszczególne elementy interfejsu umieszczamy wewnątrz wybranego widoku. Przy pomocy narzędzia Constraints możemy zdefiniować rozmiary elementu lub poszczególne odległości od innych elementów i krawędzi widoków. Na poniższym rysunku widzimy element z określonym rozmiarem i wyśrodkowaniem względem głównego widoku.

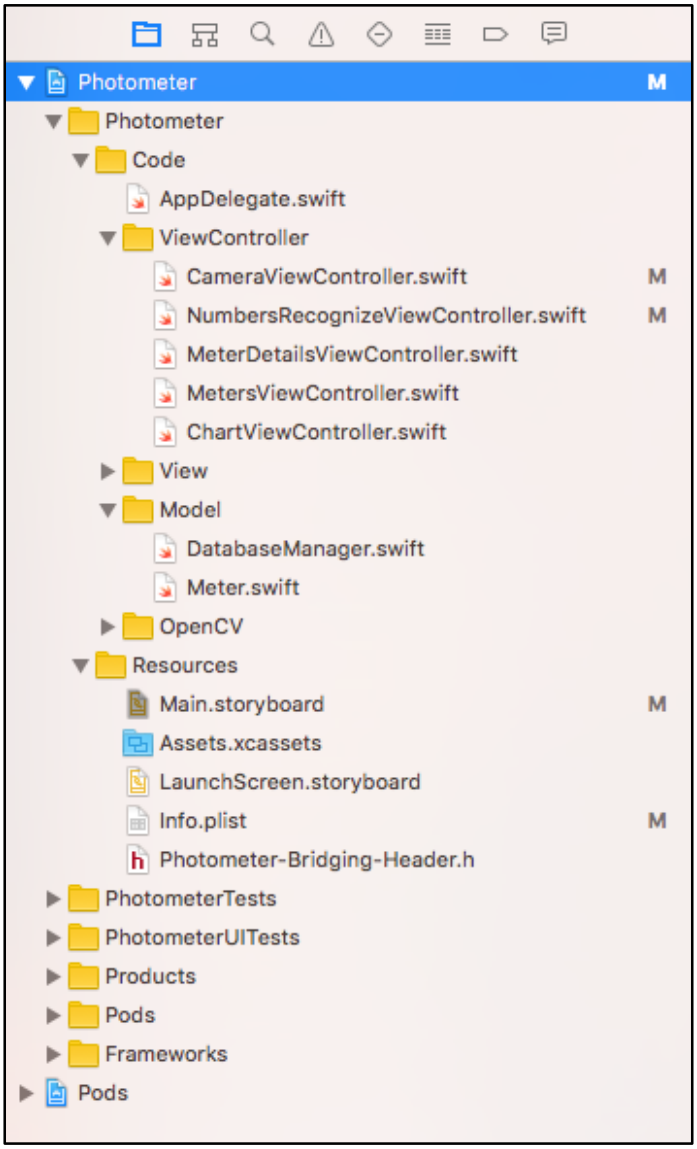


Rysunek 5 Pojedynczy widok z zaimplementowanymi elementami

4.3. Implementacja aplikacji

4.3.1. Struktura plików

Wszystkie pliki projektu zostały podzielone na odpowiednie grupy. W środowisku Xcode pracujemy w przestrzeni roboczej, która dzieli się na dwa projekty. Pierwszy to projekt aplikacji stworzony przez nas, drugi to projekt nazwany Pods zawierający biblioteki zewnętrzne zintegrowane z naszym projektem przez narzędzie CocoaPod. Dwie główne grupy projektu naszej aplikacji to Code oraz Resources. Grupa Code zawiera wszystkie pliki kodu źródłowego podzielone zgodnie z zasadami architektury Model View Controller. Grupa Resources zawiera wszystkie inne pliki, takie jak zasoby graficzne, pliki interfejsu użytkownika, czy wspomniany wcześniej plik bridging header.



Rysunek 6 Struktura projektu widoczna w środowisku Xcode

4.3.2. Protokoły

Każda klasa w języku Swift, posiada możliwość rozszerzania swojej funkcjonalności poprzez implementację metod zdefiniowanych w protokołach. Protokół definiuje zestaw metod, pól lub innych wymagań, które pasują do konkretnego zadania lub funkcjonalności. Protokół może być zaadoptowany przez klasę by zapewnić implementację tych wymagań. Mówimy wtedy, że klasa implementuje dany protokół. W napisanej przeze mnie aplikacji, napisane przeze mnie klasy kilkukrotnie implementowały różne protokoły.

Jest to powszechna praktyka w tworzeniu i rozwijaniu aplikacji w języku Swift oraz Objective-C.

```
public protocol UIImagePickerControllerDelegate : NSObjectProtocol {


    @available(iOS 2.0, *)
    optional public func imagePickerController(_ picker: UIImagePickerController,
        didFinishPickingMediaWithInfo info: [String : Any])

    @available(iOS 2.0, *)
    optional public func imagePickerControllerDidCancel(_ picker: UIImagePickerController)
}
```

Listing 3 Porotokół UIImagePickerControllerDelegate

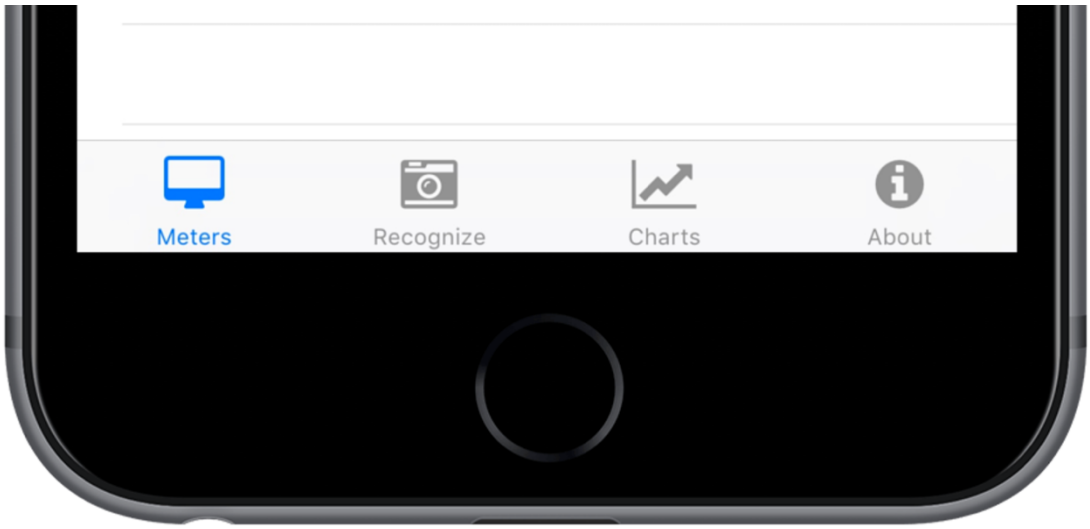
```
extension MeterDetailsViewController: UIImagePickerControllerDelegate,
    UINavigationControllerDelegate {

    func imagePickerControllerDidCancel(_ picker: UIImagePickerController) {
        dismiss(animated: true, completion: nil)
    }

    func imagePickerController(_ picker: UIImagePickerController,
        didFinishPickingMediaWithInfo info: [String : Any]) {
        if let pickedImage = info[UIImagePickerControllerOriginalImage] as? UIImage {
            meterPhotoImageView.image = pickedImage;
            let realm = try! Realm()
            try! realm.write {
                meter.image = pickedImage
            }
        }
        dismiss(animated: true, completion: nil)
    }

}
```

Listing 4 Przykład implementacji protokołu przez klasę MeterDetailsController

4.3.3. Warstwa kontrolerów widoków

Do wcześniej zaprojektowanych widoków przy pomocy narzędzia Interface Builder stworzyłem odpowiadające im klasy kontrolerów widoków dziedziczące po klasie UIViewController dostarczonej przez firmę Apple w bibliotece UIKit. Głównym kontrolerem zarządzającym nawigacją aplikacji jest obiekt klasy UITabBarController. Klasa ta implementuje specjalny

kontroler widoku, który posiada referencję do wszystkich kontrolerów widoków wchodzących w skład aplikacji. Interfejs omawianego kontrolera posiada ikonki odpowiadające każdemu kontrolerowi. Pojawia się on na dole interfejsu aplikacji. Po naciśnięciu odpowiedniej ikonki kontroler przenosi nas do odpowiedniego widoku.



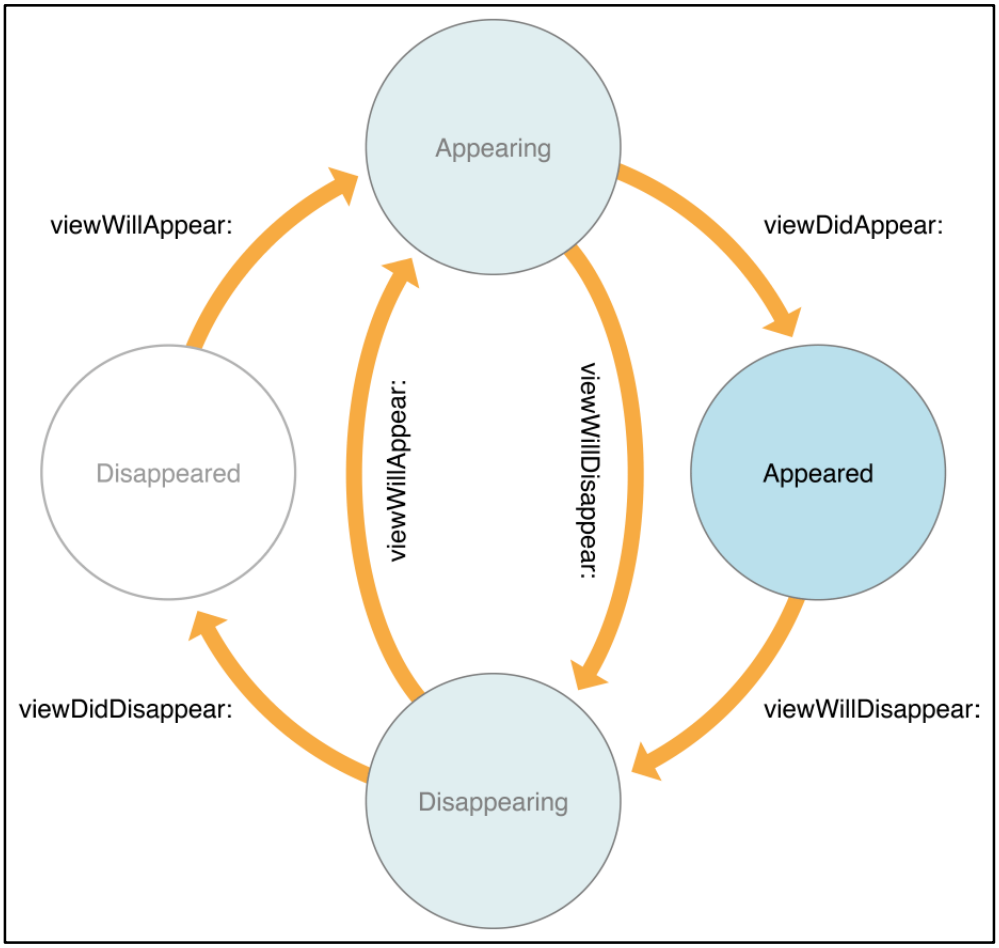
Rysunek 7 Interfejs umożliwiający nawigację w aplikacji

Każdy z zaimplementowanych kontrolerów widoków dzięki dziedziczeniu po klasie UIViewController posiada zaimplementowany stały zestaw metod odpowiadających za cykl życia widoku. Ułatwiają one implementacje konfiguracji, lub odpowiednich zmian na widoku w momentach załadowania, pojawienia się lub zniknięcia widoku. Do najczęściej używanych metod należą:

- viewDidLoad() – wywoływana po załadowaniu przez kontroler widoku wszystkich jego elementów
- viewDidAppear() / viewWillAppear() – wywoływane przed/po pojawieniu się widoku
- viewDidDisappear() / viewWillDisappear() – wywowływane przed/po zniknięciu widoku

```swift
override func viewDidLoad() {
    super.viewDidLoad()
    configureModel()
    configureChart()
}

override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
    updateChart()
}
```

Listing 5 Przykładowa implementacja metod odpowiadających za cykl życia kontrolera widoku



Rysunek 8 Przejścia między stanami kontrolera widoku[8]

W aplikacji zostało zaimplementowanych kilka kontrolerów widoków. Najważniejsze z nich to:

- CameraViewController
- MetersViewController
- MeterDetailsViewController
- ChartViewController
- AboutViewController.

Każdy z wyżej wymieniony kontrolerów widoków posiadają odpowiedzialność zarządzania przydzielonymi im widokami oraz funkcjonalnościami opisanymi wcześniej.

4.3.4. Warstwa modelowa

Warstwa modelowa jest integralną częścią architektury Model View Controller. Klasy modelowe wydzielają dane oraz definiują logikę przetwarzającą te dane. W napisanej przeze mnie aplikacji zostało stworzonych kilka klas modelowych. Są to klasy:

- Meter – reprezentuje rzeczywisty obiekt licznika. Posiada pola reprezentujące kolejno nazwę, datę stworzenia obiektu, obrazek dodany dla licznika oraz tablicę wartości odczytanych przez aplikację. Klasa posiada również zaimplementowaną logikę umożliwiającą zapisywanie i odczytywanie obrazu dodanego dla licznika.
- MeterValue – klasa reprezentująca wartość odczytaną z licznika. Posiada pola reprezentujące datę stworzenia obiektu oraz samą wartość stanu licznika.

```swift
import Foundation
import RealmSwift
import Realm

class Meter: Object {
    dynamic var name = ""
    dynamic var createdAt = NSDate()
    var values: List<MeterValue> = List()

    var image: UIImage? {
        get {
            let nsDocumentDirectory = FileManager.SearchPathDirectory.documentDirectory
            let nsUserDomainMask    = FileManager.SearchPathDomainMask.userDomainMask
            let paths               = NSSearchPathForDirectoriesInDomains(nsDocumentDirectory, nsUserDomainMask, true)
            if let dirPath        = paths.first
            {
                let imageURL = URL(fileURLWithPath: dirPath).appendingPathComponent(name)
                return UIImage(contentsOfFile: imageURL.path)
            }
            return nil
        }
        set {
            let writePath = FileManager.documentsDirectory().appendingPathComponent(name)
            if let newImage = newValue, let data = UIImageJPEGRepresentation(newImage, 1.0) {
                try? data.write(to: writePath)
            }
        }
    }

    override static func ignoredProperties() -> [String] {
        return ["image"]
    }
}
```

Listing 6 Implementacja klasy modelowej Meter

W celu integracji klasy modelowej z bazą danych Realm musi ona implementować protokół o nazwie Object. Protokół ten definiuję zestaw funkcjonalności umożliwiających zapisywanie i odczytywanie obiektów z bazy danych.

4.3.5. Przetwarzanie obrazu

W celu przetwarzania obrazów został zaimplementowanych szereg funkcjonalności. Umożliwiają one wykonać po kolei czynności dążące do ostatecznego rozpoznania licznika bądź odczytania jego stanu.

Pierwszą z nich jest możliwość wyświetlenia użytkownikowi obrazu z kamery, dzięki czemu może on skierować urządzenie na licznik w celu wykonania zdjęcia. W tym celu została wykorzystana biblioteka AVFoundation. Jest to biblioteka dostarczona dla programistów przez firmę Apple. Zapewnia ona istotne funkcjonalności umożliwiające pracę z urządzeniami

audiowizualnymi urządzenia w czasie rzeczywistym. Dzięki niemu możemy łatwo integrować naszą aplikację z mikrofonem, kamerą przednią lub tylną.

W celu wyświetlenia obrazu z kamery należy stworzyć prosty obiekt klasy UIView i umieścić go na docelowym widoku. Następnie tworzymy obiekt klasy AVCaptureSession oraz konfigurujemy go wedle uznania, w przypadku naszej aplikacji w celu przechwycenia obrazu z kamery tylnej. Ostatecznie tworzymy warstwę podglądu obrazu kamery i dodajemy go do wcześniej stworzonego widoku. Dodatkowo została zaimplementowana możliwość przybliżania oraz oddalania obrazu z kamery. W ten sposób mamy zaimplementowaną podstawową możliwość podglądu obrazu z kamery wewnątrz aplikacji.

```swift
// MARK: AVFoundation

func beginSession() {
    self.stillImageOutput = AVCaptureStillImageOutput()
    if UIDevice.current.userInterfaceIdiom == .phone && max(UIScreen.main.bounds.size.width,
        UIScreen.main.bounds.size.height) < 568.0 {
        self.captureSession.sessionPreset = AVCaptureSessionPresetPhoto
    } else {
        self.captureSession.sessionPreset = AVCaptureSessionPreset1280x720
    }

    self.captureSession.addOutput(self.stillImageOutput)

    DispatchQueue.global().async {
        do{
            self.captureSession.addInput(try AVCaptureDeviceInput(device: self.device))
        } catch {
            print("AVCaptureDeviceInput Error")
        }

        let previewLayer = AVCaptureVideoPreviewLayer(session: self.captureSession)
        previewLayer?.frame.size = self.cameraView.frame.size
        previewLayer?.frame.origin = CGPoint.zero
        previewLayer?.videoGravity = AVLayerVideoGravityResizeAspectFill

        do {
            try self.device?.lockForConfiguration()

            self.device?.focusPointOfInterest = CGPoint(x: 0.5, y: 0.5)
            self.device?.focusMode = .continuousAutoFocus

            self.device?.unlockForConfiguration()

        } catch {
            print("captureDevice?.lockForConfiguration() denied")
        }

        //Set initial Zoom scale

        do {
            let device = AVCaptureDevice.defaultDevice(withMediaType: AVMediaTypeVideo)
            try device?.lockForConfiguration()

            let zoomScale:CGFloat = 2.5

            if zoomScale <= (device?.activeFormat.videoMaxZoomFactor)! {
                device?.videoZoomFactor = zoomScale
            }

            device?.unlockForConfiguration()

        } catch {
            print("captureDevice?.lockForConfiguration() denied")
        }

        DispatchQueue.main.async(execute: {
            self.cameraView.layer.addSublayer(previewLayer!)
            self.captureSession.startRunning()
        })
    }
}
```

Listing 7 Konfiguracja oraz uruchomienie sesji AVCaptureSession.

Następnym krokiem jest przechwycenie obrazu z kamery po naciśnięciu przycisku. W tym celu należy wywołać metodę captureStillImageAsynchronously(from:completionHandler:) na obiekcie klasy AVCaptureStillImageOutput, który został wcześniej stworzony w celu konfiguracji sesji. Metoda w swoim wywołaniu zwrotnym zwraca nam obiekt reprezentujący dane przechwyconego zdjęcia. Następnie przekazujemy go dalej w celu przetworzenia.

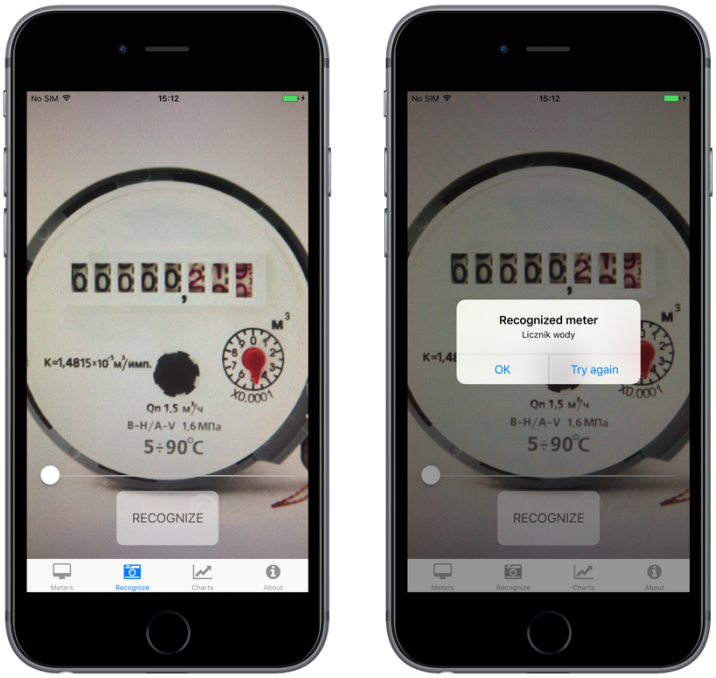
```
@IBAction func takePhotoButtonPressed(_ sender: UIButton) {
    self.stillImageOutput.captureStillImageAsynchronously(from: self.stillImageOutput.connection
        (withMediaType: AVMediaTypeVideo)) { (buffer, error) -> Void in
        guard let buffer = buffer, let imageData = AVCaptureStillImageOutput.
            jpegStillImageNSDataRepresentation(buffer), let image = UIImage(data: imageData) else {
            return
        }
        switch self.viewType {
        case .meter:
            self.recognizeMeter(image: image)
        case .values:
            self.recognizeValues(image: image)
        }
    }
}
```

Listing 8 Metoda wykonująca przechwytywanie obrazu z kamery

Jeśli mamy na celu przetworzenie obrazu w celu rozpoznania licznika, wywołujemy metodę recognizeMeter, która za argument przyjmuje obiekt klasy UIImage. Pobiera ona wszystkie dostępne obiekty klasy Meter z bazy danych oraz wykonuje porównanie ich obrazów wzorcowych z przechwyconym obrazem. Licznik, którego obraz wzorcowy uzyska najlepszy współczynnik porównania zostaje uznany za rozpoznany. Następnie jego nazwa zostaje wyświetlona w powiadomieniu.

```
private func recognizeMeter(image: UIImage) {
    let realm = try! Realm()
    let meters = Array(realm.objects(Meter.self))
    var recognizedIndex = 0
    var max = 0.0
    for (index, meter) in meters.enumerated() {
        let result = OpenCV.compare(image, with: meter.image)
        print("Reult for meter name: \(meter.name) \(result)")
        if (result?.first?.doubleValue)! > max {
            recognizedIndex = index
            max = (result?.first?.doubleValue)!
        }
    }
    recognizedMeter = meters[recognizedIndex]
    DispatchQueue.main.async {
        self.show(meterName: meters[recognizedIndex].name)
    }
}
```

Listing 9 Metoda obsługująca rozpoznanie licznika z przechwyconego obrazu



Rysunek 9 Interfejs graficzny dla rozpoznawania licznika

Porównanie przechwyconego obrazu z obrazami wzorcowymi jest wykonywane na zewnątrz metody. Stworzona przeze mnie klasa OpenCV w języku Objective-C umożliwia wykorzystanie metod biblioteki OpenCV, która jest napisana w języku C++. Dzięki automatycznie wygenerowanemu plikowi bridging header zaimplementowane porównanie obrazów można wykorzystać wewnątrz klas napisanych w języku Swift.

```
#import <Foundation/Foundation.h>
#import <UIKit/UIKit.h>

@interface OpenCV : NSObject

+ (NSString*)versionString;
+ (UIImage*)makeGrayscale:(UIImage*)rawImage;
+ (NSArray<NSNumber*>*) compareImage:(UIImage *)image with:(UIImage *)templateImage;

@end
```

Listing 10 Interfejs klasy OpenCV

Klasa OpenCV posiada zaimplementowaną metodę porównującą obrazy, która przyjmuje jako argumenty dwa obiekty klasy UIImage oraz zwraca tablicę czterech obiektów klasy NSNumber reprezentujących współczynniki porównania obrazów wykorzystując cztery różne metody porównania histogramów.

W celu porównania histogramów została wykorzystana metoda compareHist dostarczona przez bibliotekę OpenCV.

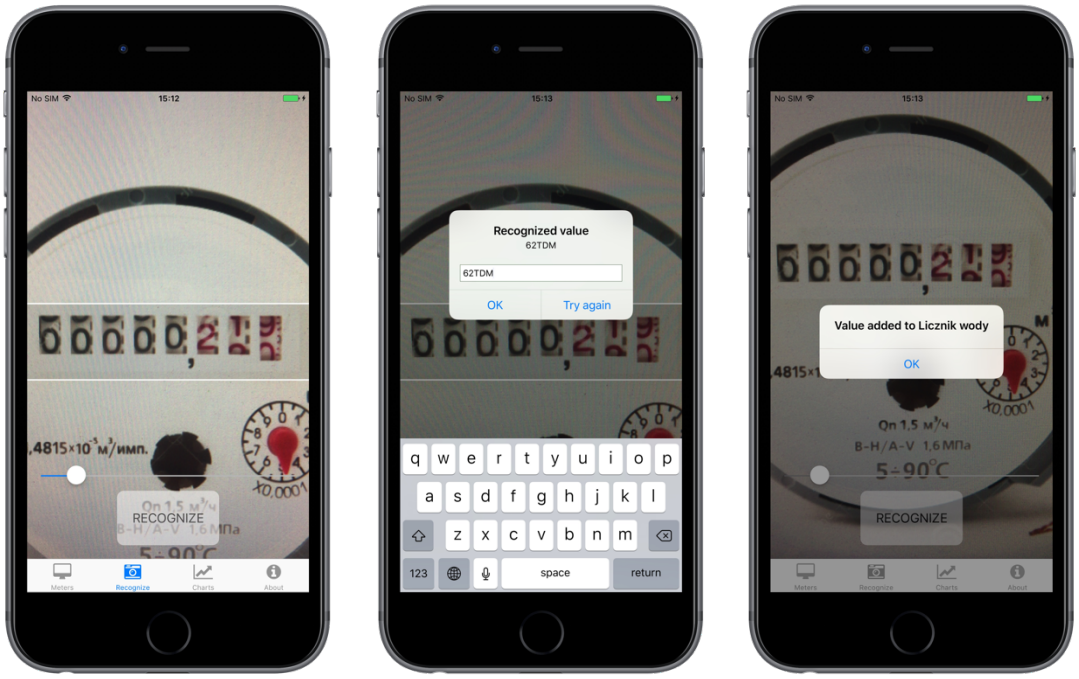
```
+ (NSArray<NSNumber*>*) compareImage:(UIImage *)image with:(UIImage *)templateImage
{
    Mat src_base, hsv_base;
    Mat src_test1, hsv_test1;

    UIImageToMat(image, src_base);
    UIImageToMat(templateImage, src_test1);

    /// Convert to HSV
    cvtColor( src_base, hsv_base, COLOR_BGR2HSV );
    cvtColor( src_test1, hsv_test1, COLOR_BGR2HSV );

    /// Using 50 bins for hue and 60 for saturation
    int h_bins = 50; int s_bins = 60;
    int histSize[] = { h_bins, s_bins };

    // hue varies from 0 to 179, saturation from 0 to 255
    float h_ranges[] = { 0, 180 };
    float s_ranges[] = { 0, 256 };

    const float* ranges[] = { h_ranges, s_ranges };

    // Use the o-th and 1-st channels
    int channels[] = { 0, 1 };


    /// Histograms
    MatND hist_base;
    MatND hist_test1;

    /// Calculate the histograms for the HSV images
    calcHist( &hsv_base, 1, channels, Mat(), hist_base, 2, histSize, ranges, true, false );
    normalize( hist_base, hist_base, 0, 1, NORM_MINMAX, -1, Mat() );

    calcHist( &hsv_test1, 1, channels, Mat(), hist_test1, 2, histSize, ranges, true, false );
    normalize( hist_test1, hist_test1, 0, 1, NORM_MINMAX, -1, Mat() );
    NSMutableArray *array = [@[] mutableCopy];
    /// Apply the histogram comparison methods
    for( int i = 0; i < 4; i++ )
    {
        int compare_method = i;
        double base_test1 = compareHist( hist_base, hist_test1, compare_method );

        [array addObject:[NSNumber numberWithDouble:base_test1]];
    }
    return array;
}
```

Listing 11 Metoda wykonująca porównanie histogramów obrazów wykorzystująca bibliotekę OpenCV

W celu odczytania wartości licznika została wykorzystana wcześniej opisana biblioteka zewnętrzna SwiftOCR. Dostarcza ona odpowiednie metody umożliwiające odczytanie wartości liczbowych z obrazu. Przechwycony obraz zostaje przekazany metodze recognizeValues. Tam następuje inicjalizacja instancji klasy SwiftOCR, na której wywołujemy metodę rozpoznającą ciąg cyfr.

```swift
private func recognizeValues(image: UIImage) {
    let croppedImage = self.cropImage(image)

    let ocrInstance = SwiftOCR()
    ocrInstance.recognize(croppedImage) { recognizedString in
        DispatchQueue.main.async(execute: {
            self.show(meterValue: recognizedString)
        })
    }
}
```

Listing 12 Metoda obsługująca rozpoznanie stanu licznika z przechwyconego obrazu



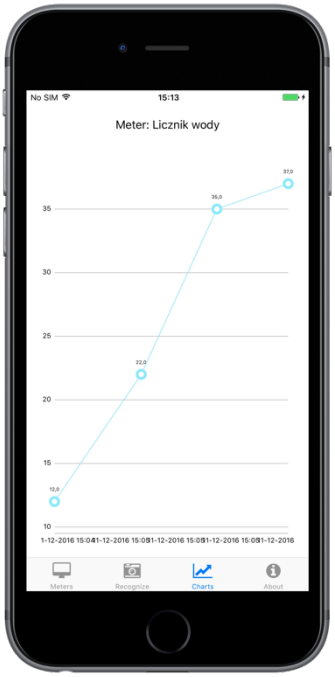
Rysunek 10 Interfejs graficzny dla rozpoznawania stanu licznika

4.3.6. Wizualizacja wyników

Jedną z ważniejszych funkcjonalności aplikacji jest wizualizacja przechowywanych wyników. Została ona zaimplementowana w kontrolerze widoku ChartViewController. Jest to prosta funkcjonalność wyświetlająca wszystkie dostępne odczytane stany licznika na przestrzeni czasu na wykresie. Do implementacji wykresów została wykorzystana zewnętrzna biblioteka Charts[9]. Integracja i jej wykorzystanie jest bardzo proste.

```
private func configureChart() {
    guard let meter = currentMeter else {
        return
    }
    titleLabel.text = "Meter: " + meter.name
    setupLeftAxis()
    setupXAxis()
    setupRightAxis()
    setupChartAppearace()
}

private func updateChart() {
    guard let meter = currentMeter else {
        return
    }
    var chartDataEntries: [ChartDataEntry] = []
    for value in meter.values {
        let xValue = value.createdAt.timeIntervalSince1970
        let entry = ChartDataEntry(x: xValue, y: value.value)
        chartDataEntries.append(entry)
    }
    let dataSet = LineChartDataSet(values: chartDataEntries, label: "Meter values")
    let chartData = LineChartData(dataSet: dataSet)
    chart.data = chartData
    chart.notifyDataSetChanged()
}
```

Listing 13 Implementacja i konfiguracja wykresu



Rysunek 11 Wizualizacja wyników na urządzeniu.

4.3.7. Testy

Testowanie jest integralną częścią tworzenia i rozwoju aplikacji. Pomaga uniknąć wielu błędów podczas tworzenia aplikacji oraz regresji w przyszłym jest rozwoju. W aplikacji zaimplementowałem kilka testów sprawdzających podstawowe funkcjonalności. Jest to możliwe dzięki bibliotece XCTest dostarczonej przez firmę Apple. XCTest dostarcza szereg funkcjonalności umożliwiających pisanie testów w łatwy sposób oraz integralność z środowiskiem Xcode. W aplikacji zostało zaimplementowanych kilka testów jednostkowych. Zwykłe testy jednostkowe sprawdzające pojedynczą funkcjonalność aplikacji oraz testy interfejsu użytkownika, które z łatwością można wygenerować dzięki środowisku Xcode. Posiadają one standardową strukturę testów z odpowiednio wywoływaną asercją w ciele funkcji testowej.

```
import XCTest
import RealmSwift
@testable import Photometer

class PhotometerTests: XCTestCase {

    var realm: Realm!
    var meters: [Meter] = []

    override func setUp() {
        super.setUp()
        realm = try! Realm()
        meters = Array(realm.objects(Meter.self))
    }

    override func tearDown() {
        // Put teardown code here. This method is called after the invocation of each test method in the class.
        super.tearDown()
    }

    func testMeterImageRecognize() {
        var recognizedIndex = 999
        var max = 0.0
        if let sourceImageFromFirstAvailableMeter = meters.first?.image {
            for (index, meter) in meters.enumerated() {
                let result = OpenCV.compare(sourceImageFromFirstAvailableMeter, with: meter.image)
                print("Reult for meter name: \(meter.name) \(result)")
                if (result?.first?.doubleValue)! > max {
                    recognizedIndex = index
                    max = (result?.first?.doubleValue)!
                }
            }
            XCTAssert(recognizedIndex == 0)
        } else {
            print("No meters available!")
        }
    }

}
```

Listing 14 Klasa testująca rozpoznawanie licznika

5. Wnioski

Napisana przeze mnie aplikacja z powodzeniem jest w stanie spełnić funkcje menadżera do zarządzania stanami liczników posiadanych przez użytkownika. Z powodzeniem została zaimplementowana funkcja rozpoznawania liczników. Otrzymujemy dobre wyniki przy spełnieniu odpowiednich warunków, między innymi:

- Identyczne lub podobne warunki oświetlenia
- Podobny profil licznika przy robieniu zdjęcia (w porównaniu do zdjęcia wzorcowego)

Zakładając umiejscowienie liczników w pomieszczeniach przy stałym, sztucznym świetle, nie powinniśmy napotkać większych problemów przy rozpoznawaniu liczników w praktyce.

Rozpoznawanie stanu licznika w wielu przypadkach kończy się pomyślnie, jednak aplikacja napotyka więcej problemów. Głównymi czynnikami powodującymi zakłócenia przy odczytywaniu stanu mechanicznego licznika są:

- Nieregularność pokazanych przez licznik cyfr
- Odseparowanie cyfr na oddzielnych prostokątach
- Różnice między innymi licznikami

W ramach rozwoju projektu warto stworzyć aplikacje na inne popularne systemy – Windows Phone oraz Android.

Bibliografia:

[1] The Swift Programming Language: http://tinyurl.com/swiftprogramminglanguage

[2] Vaish G., „High Performance iOS Apps" 2016r.

[3] OpenCV Image Processing: http://docs.opencv.org/2.4/modules/imgproc/doc/histograms.html

[4] Rafajłowicz E., Rafajłowicz W., Rusiecki A., „Algorytmy przetwarzania obrazów i wstęp do pracy z biblioteką OpenCV" 2009r.

[5] OpenCV Histogram Comparison: http://tinyurl.com/histogramscomparison

[6] Using Swift with Cocoa and Objective-C: http://tinyurl.com/swiftwithobjectivec

[7] Dokumentacja biblioteki SwiftOCR: https://github.com/garnele007/SwiftOCR

[8] Dokumentacja klasy UIViewController: http://tinyurl.com/uiviewcontroller

[9] Dokumentacja biblioteki Charts: https://github.com/danielgindi/Charts

[10] Model View Controller Apple documentation: http://tinyurl.com/applemvcdocumentation

Spis listingów:

Spis ilustracji:

Spis tabel: