PROGRAMOWANIE APLIKACJI MOBILNYCH

WYKŁAD 2

Architektura aplikacji mobilnych

dr inż. Mateusz Pomianek

Katedra Informatyki i Automatyki, Politechnika Rzeszowska

Budynek D.103 // m.pomianek@prz.edu.pl // http://www.kia.prz.edu.pl/

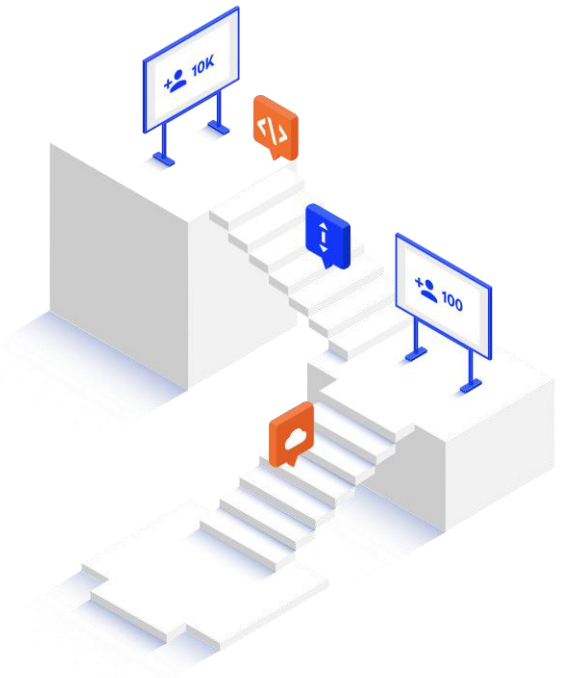
Agenda

1. Inżynieria oprogramowania
2. Wzorce projektowe
3. Architektura aplikacji
   a. MVC
   b. MVP
   c. MVVM
4. System Android
5. Komponenty aplikacji w systemie Android

Skalowalność aplikacji

Skalowalność - zdolność oprogramowania do sprawnego działania w warunkach stale rosnącej liczby użytkowników lub zwiększającej się objętości przetwarzanych danych. Jest to zdolność do takiej rozbudowy systemu, aby pracował z akceptowalną wydajnością.

Skalowalna aplikacja to taka, która działa dobrze zarówno gdy korzysta z niej kilku, jak i tysiące użytkowników przy niskim oraz wysokim natężeniu ruchu.



***Co jeszcze** świadczy o aplikacji dobrej jakości?*

Inżynieria oprogramowania

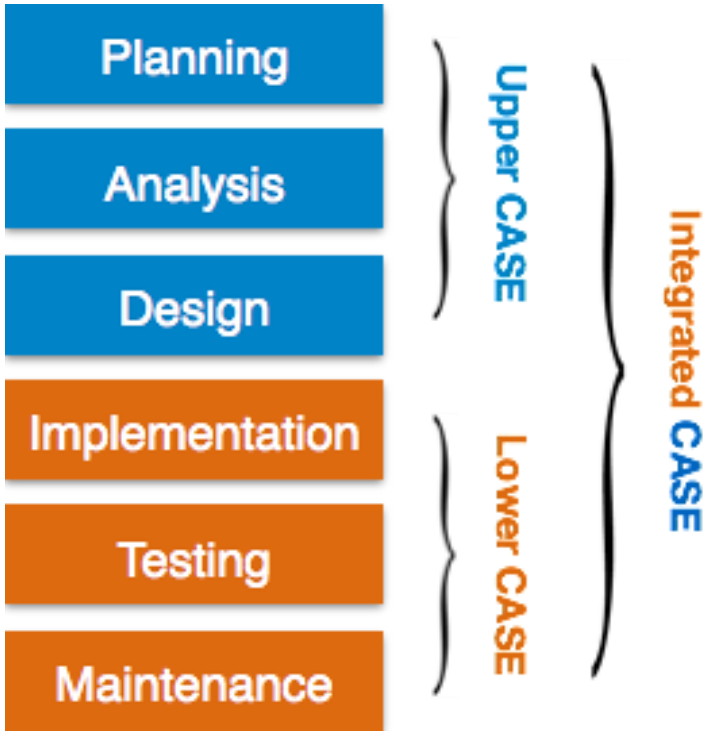
Inżynieria oprogramowania zajmuje się inżynierskim podejściem do tworzenia oprogramowania,

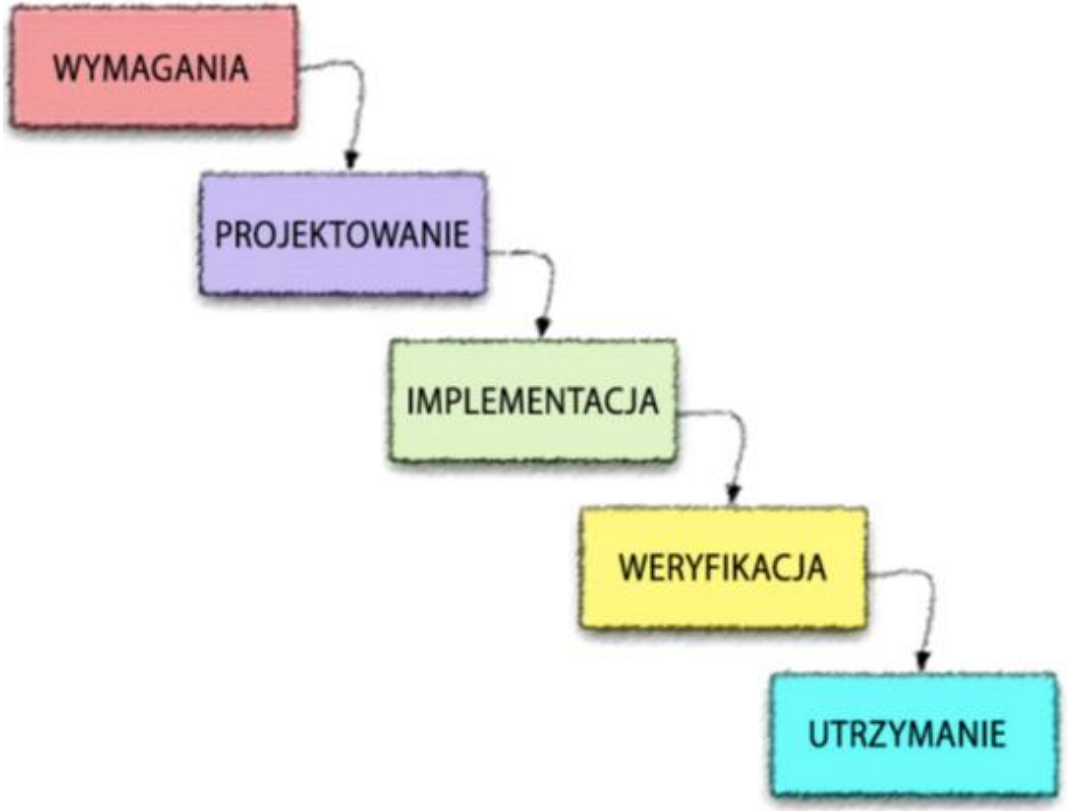
...

od analizy i określenia wymagań, przez projektowanie i wdrożenie, aż do ewolucji gotowego oprogramowania.

CASE (ang. Computer-Aided Software/Systems Engineering)

Komputerowe wspomaganie projektowania oprogramowania.





Refactoring

Refaktoryzacja to proces **poprawy istniejącego kodu**, którego celem jest zmiana oprogramowania w taki sposób, aby **nie zmieniał funkcji kodu**, ale **ulepszał jego wewnętrzną strukturę,** czynił je łatwiejszym do zrozumienia i tańszym w modyfikacji.

- Naprawa błędu nie jest refaktoryzacją.
- Dodanie nowej funkcji nie jest refaktoryzacją.

Kryzys oprogramowania

Rozwój **technik wytwarzania oprogramowania** nie nadąża za **rozwojem sprzętu komputerowego**.

Przyczyny kryzysu oprogramowania:

- duża złożoność systemów komputerowych,
- niepowtarzalność rozwiązań tworzonych na poszczególnych etapach projektu,
- pozorna łatwość tworzenia oprogramowania i programowania,
- trudność w ocenie stopnia zaawansowania prac:

  „jeżeli po miesiącu realizacji projektu informatycznego usłyszymy od programistów, że prace są zaawansowane w 90%, to można się spodziewać, że przedsięwzięcie potrwa jeszcze cały rok”,

Kryzys oprogramowania

Typowe źródła błędów:

- **błąd w kodzie**, spowodowany przez roztargnionego, zmęczonego (lub poganianego) programistę,
- **błąd niezrozumienia kodu** – programista, dostając kod po kimś innym lub wracając do swojego starego kodu, nie wie lub nie pamięta „*o co w nim chodzi*”,
- **błąd projektowania** – nieprzemyślana architektura systemu,
- **błędy komunikacji** – niezrozumienie między klientem, a programistą,
- **błąd w oszacowaniu czasu** trwania projektu.

Dług technologiczny

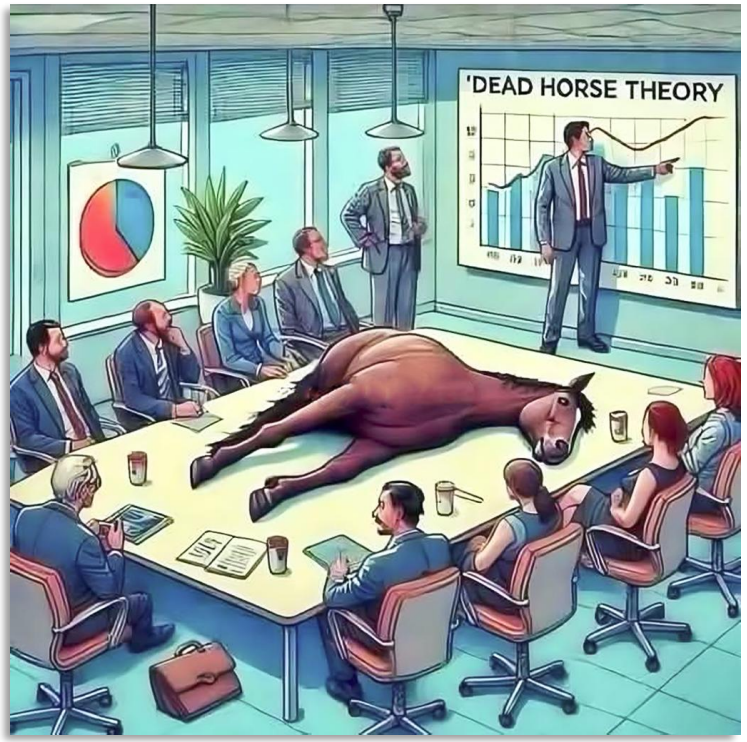
Dług technologiczny (techniczny, projektowy, kodowy) to pojęcie określające sytuację, w podejmowane **decyzje** projektowe, są **szybkie i wygodne** w krótkim terminie, ale **prowadzą do większych kosztów w przyszłości**.

Mimo że firma oszczędza dzięki wdrażaniu szybciej i taniej stworzonego rozwiązania musi ponosić tego koszty później. Odsetki od zaciągniętego długu z czasem rosną i staje się on trudniejszy do spłacenia, ponieważ oprogramowanie rodzi coraz więcej problemów i błędów.

Teoria martwego konia

„Kiedy odkryjesz, że jeździsz na koniu, a koń pada, to najlepszą strategią jest zsiąść”

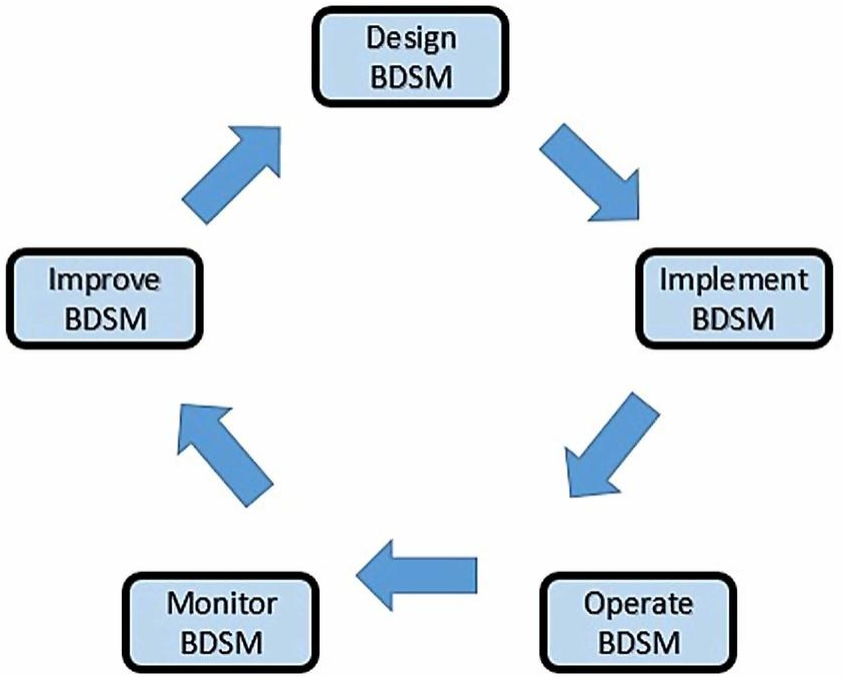
Martwy koń to potoczne określenie na produkt, którego oddanie do użytku jest **działaniem bez szans na powodzenie**. Teoria martwego konia prezentuje model wielu firm, w którym naturalnym odruchem menedżerów jest nakłanianie ludzi do **cięższej i szybszej pracy.**



- Kupno mocniejszego bata.
- Groźba w stosunku do konia.
- Powołanie komisji do zbadania konia.
- Zorganizowanie wyjazdu aby zobaczyć, jak jeżdżą na martwych koniach gdzie indziej.
- Obniżenie standardów, tak aby można było uwzględnić martwe konie.
- Wyznaczenie zespołu interwencyjnego do reanimacji martwego konia.
- Zmiana norm, tak aby koń mógłby być uznany za żywego.
- Awansowanie martwego konia na stanowisko nadzorcze.

Akronimy i nazwy aplikacji

Przed utworzeniem akronimu upewnij się, że nie jest on już używany.

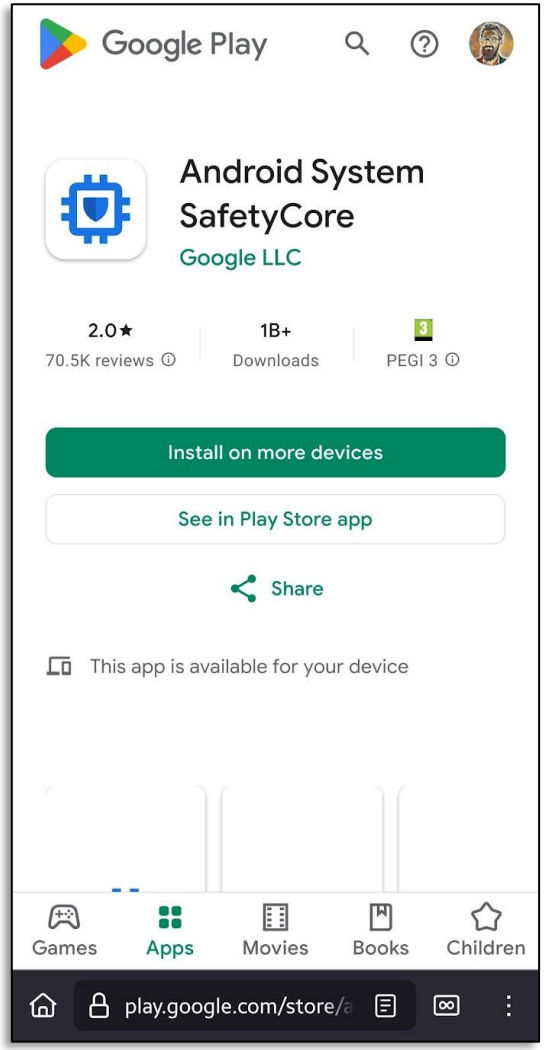


Deep tech

Obszar biznesu lub badań, który obejmuje wykorzystanie zaawansowanej nauki i technologii, takich jak sztuczna inteligencja, blockchain lub postępy w biotechnologii, w celu rozwiązania skomplikowanych problemów.

W kontekście biznesowym start-upy z sektora deep tech wyróżniają trzy kluczowe cechy: wysoki potencjał wywierania wpływu zarówno na rynek, jak i świat, duże wymagania kapitałowe oraz długi czas potrzebny na osiągnięcie dojrzałości rynkowej.

NEW FEATURE



1. **Open Settings**: Go to your device's Settings app
2. **Access Apps**: Tap on 'Apps' or 'Apps & Notifications'
3. **Show System Apps**: Select 'See all apps' and then tap on the three-dot menu in the top-right corner to choose 'Show system apps'
4. **Locate SafetyCore**: Scroll through the list or search for 'SafetyCore' to find the app
5. **Uninstall or Disable**: Tap on Android System SafetyCore, then select 'Uninstall' if available. If the uninstall option is grayed out, you may only be able to disable it
6. **Manage Permissions**: If you choose not to uninstall the service, you can also check and try to revoke any SafetyCore permissions, especially internet access

Note: depending on the software version and manufacturer of your device, these instructions may be slightly off. I personally couldn't test them because my Samsung has not received the October patch yet due to the patch gaps.

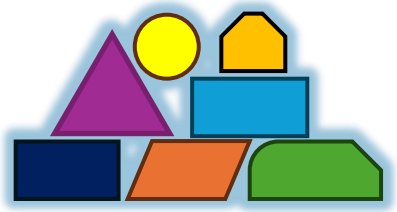
Google wykorzystuje AI do analizowania obrazów na urządzeniu użytkownika w celu identyfikacji i, w razie potrzeby, blokowania nagich zdjęć.

- Od października 2024 r. Google skanuje urządzenia z Androidem w poszukiwaniu nagich zdjęć otrzymanych za pośrednictwem Wiadomości Google.
- Chociaż Google zapewnia, że analiza obrazu odbywa się lokalnie na urządzeniu, pozostają pytania dotyczące przetwarzania i możliwej komunikacji z serwerami zewnętrznymi.
- Użytkownicy mogą dezaktywować funkcję „SafetyCore", ale niektórzy zgłaszają, że jest ona automatycznie ponownie aktywowana po krótkim czasie.

Wzorce projektowe

Wzorce projektowe to typowe rozwiązania problemów często napotykanych przy projektowaniu oprogramowania. Stanowią coś na kształt gotowych planów które można dostosować, aby rozwiązać powtarzający się problem w kodzie.

WZORCE ARCHITEKTONICZNE ≠ **WZORCE PROJEKTOWE**



Wzorce projektowe

1. **Wzorce kreacyjne** wprowadzają elastyczniejsze mechanizmy tworzenia obiektów i pozwalają na ponowne wykorzystanie istniejącego kodu.

   a. **Factory method** (*metoda wytwórcza*) - udostępnia interfejs do tworzenia obiektów w ramach klasy bazowej, ale pozwala podklasom zmieniać typ tworzonych obiektów.

   b. **Abstract factory** (*fabryka abstrakcyjna*) - pozwala tworzyć rodziny spokrewnionych ze sobą obiektów, bez określania ich konkretnych klas.

   c. **Builder** (*budowniczy*) - daje możliwość tworzenia złożonych obiektów etapami, krok po kroku. Wzorzec ten pozwala produkować różne typy oraz reprezentacje obiektu używając tego samego kodu konstrukcyjnego.

   d. **Prototype** (*prototyp*) - umożliwia kopiowanie już istniejących obiektów bez tworzenia zależności pomiędzy kodem, a klasami obiektów.

   e. **Singleton** - pozwala zapewnić istnienie wyłącznie jednej instancji danej klasy. Ponadto daje globalny punkt dostępowy do tejże instancji.

Wzorce projektowe

2. **Wzorce strukturalne** wyjaśniają jak składać obiekty i klasy w większe struktury, zachowując przy tym elastyczność i efektywność struktur.

   a) **Adapter** (*adapter*) - pozwala na współdziałanie obiektów o niekompatybilnych interfejsach.

   b) **Bridge** (*most*) - pozwala na rozdzielenie dużej klasy na dwie hierarchie: abstrakcję oraz implementację. Nad obiema można wówczas pracować niezależnie.

   c) **Composite** (*kompozyt*) - pozwala komponować obiekty w struktury drzewiaste, a następnie traktować te struktury jakby były osobnymi obiektami.

   d) **Decorator** (*dekorator*) - pozwala dodawać nowe obowiązki obiektom poprzez umieszczanie tych obiektów w specjalnych obiektach opakowujących, które zawierają odpowiednie zachowania.

   e) **Fasade** (*fasada*) - wyposaża bibliotekę, framework lub inny zestaw klas w uproszczony interfejs.

   f) **Flyweight** (*pyłek*) - pozwala zmieścić więcej obiektów w danej przestrzeni pamięci RAM poprzez współdzielenie części opisu ich stanów.

   g) **Proxy** (*pełnomocnik*) - pozwala stworzyć obiekt zastępczy w miejsce innego obiektu. Nadzoruje dostęp do pierwotnego obiektu.

Wzorce projektowe

3. **Wzorce behawioralne** które zajmują się komunikacją i podziałem obowiązków pomiędzy obiektami.

   a) **Chain of Responsibility** (łańcuch zobowiązań) – pozwala przekazywać żądania wzdłuż łańcucha obiektów obsługujących.

   b) **Command** (polecenie) - zmienia żądanie w samodzielny obiekt zawierający wszystkie informacje o tym żądaniu. Umożliwia opóźnianie lub kolejkowanie wykonywania żądań oraz cofanie operacji.

   c) **Iterator** - pozwala sekwencyjnie przechodzić od elementu do elementu jakiegoś zbioru bez konieczności eksponowania jego formy (lista, stos, drzewo, itp.).

   d) **Mediator** - ogranicza bezpośrednią komunikację pomiędzy obiektami i zmusza je do współpracy wyłącznie za pośrednictwem obiektu mediatora

   e) **Memento** (pamiątka) - pozwala zapisywać i przywracać wcześniejszy stan obiektu bez ujawniania szczegółów jego implementacji.

Wzorce projektowe

f) **Observer** (obserwator) - pozwala zdefiniować mechanizm subskrypcji w celu powiadamiania wielu obiektów o zdarzeniach dziejących się w obserwowanym obiekcie.

g) **State** (stan) - pozwala obiektowi zmienić swoje zachowanie gdy zmieni się jego stan wewnętrzny.

h) **Strategy** (strategia) - pozwala zdefiniować rodzinę algorytmów, umieścić je w osobnych klasach i uczynić obiekty tych klas wymienialnymi.

i) **Template Method** (metoda szablonowa) - definiuje szkielet algorytmu w klasie bazowej, ale pozwala podklasom nadpisać etapy tego algorytmu bez zmiany jego struktury.

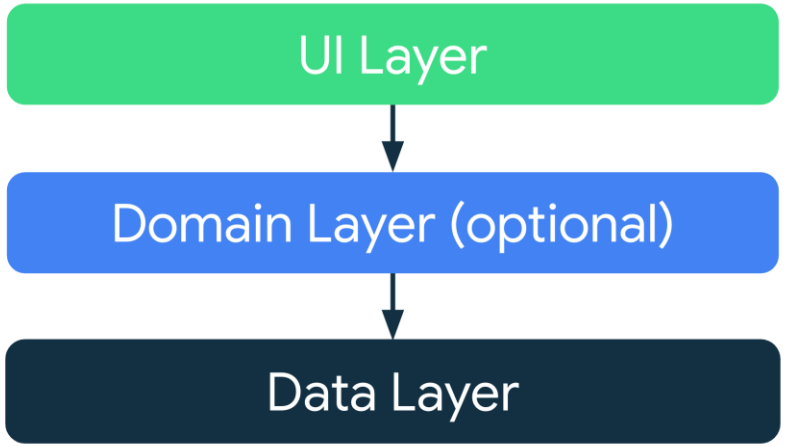
j) **Visitor** (odwiedzający) - pozwala oddzielić algorytmy od obiektów na których pracują.

Architektura aplikacji

Architektura aplikacji określa granice między częściami aplikacji i obowiązki, jakie powinna mieć każda z nich.

Wzorzec architektoniczny to **rozwiązanie wielokrotnego użytku** dla **problemu** występującego **w architekturze oprogramowania**.

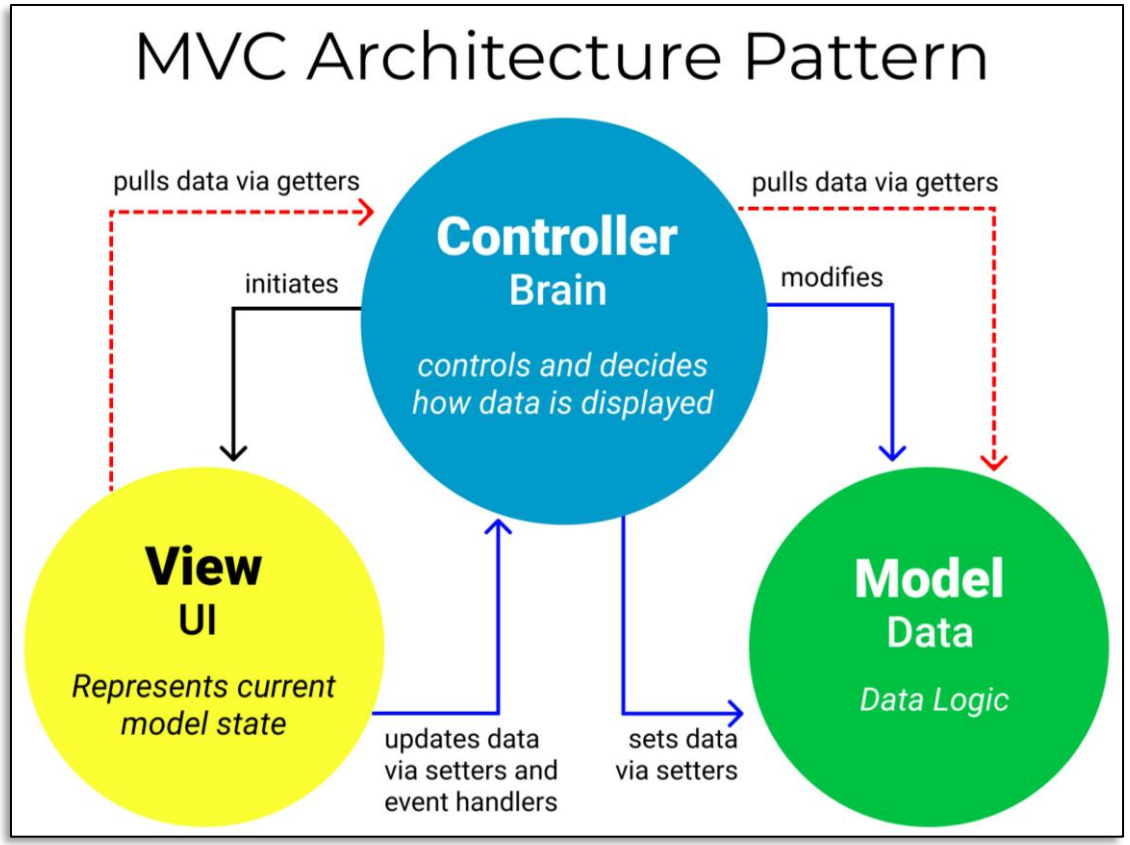
1. **MVC** (Model View Controller)
2. **MVP** (Model View Presenter)
3. **MVVM** (Model View ViewModel)



MVC (Model-View-Controller)

MVC to wzorzec dzielący aplikację na komponenty **Modelu**, **Widoku** i **Kontrolera**.

Służy do organizowania struktury aplikacji posiadających graficzne interfejsy użytkownika.

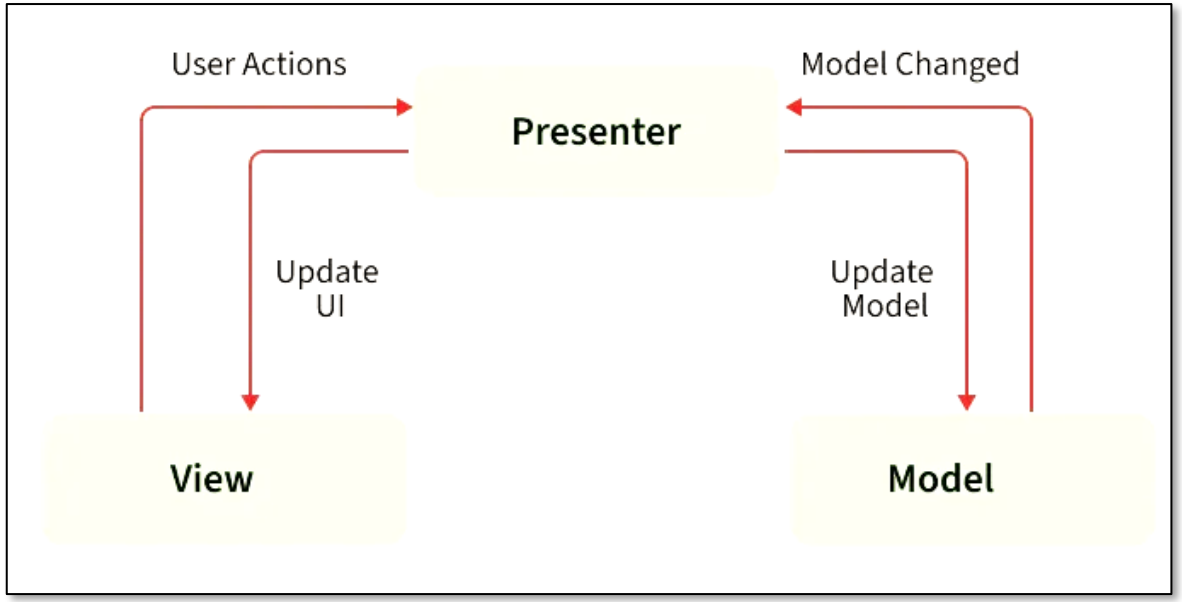


1. **Model -** przechowuje dane aplikacji,
2. **Widok** – warstwa UI, która przechowuje komponent widoczny na ekranie,
3. **Kontroler** - ustanawia relację między Modelem a Widokiem

MVP (Model View Presenter)

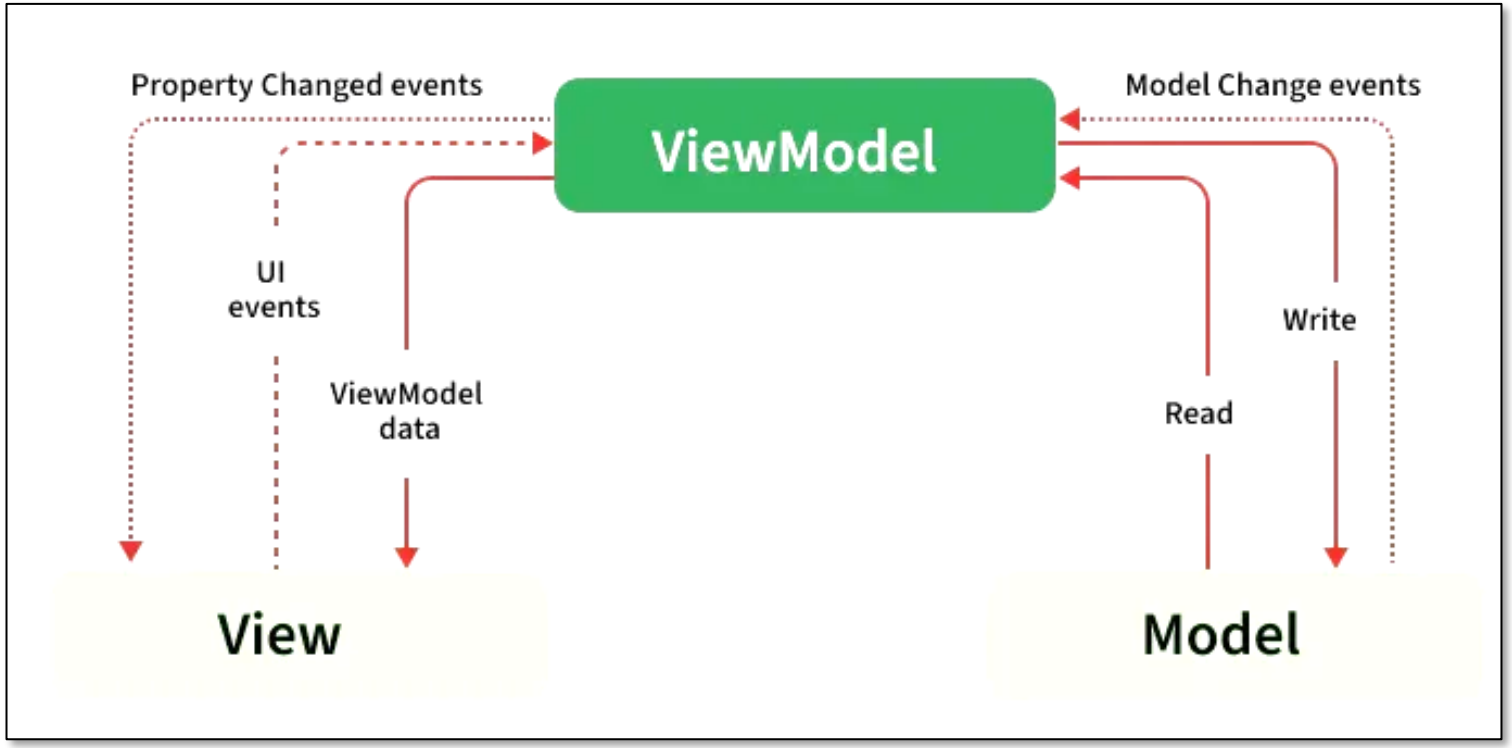
MVP - Aby uniknąć złożoności, takich jak łatwość konserwacji, czytelność, skalowalność i refaktoryzacja aplikacji, używamy modelu MVP.

1. Komunikacja między **View-Presenter** i **Presenter-Model** odbywa się za pomocą **interfejsu** (zwanego również kontraktem).
2. Pomiędzy **Prezenterem** a **Widokiem** istnieje relacja jeden do jednego, jedna klasa prezentera zarządza tylko jednym widokiem w danym czasie.
3. **Model** i **Widok** nie mają żadnej wiedzy o sobie nawzajem.



MVVM (Model View ViewModel)

MVVM, jak sama nazwa wskazuje, podobnie jak model MVC, zawiera trzy komponenty: **Model**, **Widok** i **ViewModel**.



Korzyści płynące z architektury

1. Poprawia łatwość utrzymania, jakość i solidność aplikacji

2. Skalowalność aplikacji jest zwiększona przy użyciu architektury, ponieważ użytkownicy i programiści nie mogą zwiększyć zaangażowania w aplikację.

3. Testowanie aplikacji staje się dzięki temu łatwiejsze.

4. Błędy można łatwo zidentyfikować i usunąć dzięki dobrze zdefiniowanemu procesowi.

Architektura aplikacji

Rozdzielenie zagadnień (separation of concerns)

Częstym błędem jest pisanie całego kodu w **Activity** lub **Fragment**. Te klasy oparte na interfejsie użytkownika (UI) powinny zawierać tylko logikę, która obsługuje UI i interakcje z systemem operacyjnym. Utrzymując te klasy tak szczupłe, jak to tylko możliwe, można uniknąć wielu problemów związanych z cyklem życia komponentu i poprawić testowalność tych klas.

Pamiętaj, że nie jesteś właścicielem implementacji **Activity** i **Fragment.** System operacyjny może je zniszczyć w dowolnym momencie na podstawie interakcji użytkownika lub z powodu warunków systemowych, takich jak niski poziom pamięci. Aby zapewnić łatwiejsze zarządzanie konserwacją aplikacji, najlepiej jest **zminimalizować zależność od nich**.

Sterowanie interfejsem użytkownika z modeli danych

Interfejs użytkownika powinien opierać się na modelach danych, najlepiej modelach trwałych. Modele danych reprezentują dane aplikacji. Są one niezależne od elementów UI i innych komponentów aplikacji. Nie są one powiązane z UI i cyklem życia komponentów aplikacji, ale nadal zostaną zniszczone, gdy system operacyjny zdecyduje się usunąć proces aplikacji z pamięci. Modele trwałe są idealne z następujących powodów:

- Użytkownicy nie tracą danych, jeśli system operacyjny Android zniszczy aplikację w celu zwolnienia zasobów.
- Aplikacja nadal działa w przypadkach, gdy połączenie sieciowe jest słabe lub niedostępne.

Jeśli oprzesz architekturę aplikacji na klasach modelu danych, sprawisz, że Twoja aplikacja będzie bardziej testowalna i niezawodna.

Architektura aplikacji

Pojedyncze źródło prawdy (Single Source of Truth)

Po zdefiniowaniu nowego typu danych w aplikacji należy przypisać do niego pojedyncze źródło prawdy (Single Source of Truth, SSOT). SSOT jest właścicielem tych danych i tylko SSOT może je modyfikować lub mutować. Aby to osiągnąć, SSOT eksponuje dane przy użyciu niezmiennego typu, a aby zmodyfikować dane, SSOT eksponuje funkcje lub odbiera zdarzenia, które mogą wywoływać inne typy. Ten wzorzec przynosi wiele korzyści:

- Centralizuje wszystkie zmiany danych w jednym miejscu.
- Chroni dane, aby inne typy nie mogły nimi manipulować.
- Sprawia, że zmiany w danych są bardziej identyfikowalne, więc błędy są łatwiejsze do wykrycia.

W aplikacji offline-first źródłem prawdy dla danych aplikacji jest zazwyczaj baza danych. W niektórych innych przypadkach źródłem prawdy może być ViewModel lub nawet UI.

Czym jest Android

1. Android to **mobilny system operacyjny** o otwartym kodzie źródłowym. **Projekt Android Open Source** pozwala każdemu pobrać kod źródłowy lub przejrzeć dokumentację związaną z nowymi wersjami systemu.

2. System operacyjny Android to **wieloużytkownikowy system Linux**, w którym **każda aplikacja jest innym użytkownikiem**.

3. System przypisuje każdej aplikacji unikalny **identyfikator użytkownika** (ID) Linux, który jest używany tylko przez system i jest nieznany aplikacji. System ustawia uprawnienia dla wszystkich plików w aplikacji, aby tylko ID przypisany do tej aplikacji mógł uzyskać do nich dostęp.

Czym jest Android

4. Każdy **proces** ma **własną maszynę wirtualną (VM)**, więc kod aplikacji działa w izolacji od innych aplikacji. Domyślnie każda aplikacja działa we własnym procesie Linux. System Android **uruchamia proces**, gdy którykolwiek ze **składników aplikacji** musi zostać wykonany, a następnie **zamyka proces**, gdy **nie jest już potrzebny** lub gdy **system musi zwolnić pamięć** dla innych aplikacji.

5. System Android implementuje **zasadę najmniejszych uprawnień** (*principle of least privilege*). Oznacza to, że każda aplikacja ma domyślnie dostęp tylko do tych komponentów, których potrzebuje do wykonania swojej pracy i nie więcej.

Komponenty aplikacji

Komponenty to **podstawowe elementy składowe aplikacji**. Każdy komponent jest **punktem wejścia**, przez który **system** lub **użytkownik** może wejść do aplikacji.

1. **Activities** (aktywności, działania)
2. **Services** (usługi)
3. **Broadcast receivers** (odbiorniki sygnałów, transmisji)
4. **Content providers** (dostawcy treści lub usług)

Aktywności

Aktywność implementuje się jako podklasę klasy ***Activity***. Umożliwia ona **interakcję użytkownika z aplikacją**. Zazwyczaj reprezentuje **pojedynczy ekran** z interfejsem użytkownika. Przykłady:

- Aplikacja poczty e-mail może mieć jedną aktywność, która pokazuje listę nowych wiadomości e-mail, inną aktywność do tworzenia wiadomości e-mail i inną aktywność do czytania wiadomości e-mail. Chociaż aktywności współpracują ze sobą, **każda z nich** jest **niezależna od innych**.

- Inna aplikacja może **uruchomić dowolną** z jej **aktywności**. Aplikacja aparatu może uruchomić aktywność utworzenia nowej wiadomości e-mail, aby umożliwić użytkownikowi udostępnienie zdjęcia.

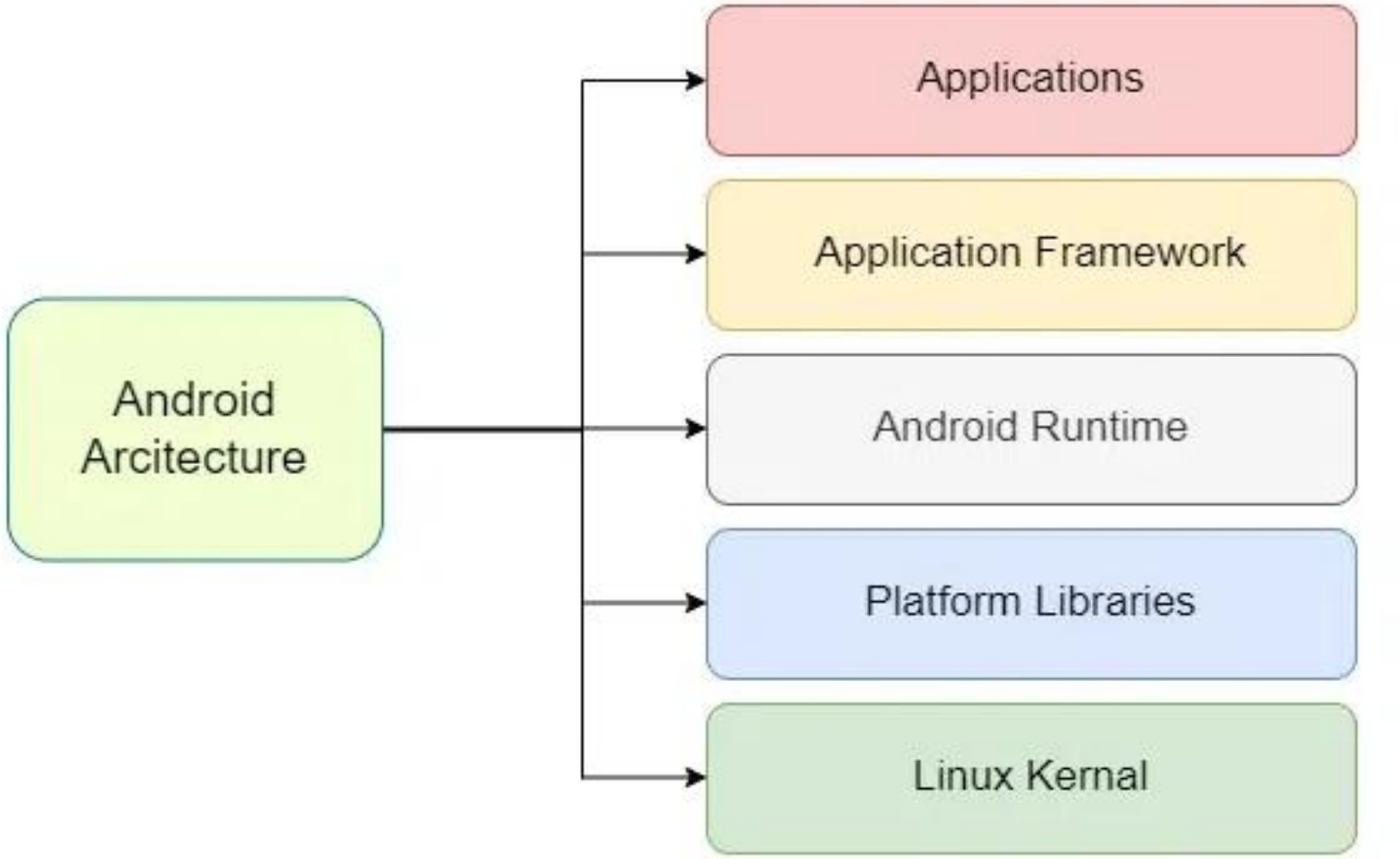
Usługi

Usługa jest zaimplementowana jako podklasa klasy ***Service***

Usługa umożliwia utrzymywanie aplikacji **działającej w tle,** aby wykonywać długotrwałe operacje lub wykonywać pracę dla zdalnych procesów.

Usługa **nie zapewnia interfejsu użytkownika**.

Architektura aplikacji



Komponenty aplikacji

Narzędzia **pakietu Android SDK** kompilują kod wraz z danymi i plikami zasobów do **pliku APK** lub **pakietu Android App Bundle**.

- **APK** jest **plikiem archiwum**, zawiera zawartość aplikacji na Androida wymaganej w czasie jej działania.
- **Android App Bundle**, który jest **plikiem archiwum**, **zawiera zawartość projektu aplikacji** na Androida, w tym **dodatkowe metadane**, które nie są wymagane na w środowisku wykonawczym. Pakiet aplikacji na Androida jest formatem publikowania i **nie można go zainstalować na urządzeniu**.
- Komponenty aplikacji zgłaszane są w **manifeście aplikacji**. Na jego podstawie system operacyjny integruje aplikację z ogólnymi funkcjami urządzenia,

Manifest aplikacji

Każdy projekt aplikacji musi mieć plik **AndroidManifest.xml**, w katalogu głównym projektu. Dla każdego komponentu aplikacji należy zadeklarować odpowiedni element XML w pliku manifestu:

- **<activity>** for each subclass of Activity
- **<service>** for each subclass of Service
- **<receiver>** for each subclass of BroadcastReceiver
- **<provider>** for each subclass of ContentProvider

```
<manifest ... >
    <application ... >
        <activity android:name="com.example.myapp.MainActivity" ... >
        </activity>
    </application>
</manifest>
```

```
<manifest ... >
    <application ... >
        <activity android:name=".MainActivity" ... >
            ...
        </activity>
    </application>
</manifest>
```

Dobra architektura

Ze względu na to, że zwykła aplikacja na Androida może zawierać wiele **komponentów**, a użytkownicy często korzystają z **wielu aplikacji** w krótkim czasie, aplikacje muszą się dostosować do różnych procesów i zadań związanych z działaniami użytkownika.

System operacyjny może więc w każdej chwili przerwać niektóre procesy aplikacji, żeby zrobić miejsce na nowe.

W warunkach tego środowiska komponenty aplikacji mogą być uruchamiane pojedynczo lub w złej kolejności, a system operacyjny lub użytkownik może je zniszczyć w dowolnym momencie. Zdarzenia te nie są pod Twoją kontrolą,

Dlatego nie przechowujemy ani nie przechowuj w pamięci żadnych danych ani stanu aplikacji w komponentach aplikacji. Komponenty aplikacji nie powinny być od siebie zależne.