

WYKŁAD 4

Programowanie Urządzeń Mobilnych

Kompleksowa analiza technologii, architektury i implementacji aplikacji mobilnych

 2026  Informatyka

Struktura prezentacji

1 Wprowadzenie

Rynek aplikacji mobilnych, statystyki i trendy rozwojowe

2 Typy aplikacji

Native, hybrydowe, PWA - analiza porównawcza

3 Środowiska IDE

Android Studio i Xcode - narzędzia deweloperskie

4 Architektura

MVC, MVP, MVVM i Clean Architecture

5 Optymalizacja

Wydajność, pamięć, sieć i bateria

6 Sensory i GPS

IMU, GNSS, mapy - specyfikacje techniczne

7 Obsługa kamery

Camera2 API, AVFoundation, analiza obrazu w czasie rzeczywistym

ROZDZIAŁ 1

Wprowadzenie do programowania mobilnego

Podstawy ekosystemu mobilnego,
statystyki rynkowe i trendy rozwojowe



Statystyki i trendy rynkowe



5.3B

Użytkowników smartfonów
Globalnie w 2026



255B

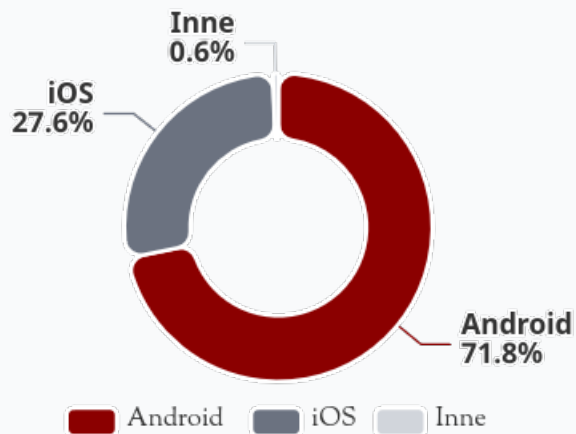
Pobrań aplikacji
Rocznie (2025)



\$542B

Przychody z aplikacji
Prognoza 2026

Dystrybucja platform



Liczba aplikacji w sklepach



Trend: Wzrost znaczenia aplikacji natywnych w sektorze fintech (bezpieczeństwo) i zdrowia (HIPAA compliance). Frameworki cross-platform zyskują 15% rynku rocznie.

Typy aplikacji mobilnych

Analiza porównawcza aplikacji natywnych, hybrydowych i PWA



2.1 APLIKACJE NATYWNE

Charakterystyka techniczna

Android

Kotlin

Java

Oficjalny język od 2017, wsparcie Google

iOS

Swift

Objective-C

Nowoczesny język Apple, bezpieczeństwo typów

Koszty deweloperskie





1.5-2x wyższe niż hybryda

Konieczność utrzymania dwóch osobnych baz kodu, zespoły specjalistów dla każdej platformy

Kluczowe zalety

- ✓ **Maksymalna wydajność**
Bezpośrednia kompilacja do kodu maszynowego, brak warstw pośrednich
- ✓ **Pełen dostęp do API**
Natywne komponenty UI, dostęp do wszystkich sensorów i funkcji sprzętowych
- ✓ **Płynność 60 FPS**
Natywne animacje, optymalizacja pod konkretny sprzęt

Zastosowania

-  Gry mobilne
-  Aplikacje AR/VR
-  Systemy płatności
-  Aplikacje medyczne

Frameworki i analiza techniczna



React Native

Facebook/Meta

Język: JavaScript/TypeScript

Używa natywnych komponentów UI, mostki do modułów natywnych



Flutter

Google

Język: Dart

Własny silnik renderujący, kompilacja AOT do kodu maszynowego



Xamarin

Microsoft

Język: C#/.NET

Udostępniony kod biznesowy, natywne interfejsy platformowe

Zalety hybrydy

1 Jedna baza kodu

Redukcja kosztów o 40-60%

2 Szybszy time-to-market

30-50% krótszy czas wdrożenia

3 Jednoczesne aktualizacje

Jedna wersja dla wszystkich platform

⚠ Ograniczenia

- **Wydajność:** 10-15% niższa w złożonych animacjach
- **Dostęp do sprzętu:** Wymaga mostków natywnych
- **Zależności:** Od frameworków i pluginów

★ Case study

Instagram, Facebook, Google Ads - aplikacje hybrydowe z setkami milionów użytkowników, udowadniające produkcyjną gotowość frameworków cross-platform.

Aplikacje webowe z funkcjonalnością natywną

Kluczowe technologie

- 1 Service Workers**
Skrypty działające w tle, obsługa offline, push notifications
- 2 Web App Manifest**
JSON z metadanymi aplikacji (ikona, nazwa, theme color)
- 3 Cache API**
Programistyczna kontrola nad cache'owaniem zasobów

Zalety PWA

- ✓ Natychmiastowe aktualizacje bez App Store
- ✓ Indeksowalność przez wyszukiwarki (SEO)
- ✓ Niskie zużycie pamięci (brak instalacji)
- ✓ Najniższe koszty deweloperskie

Ograniczenia krytyczne

Dostęp do sensorów

Brak Bluetooth/NFC na iOS, ograniczony dostęp do zaawansowanych sensorów

Dystrybucja

Brak możliwości publikacji w App Store (iOS), brak discoverability

Zużycie baterii

Wyższe niż natywne ze względu na działanie w przeglądarce

Przykłady produkcyjne



Twitter Lite



Pinterest



Starbucks

2.4 PORÓWNANIE TYPOW APLIKACJI

Parametry techniczne i biznesowe

• Kryterium	• Native	• Hybrid	• PWA
• Wydajność	• Excellent	• Very Good	• Good
• Koszty (średnia złożoność)	• \$150k-300k	• \$80k-150k	• \$50k-100k
• Dostęp do sprzętu	• 100%	• 85-90%	• 40-60%
• Czas wdrożenia	• 6-12 miesięcy	• 3-6 miesięcy	• 2-4 miesiące
• Zużycie baterii	• Niskie	• Średnie	• Wysokie
• Pamięć RAM	• Optymalne	• Średnie	• Zmienne
• Aktualizacje	• Via App Store	• Via App Store	• Natychmiastowe

Rekomendacja dla gier

Native

Rekomendacja dla biznesu

Hybrid

Rekomendacja dla contentu

PWA

ROZDZIAŁ 3

Środowiska programistyczne

Android Studio i Xcode - narzędzia, funkcje i możliwości
diagnostyczne



3.1 ANDROID STUDIO

Kompletne środowisko IDE dla Androida

Layout Editor

WYSIWYG edytor z **ConstraintLayout** - deklaratywne definiowanie interfejsu z podglądem na żywo.

Visual Design

Blueprint

Component Tree

Real-Time Profilers

CPU

Śledzenie wątków,
metody

Memory

Heap, allocations

Network

Żądania HTTP,
payload

Narzędzia dodatkowe

- ✓ APK Analyzer
- ✓ Layout Inspector

- ✓ Device Manager
- ✓ Database Inspector

Wspierane języki

Kotlin

Oficjalny

Java

Wsparcie

C++

NDK

Flutter

Plugin

Emulator

- Symulacja lokalizacji GPS
- Symulacja połączenia sieciowego
- Poziom baterii i ładowania
- Emulacja odcisku palca

Integracje

Firebase

Google Cloud

Gradle

3.2 XCODE

Narzędzia Apple dla deweloperów iOS

Interface Builder & Previews

Storyboards dla UIKit i **SwiftUI Previews** - podgląd interfejsu w czasie rzeczywistym podczas kodowania.

Live Preview: Interaktywny podgląd, zmiany widoczne natychmiast

Instruments Suite

Time Profiler

Śledzenie CPU, analiza stack traces

Allocations & Leaks

Wykrywanie wycieków pamięci

Energy Log

Optymalizacja zużycia baterii

Metal System Trace

Analiza GPU i renderowania

Memory Graph Debugger

Wizualizacja obiektów w pamięci, identyfikacja cykli retencji (retain cycles) w czasie rzeczywistym.

Widok grafu referencji między obiektami

Simulator

- Różne rozmiary ekranów iPhone/iPad
- Symulacja Face ID, Apple Pay
- Dark Mode, Dynamic Type
- Symulacja różnych iOS

Xcode Cloud

CI/CD wbudowane w IDE - automatyczne budowanie, testowanie i dystrybucja.

Metryki krytyczne i techniki debugowania

Kluczowe metryki wydajności

Czas uruchomienia

< 3s

Cold start - od kliknięcia ikony do interaktywności

Wydajność UI

60 FPS

16ms na klatkę - płynne animacje i przewijanie

Zużycie pamięci

Monitor

Heap, wykrywanie wycieków, memory pressure

Optymalizacja baterii

Energy Log (Xcode)

Śledzenie zużycia energii przez komponenty aplikacji

Battery Profiler (Android)

Analiza wake locks, background services

Cel

Minimalizacja zużycia baterii przy zachowaniu funkcjonalności

Narzędzia zewnętrzne i techniki



LeakCanary

Android - wycieki



Firebase

Performance Monitoring



LLDB

Debugger Xcode



Breakpoints

Conditional, symbolic

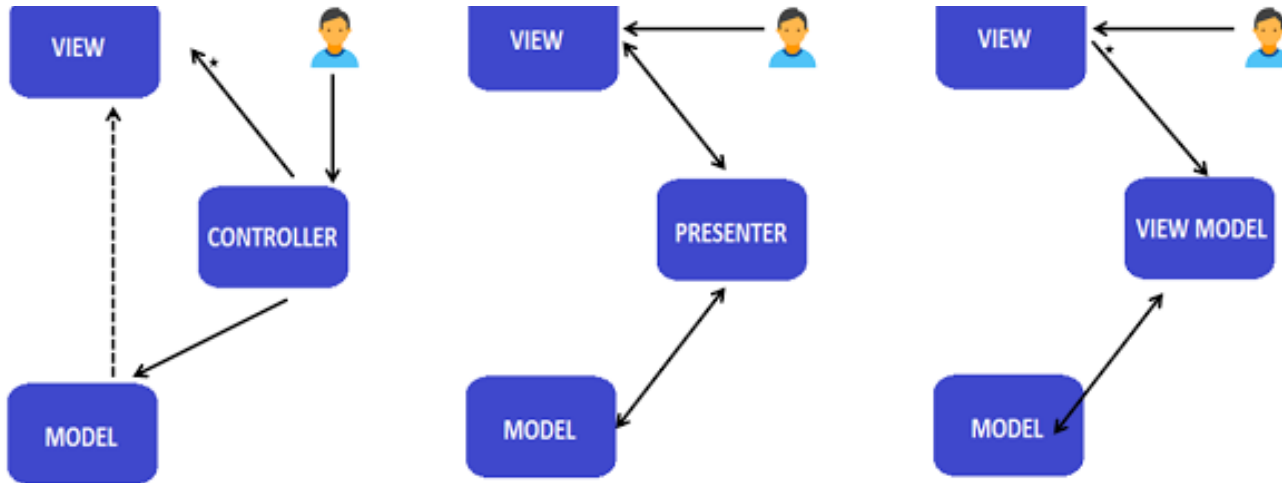
Architektura aplikacji mobilnych

Wzorce projektowe: MVC, MVP, MVVM i Clean
Architecture



4.1 WZORZEC MVC (MODEL-VIEW-CONTROLLER)

Klasyczna architektura trójwarstwowa



Komponenty MVC

- M Model**
Dane, logika biznesowa, komunikacja z bazą/API
- V View**
Interfejs użytkownika, komponenty UI, prezentacja
- C Controller**
Pośrednik między Model a View, obsługa zdarzeń

+ Zalety

- ✓ Prosta struktura
- ✓ Łatwa do zrozumienia
- ✓ Dobra dla małych projektów

- Wady w kontekście mobilnym

Problem z Androidem

Activities/Fragments pełnią rolę View i Controller

Ścisłe sprzężenie

View i Model bezpośrednio połączone

Trudności w testowaniu

Zależność od frameworków UI

Rekomendacja

Małe projekty prototypowe

4.2 WZORZEC MVP (MODEL-VIEW-PRESENTER)

Ewolucja MVC z jednym pośrednikiem

Architektura MVP

Model

Dane i logika biznesowa - identycznie jak w MVC

View

UI - przekazuje zdarzenia użytkownika do Presentera

Presenter

Jedyny pośrednik - logika prezentacji, komunikacja z Modelem

Kluczowa różnica od MVC

Brak bezpośredniej komunikacji między Model a View.

Wszystko przechodzi przez Presenter.

✓ Zalety MVP

- ✓ **Lepsza separacja**
View nie zna Modelu
- ✓ **Łatwiejsze testowanie**
Presenter testowalny niezależnie
- ✓ **Jeden Presenter = jeden View**
Jasna odpowiedzialność

⚠ Wady

- Więcej kodu boilerplate
- Presenter może stać się wąskim gardłem
- Wymaga więcej plików

Stosowany przez: Google (niektóre aplikacje Android)

4.3 WZORZEC MVVM (MODEL-VIEW-VIEWMODEL)

Najpopularniejszy wzorzec współczesny

Architektura MVVM

Model

Abstrakcja źródeł danych, encje biznesowe

View

Obserwuje ViewModel, przekazuje akcje użytkownika

ViewModel

Logika prezentacji, data binding, LiveData/StateFlow

Kluczowa zaleta

ViewModel nie ma referencji do View - całkowite rozłączenie, najlepsza testowalność.

★ Zalety MVVM

Data Binding

Automatyczna synchronizacja danych

Wielokrotne użycie

leden ViewModel dla wielu View

Najlepsza testowalność

Brak zależności od UI framework

</> Implementacje

🍏 **SwiftUI:** @StateObject, @ObservedObject

🤖 **Android:** Jetpack ViewModel, LiveData

Rekomendacja
Duże aplikacje enterprise

4.4 CLEAN ARCHITECTURE

Zaawansowana architektura warstwowa

Warstwy Clean Architecture

Presentation Layer

UI, ViewModels, Fragments/ViewControllers

Zewnętrzna

Domain Layer

Use cases, encje biznesowe - niezależna od frameworków

Wewnętrzna

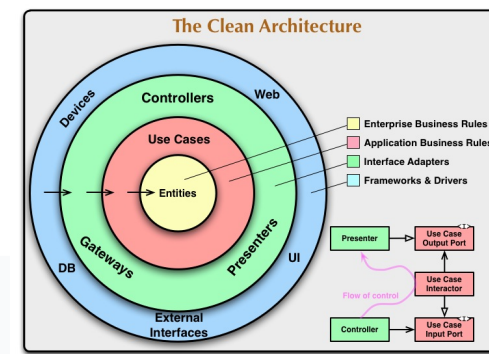
Data Layer

Repositories, źródła danych (API, baza lokalna)

Zewnętrzna

Zasada Dependency Inversion

Interfejsy definiowane w warstwach wewnętrznych, implementacje w zewnętrznych. Zależności skierowane do środka.



🏆 Zalety

- ✓ Maksymalna testowalność
- ✓ Łatwa wymiana komponentów
- ✓ Niezależność od frameworków
- ✓ Łatwa migracja API/bazy

Implementacja

Clean Architecture często implementowana w połączeniu z **MVVM** - ViewModel jako część Presentation Layer.

Autor 2012

Robert C. Martin (Uncle Bob)



4.5 PORÓWNANIE WZORCÓW ARCHITEKTONICZNYCH

Analiza kryteriów wyboru architektury

Kryterium	MVC	MVP	MVVM
Testowalność	Trudna	Dobra	Najlepsza
Separacja komponentów	Słaba	Dobra	B. dobra
Złożoność	Niska	Średnia	Śr.-wys.
Skalowalność	Słaba	Dobra	B. dobra
Learning curve	Łatwy	Średni	Średni
Boilerplate code	Minimalny	Średni	Średni

MVC

Małe projekty prototypowe, MVP (Minimum Viable Product)

Szybki start, ograniczona skalowalność

MVP

Średnie aplikacje z złożonym UI, legacy projects

Dobra separacja, więcej kodu

MVVM + Clean

Duże aplikacje enterprise, długoterminowe projekty

Maksymalna skalowalność i testowalność

ROZDZIAŁ 5

Optymalizacja wydajności

Techniki optymalizacji pamięci, CPU, sieci i baterii



Zarządzanie zasobami i wykrywanie wycieków

Wykrywanie wycieków pamięci

LeakCanary (Android)

Automatyczne wykrywanie wycieków Activity/Fragment

Memory Graph Debugger (iOS)

Wizualizacja cykli retencji między obiektami

Object Pooling

Ponowne używanie obiektów zamiast tworzenia nowych - redukcja garbage collection.

Przykład:

Pula obiektów dla cząstek w grach, buffery dla operacji I/O

Zarządzanie bitmapami

Downscaling

Skalowanie obrazów do rozmiaru wyświetlania przed załadowaniem

Bitmap Recycling (Android)

Ręczne zwalnianie pamięci bitmap przed GC

LRU Cache

Cache o stałym rozmiarze, usuwanie najstarszych elementów

Case study: ShopFast

Zużycie pamięci przed: **250 MB**

Zużycie pamięci po: **220 MB**

Redukcja: **30 MB (12%)**

Poprzez eliminację wycieków contextu i wewnętrznych klas

Płynność interfejsu - target 60 FPS



60

FPS

Płynne animacje



16

ms

Na klatkę



>16

ms

Frame drops

Optymalizacja layoutów

ConstraintLayout (Android)

Płaskie hierarchie zamiast głęboko zagnieżdżonych layoutów

Eliminacja overdraw

Unikanie wielokrotnego rysowania tych samych pikseli



RecyclerView (Android) / UITableView (iOS)

ViewHolder pattern dla efektywnego przewijania

Animacje

- **Hardware acceleration** - GPU dla animacji
- **MotionLayout** - deklaratywne animacje
- **Choreographer** - synchronizacja z VSync
- **Unikanie animacji** na głównym wątku

Narzędzia

-  GPU Overdraw (Android)
-  Core Animation (Xcode)

Efektywność energetyczna i komunikacja

Strategie sieciowe

Batching requestów

Grupowanie żądań HTTP - redukcja wake-up radia

Kompresja GZIP

Redukcja rozmiaru payload o 60-80%

Caching

OkHttp (Android), NSURLCache (iOS)

HTTP/2 i gRPC

Multiplexing, streaming, mniejszy overhead

Przykład: WhatsApp

Kompresja obrazów i wideo przed wysłaniem - aplikacja „szybka” i efektywna pod względem danych.

Optymalizacja baterii

WorkManager / BackgroundTasks

Zamiast ciągłych usług - taski w odpowiednim czasie

FusedLocationProvider

PRIORITY_BALANCED_POWER_ACCURACY dla GPS

Doze mode i App Standby

Dostosowanie aplikacji do trybów oszczędzania

Case study: ShopFast

Zużycie baterii (idle): **12%/h → 3.6%/h**

Redukcja: **70%**

Przejęcie na WorkManager z ograniczeniami

Narzędzia

 Energy Log

 Battery Profiler

Systemy sensorów mobilnych

IMU, akcelerometr, żyroskop, magnetometr -
specyfikacje i implementacja



6.1 INERTIAL MEASUREMENT UNIT (IMU)

Specyfikacje techniczne sensorów MEMS

Akcelerometr

Zakres pomiarowy:	$\pm 2g$ do $\pm 16g$
Częstotliwość:	100-1000 Hz
Dokładność (bias):	0.1-10 mg
Pomiar przyspieszenia liniowego w 3 osiach (X, Y, Z)	

Żyroskop

Zakres:	$\pm 250-2000^\circ/s$
Częstotliwość:	200-2000 Hz
Drift:	1-100$^\circ/h$
Pomiar prędkości kątowej (rotacji) w 3 osiach	

Magnetometr

Zakres:	$\pm 4800 \mu T$
Rozdzielczość:	0.1 μT
Pomiar pola magnetycznego Ziemi - orientacja względem północy	

Parametry systemowe

Zużycie energii:	10-100 mW
Stopnie swobody:	6 DoF
Technologia:	MEMS

Sensor Fusion

Łączenie danych z akcelerometru, żyroskopu i magnetometru dla poprawy dokładności orientacji.

Konwersja ADC w sensorach mobilnych

Proces konwersji ADC

1 Sygnał analogowy

Napięcie proporcjonalne do mierzonej wielkości fizycznej (temperatura, światło, dźwięk)

2 Próbkowanie

Pomiar amplitudy sygnału w regularnych odstępach czasu (sampling rate)

3 Kwantyzacja

Mapowanie wartości analogowej na dyskretną wartość cyfrową

Błąd kwantyzacji

Różnica między rzeczywistą wartością analogową a jej reprezentacją cyfrową. Zależy od rozdzielczości ADC.

Parametry ADC

Rozdzielczość

Liczba bitów (12-16 bit w smartfonach) - określa dokładność

Częstotliwość próbkowania

Hz - jak często sygnał jest próbkowany

Zakres wejściowy

Napięcie, które może być przetworzone

Przykłady w smartfonach



Mikrofon

Audio ADC: 44.1-48 kHz, 16-24 bit



Czujnik światła

Automatyczna jasność ekranu



Czujnik temperatury

Monitorowanie baterii

API systemowe i praktyczne zastosowania

Android - SensorManager

Rejestracja listenera

```
sensorManager.registerListener(  
listener,  
accelerometer,  
SensorManager.SENSOR_DELAY_GAME  
)
```

NORMAL 200ms - oszczędność baterii

GAME 50ms - gry, interakcja

FASTEST 0ms - maksymalna częstotliwość

Typy sensorów

- ✓ TYPE_ACCELEROMETER
- ✓ TYPE_MAGNETIC_FIELD
- ✓ TYPE_GYROSCOPE
- ✓ TYPE_ROTATION_VECTOR

iOS - CoreMotion

CMMotionManager

```
let motionManager = CMMotionManager()  
motionManager.accelerometerUpdateInterval = 0.05  
motionManager.startAccelerometerUpdates()
```

CMDeviceMotion

Przetworzone dane z sensor fusion - quaterniony, macierz rotacji

Różnice w implementacji

Android:

Surowe dane z sensorów, większa kontrola

iOS:

Bardziej przetworzone dane, sensor fusion wbudowany

Kalibracja:

Wymagana dla MEMS - kompensacja biasu i driftu

ROZDZIAŁ 7

GPS i systemy map mobilnych

GNSS, dokładność pozycjonowania, integracja map



Dokładność i parametry pozycjonowania

Systemy satelitarne

GPS (USA)

Najstarszy i najbardziej rozpowszechniony system

3-5m

GLONASS (Rosja)

Lepsza widoczność na wysokich szerokościach

3-7m

Galileo (UE)

HAS: <20cm (High Accuracy Service)

1m

BeiDou (Chiny)

Najlepszy w regionie Azji-Pacyfik

2.5-5m

Multi-GNSS w smartfonach

Współczesne smartfony używają wszystkich systemów jednocześnie dla lepszej dokładności i niezawodności.

- ✓ Szybszy fix (time-to-first-fix)
- ✓ Lepsza dokładność w miastach
- ✓ Większa niezawodność

Dual-frequency (L1+L5)

Eliminacja błędów jonosferycznych

Lane-level accuracy w miastach

Redukcja multipath

Mniej odbić od budynków

Zużycie baterii

Ciągłe śledzenie GPS: **10-15% baterii/godzinę**

Porównanie platform mapowych

MapKit (iOS)

Zalety

- ✓ Natywna integracja z CoreLocation
- ✓ Płynne animacje (MKAnnotation jako UIView)
- ✓ Tryb "Follow user location"
- ✓ Lepsza wydajność na urządzeniach Apple

Ograniczenia

- ✗ Tylko iOS
- ✗ Mniej funkcji niż Google Maps

Google Maps SDK

Zalety

- ✓ Cross-platform (Android/iOS)
- ✓ Zaawansowane funkcje (Street View, indoor)
- ✓ Traffic, Places API
- ✓ Spójny wygląd na wszystkich platformach



Wymagania

- ! API Key
- ! Płatność przy dużym ruchu

MKDirections

```
let request = MKDirections.Request()
request.source = sourceMapItem
request.destination = destMapItem
let directions = MKDirections(request: request)
```

Rekomendacje

-  **MapKit** dla natywnych aplikacji iOS
-  **Google Maps** dla cross-platform i zaawansowanych funkcji

ROZDZIAŁ 8

Obsługa kamery mobilnej

API kamery, parametry, przechwytywanie i analiza obrazu



Architektura i możliwości kamery w Androidzie

Camera2 API (niski poziom)

Pełna kontrola nad parametrami

- Czas ekspozycji (shutter speed)
- ISO (czułość sensora)
- Balans bieli (white balance)
- Tryby autofocus

Przechwytywanie RAW (DNG)

Nieskompresowane dane z sensora dla profesjonalnej obróbki

Manualne sterowanie

Dla aplikacji pro i zaawansowanych użytkowników

Tryby autofocus

- | | |
|----------------------|--------------------|
| ✓ CONTINUOUS_PICTURE | ✓ CONTINUOUS_VIDEO |
| ✓ MACRO | ✓ EDOF |

CameraX (wysoki poziom)

Jetpack Library

Abstrakcja nad Camera2, łatwiejsze użycie

UseCase Pattern

- **Preview** - podgląd na żywo
- **ImageCapture** - zdjęcia
- **ImageAnalysis** - analiza w czasie rzeczywistym

Automatyczna konfiguracja

Dostosowanie do możliwości urządzenia

Parametry techniczne

Rozdzielczość preview:	do 1080p
Rozdzielczość capture:	do 48MP+
Frame rate:	30-240 FPS
Analysis:	640x480

Przechwytywanie i przetwarzanie w czasie rzeczywistym

Architektura AVFoundation

AVCaptureSession

Koordynuje input (kamera, mikrofon) i output (zdjęcie, wideo, metadata)

AVCaptureDevice

Konfiguracja kamery: focus mode, exposure, white balance

AVCapturePhotoOutput

Zdjęcia: HEIF, JPEG, RAW, Live Photos

AVCaptureVideoDataOutput

Przetwarzanie w czasie rzeczywistym

Zaawansowane funkcje

- ✓ Portrait mode (depth data)
- ✓ Night mode
- ✓ ProRAW
- ✓ Cinematic mode



Przykład kodu Swift

```
// Konfiguracja sesji
let session = AVCaptureSession()
session.beginConfiguration()
// Dodanie input (kamera tylna)
guard let device = AVCaptureDevice.default(
    .back, for: .video)
else { return }
let input = try AVCaptureDeviceInput(device: device)
session.addInput(input)
// Dodanie output (zdjęcia)
let output = AVCapturePhotoOutput()
session.addOutput(output)
session.commitConfiguration()
```

Przetwarzanie na GPU

Metal framework dla zaawansowanych filtrów i przetwarzania obrazu w czasie rzeczywistym.

Integracje

-  **CoreImage** - filtry i efekty
-  **Vision** - analiza obrazu (ML)

8.3 ANALIZA OBRAZU I PRZETWARZANIE W CZASIE RZECZYWISTYM

ML Kit, Core ML i TensorFlow Lite

ML Kit

- ✓ Rozpoznawanie tekstu (OCR)
- ✓ Wykrywanie twarzy
- ✓ Kody kreskowe/QR
- ✓ Etykietowanie obrazów

Android & iOS

Core ML

- ✓ Wykonywanie modeli ONNX
- ✓ TensorFlow Lite
- ✓ Optymalizacja dla Apple Silicon
- ✓ Neural Engine (ANE)

iOS Only

TensorFlow Lite

- ✓ Modele zoptymalizowane
- ✓ Delegaty GPU (Metal, OpenCL)
- ✓ Kwantyzacja (INT8)
- ✓ Cross-platform

Android & iOS

Przetwarzanie w czasie rzeczywistym

Android

ImageAnalysis.UseCase - analiza każdej klatki

iOS

AVCaptureVideoDataOutputSampleBufferDelegate

⚠ Wyzwania i optymalizacje

Wyzwania:

Zużycie baterii, ograniczona moc obliczeniowa, opóźnienia

Optymalizacje:





Downsampling, GPU (Metal, RenderScript), batchowanie

Kluczowe wnioski i perspektywy rozwojowe

Kluczowe wnioski

- 1 Wybór typu aplikacji**
Zależy od wymagań wydajnościowych i budżetu
- 2 Architektura MVVM + Clean**
Zapewnia najlepszą skalowalność i testowalność
- 3 Optymalizacja jest krytyczna**
Wpływa bezpośrednio na retencję użytkowników
- 4 Sensory mobilne**
Oferują zaawansowane możliwości pomiarowe

Przyszłe kierunki rozwoju

-  **AR/VR**
Aplikacje wykorzystujące sensory IMU i kamery
-  **AI/ML na urządzeniu**
Core ML, TensorFlow Lite, Neural Engine
-  **Postęp w GNSS**
Dual-frequency, centymetrowa dokładność
-  **Ewolucja frameworków**
Flutter, React Native - coraz bliżej natywnemu

Narzędzia współczesnego dewelopera



Android Studio
Profiler, Emulator



Firebase
Performance Monitoring



Xcode
Instruments, Simulator



LeakCanary
Wykrywanie wycieków

Statystyka końcowa

60+
FPS target

6
DoF IMU

<3s
Czas startu

1m
GPS Galileo

Programowanie urządzeń mobilnych wymaga holistycznego podejścia łączącego znajomość natywnych API, architektury oprogramowania i optymalizacji wydajnościowej.