POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Patryk Roguszewski
Nr albumu s13054

Wykorzystanie wzorców projektowych w tworzeniu wieloplatformowych aplikacji moblinych

Praca magisterska napisana pod kierunkiem

dr inż. Mariusz Trzaska

Warszawa, Lipiec, 2020

Streszczenie

Praca dotyczy problemu dopasowania odpowiednich wzorców projektowych dla tworzenia wieloplatformowych aplikacji mobilnych. Programowanie systemów dedykowanych na urządzenia przenośne wraz z biegiem czasu staję się coraz łatwiejsze. Znacznie trudniejszym zadaniem jest etap rozszerzania funkcjonalności produktu, poprzez zmiany w istniejącej bazie kodowej, nie wpływając przy tym negatywnie na dotychczasowe działanie.

Jednym z rozwiązań problemów związanych z utrzymaniem aplikacji podczas modyfikacji są uznawane przez środowisko wzorce projektowe, dzielące projekt na wiele warstw zależnych od pełnionych odpowiedzialności.

Praca opisuje wpływ zastosowanych rozwiązań architektonicznych dla technologii dedykowanych do tworzenia aplikacji mobilnych na najpopularniejsze, przenośnie systemy operacyjne Android oraz IOS, zachowując jak największą, wspólną bazę kodową. Analiza przydatności wzorców została przeprowadzona przy wykorzystaniu rozwiązania stworzonego w technologiach Xamarin.Forms oraz Flutter.

Słowa kluczowe: Flutter, Xamarin, Mobile, Android, iOS.

Spis treści

1. Wstęp

Przez ostatnie lata popyt na tworzenie aplikacji mobilnych odnotował wyraźny wzrost. Większa część społeczeństwa zmieniła codzienny rytuał przeglądania zasobów informatycznych poprzez wymianę tradycyjnych komputerów na urządzenia przenośne. Stało się to motorem dla tworzenia oprogramowania na systemy operacyjne takie jak Android czy iOS.

Podczas tego okresu, proces tworzenia oprogramowania na platformy mobilne uległ polepszeniu. Aplikacje dedykowane na urządzenia przenośne wymagają mniej czasu na ich implementację. Powstało wiele technologii umożliwiających kompatybilność cyfrowych produktów z najpopularniejszymi systemami operacyjnym przy zachowaniu jak największej, wspólnej bazy kodowej.

Niestety, jakość dostarczanych aplikacji uległa pogorszeniu. Wiele produktów tuż po ich wdrożeniu na sferę produkcyjną posiada podobne do siebie błędy w działaniu. Wynika to z zaniedbań twórców, często próbujących tworzyć własne standardy, odbiegające od ogólnie przyjętych norm. Jednym z rozwiązań, wpływającym na jakość powstałego systemu są sprawdzone, opisane przez doświadczonych, uznanych autorytetów w dziedzinie programowania - wzorce projektowe.

1.1. Organizacja pracy

Pierwsze dwa rozdziały pracy przedstawiają wartości wdrożenia wzorców projektowych dla mobilnych systemów informatycznych oraz kolejno, odsłaniają techniczne zaplecze użyte podczas tworzenia rozwiązania.

Kolejny rozdział dotyczy koncepcji dla nowego wzorca projektowego umożlwiającego operacje na danych między klientem, a serwerem.

Piąty rozdział pracy opisuję wymagania biznesowe, projektowanie prototypu oraz architekturę aplikacji, na której zostało przeprowadzone badanie.

Na końcu dokumentu przedstawiona jest analiza wpływu wzorców projektów na całość powstałego systemu pod kątem wydajności, bezpieczeństwa, błędogenności czy trudności dla wdrożenia nowych osób – progiem wejścia.

1.2. Cel pracy

Celem pracy jest udokumentowanie wartości płynących z korzystania wcześniej powstałych wzorców projektów w kontekście tworzenia aplikacji mobilnych.

1.3. Rozwiązanie przyjęte w pracy

Implementacja prototypu części klienckiej została wykonana przy użyciu dwóch różnych technologii umożliwiających tworzenie wieloplatformowych aplikacji mobilnych – Flutter oraz Xamarin.Forms.

Za część serwerową pracy odpowiedzialną za bezpieczny kontakt pomiędzy zewnętrznym środowiskiem, a źródłem wrażliwych informacji jest REST API, wykonane w technologii .Net Core 3.0. Za warstwę trwałości danych odpowiada relacyjna baza danych PostgreSQL

1.4. Rezultaty pracy

Rezultatem pracy jest wynik analizy wpływu wzorców projektów w technologiach umożliwiających tworzenie wieloplatformowych aplikacji mobilnych. Następstwem tego wymagania są prototypy aplikacji dla systemów przenośnych umożliwiających zapisywanie informacji w formie krótkich notatek.

2. Rola wzorców projektowych w tworzeniu projektów informatycznych

Wzorzec opisuje problem, który powtarza się wielokrotnie w danym środowisku, oraz podaje istotę jego rozwiązania w taki sposób, aby można było je zastosować miliony razy bez potrzeby powtarzania tej samej pracy [1].

2.1. Korzyści płynące z wykorzystania wzorców projektowych

Poprzez regularne stosowanie wzorców projektowych w procesie tworzenia systemu informatycznego można uzyskać szereg benefitów podnoszących jakość produktu końcowego.

Istotną zaletą wielu wzorców informatycznych jest standaryzacja kodu. Dzięki unormowaniu struktury rozwiązanie staję się bardziej przejrzyste. Wyszukiwanie istniejących bloków jest o wiele szybsze. Oddziałuje to również na wdrożenie nowych osób do istniejącego już projektu. Programista, oddelegowany do pracy projekcie, posiadający wiedze na temat zastosowanej architektury ma znacznie mniejszy próg wejścia. Obfituje to potencjalnie krótszym procesie implementacji produktu, czego rezultatem jest zmniejszenie kosztów projektu.

Każdy wzorzec projektowy ma swoją genezę u podstaw rozwiązania danego problemu. Kiedy następuję decyzja skorzystania z wybranej architektury, wiążę się ona większą poprawnością przyszłego systemu, gdyż aplikacja jest pozbawiona błędu już dzięki sposobie implementacji. Stare powiedzenie „Najlepiej uczyć się na cudzych błędach niż swoich" najlepiej oddaje treść tego akapitu.

Wzorce projektowe przede wszystkim pozwalają swobodnie rozszerzać aplikacje o nowe funkcjonalności, nie naruszając przy tym obecnej struktury. Praca w dobrej architekturze pozwala zachować porządek oraz poprawność działania aplikacji. Kolejne dodane funkcje według wybranego wcześniej standardu, nie nakładają się oraz zapobiegają tworzeniu się między sobą wspólnych zależności. Im większe koneksje pomiędzy danymi komponentami, tym łatwiej wpłynąć na poboczne podmioty edytując jedno miejsce.

Wzorce projektowe można porównać do przepisów kulinarnych. Dobra receptura pozwala stworzyć sprawdzone i uznane przez społeczność rozwiązanie. Działa to również w kontekście wydajności. Wiele uznanych wzorców projektowych zapobiega takim problemom jak wycieki pamięci czy ograniczona liczba ilości połączeń.

Podsumowując, dzięki odpowiedniemu użyciu wzorców projektowych możemy uzyskać:

- Wydajność,
- Czytelność,
- Obniżenie progu wejścia w bazę kodową,
- Mniejszą błędogenność systemu w czasie modyfikacji,
- Uniknięcie potencjalnych błędów.

2.2. Wady płynące z wykorzystania wzorców projektowych

Nieumiejętne korzystanie z wzorców projektowych może wprowadzić więcej szkód niż potencjalnych korzyści. Nieprawidłowe zestawienie kilku uznanych praktyk może wbrew pozorom zmniejszyć czytelność projektu poprzez rozproszenie odpowiedzialności bloków kodów.

Kiedy w jednej aplikacji kilka wzorców, rozwiązuję ten sam problem, wywołuję to odwrotny skutek niż pierwotne zamierzenie. Wiele warstw architektonicznych odpowiedzialnych za to samo zadanie, niesie z sobą większe ryzyko wystąpienia błędu wpływającego na poprawne działanie systemu. Powyższy proces ma również wpływ na wydajność aplikacji. Im więcej warstw systemu jest odpowiedzialnych za tą samą pracę, tym będzie dłuższy czas jej wykonania.

Należy również pamiętać, że korzystanie z wzorców projektowych jest drogą do osiągnięcia celu, a nie jej celem. Czasem należy obejść pewien wzorzec bądź nagiąć go pod własną potrzebę. Jeżeli złamanie ustalonego schematu nie wiążę się z ryzykiem zniszczenia całej architektury, a pozwoli osiągnąć dany cel należy rozważyć jej wykorzystanie. Środowisko programistyczne często potrafi dążyć w swej ambicji do stworzenia perfekcyjnego kodu zapominając o istocie, najważniejszej części całego procesu - stworzonej aplikacji.

Obszerność bazy kodowej produktu, jest równie istotna. Nie zawsze opłaca się implementacja skomplikowanego wzorca dla małego fragmentu kodu. Gdy czas przeznaczony na wdrożenie architektury jest większy bądź porównywalny do stworzenia produktu, warto się zastanowić nad użytecznością rozwiązania.

Podsumowując, by dobrze wykorzystać uznane patenty architektonicznie należy unikać:

- Kombinacji wzorców projektowych o podobnej odpowiedzialności,
- Zamiany miejscami drogi wraz z jej celem. Wzorce są tylko pomocą,
- Czas implementacji wzorca jest większy lub porównywalny do czasu stworzenia produktu.

2.3. Proces wdrażania wzorcu projektowego

Najlepszym momentem dla wdrożenia wzorca jest początek tworzenia aplikacji. Architekt oprogramowania po wcześniejszej analizie domeny powinien z góry przewidzieć potencjalne zagrożenia oraz wybrać główny standard dla tworzenia oprogramowania, korzystając z kombinacji wzorców rozwiązującej największą ilość problemów. Rezultatem tego zabiegu jest duży poziom zgodności kodu, gdyż każdy nowo powstały blok jest tworzony w odpowiedniej normie.

Wdrożenie wzorca dla nowego projektu obejmuję:

1. Analizę potencjalnego problemu,
2. Wybór wzorców pomagających w rozwiązaniu zagrożenia,
3. Akceptację schematu o największym wachlarzu możliwości,
4. Brak naruszenia przez kandydata odpowiedzialności dzielącej przez inny komponent systemu.

Inną sytuacją jest wdrożenie wzorca dla systemu z istniejącą, pokaźną już strukturą kodową. Tu należy wziąć pod uwagę istotny element jakim jest koszt implementacji rozwiązania liczony w takich czynnościach jak przerabianie obecnej struktury czy możliwa błędogenność podczas procesu wdrożenia.

Wdrożenie wzorca dla projektu z obszerną bazą kodową obejmuje:

1. Analizę kosztu rozwiązania problemu,
2. Wybór wzorców pomagających w rozwiązaniu zagrożenia,
3. Akceptację schematu o największym wachlarzu możliwości i jak najmniejszym kosztem wdrożenia,
4. Brak naruszenia przez kandydata odpowiedzialności dzielącej przez inny komponent systemu.

2.4. Tworzenie wzorca projektowego

Nie zawsze istniejące rozwiązania pozwalają na dostarczenie satysfakcjonujących efektów. Gdy pojawia się problem na tyle uniwersalny, że możliwe jest stworzenie dla niego powtarzalnego schematu, bardzo prawdopodobnie może być to inicjatorem nowego wzorca projektowego.

Jednak zanim to nastąpi, należy upewnić się, że:

1. Problem nie jest skutkiem wybranej technologii lub innego czynnika środowiska,
2. Założenie wyczerpuję domenę przyczyny,
3. Schemat jest możliwy w przedstawieniu go w formie diagramu.

Spełnienie powyższych warunków, daje możliwość do rozpoczęcia prac nad projektowaniem wzorca. Przy projektowaniu należy pamiętać o istocie uniwersalności rozwiązania, jest to jeden z najważniejszych fundamentów nowopowstającego bytu.

3. Wykorzystane narzędzia i technologie

Praca powstała dzięki wykorzystaniu wielu technologii umożliwiających tworzenie kompleksowych aplikacji mobilnych o najszerszym zakresie możliwości. Wybrano dwa najpopularniejsze na rynku przenośnych urządzeń systemu operacyjne iOS i Android jako docelową platformę produktu końcówki klienckiej.

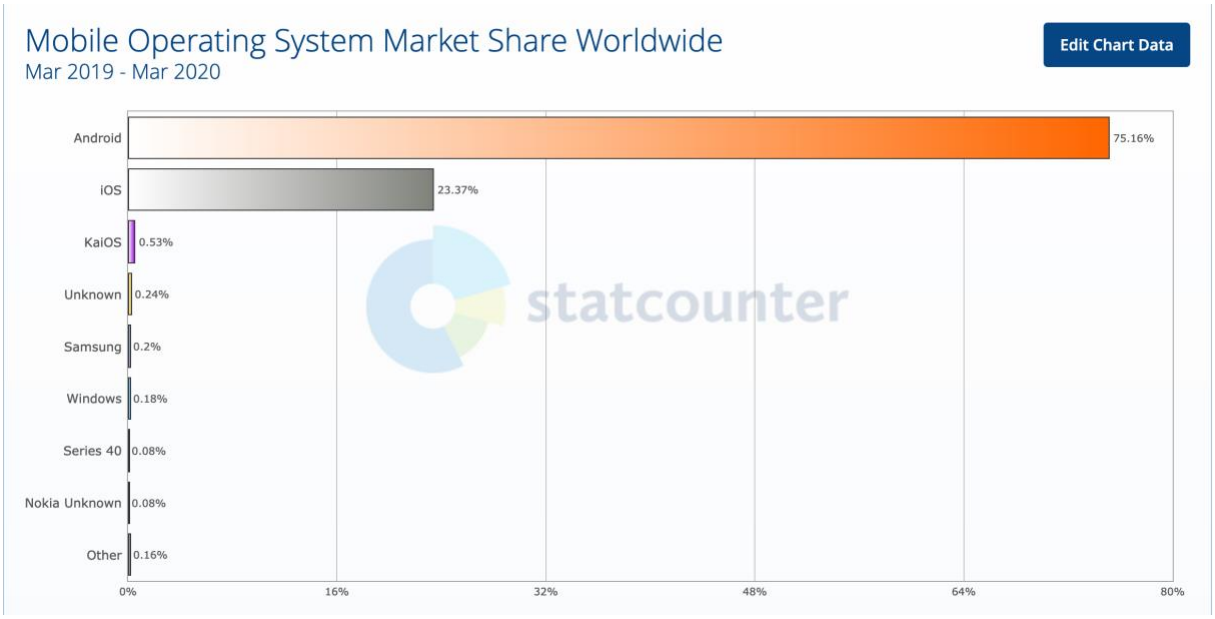
Cześć serwerowa aplikacji jest hostowana na infrastrukturze serwera Linux – Ubuntu Server 18.04 zawierająca wydzielone komponenty pomiędzy kontenerami Docker. Znajduję się również na niej warstwa trwałości danych –relacyjna baza danych PostgreSQL.

Komplikacja jak i tworzenie kodu odbywała się za pomocą systemu operacyjnego MacOS, z tego powodu zdecydowano się na wybranie programistycznych narzędzi oferujących pełne wsparcie dla platformy firmy Apple.

3.1. Systemy operacyjne

System operacyjny to zespół programów zarządzających zasobami urządzenia, definiujących środowisko działania dla innych programów (tzw. maszynę wirtualną) oraz pozwalających użytkownikowi na wykonanie uniwersalnych czynności porządkowych jak i administracyjnych [2].

Obecnie na dzień 23 kwietnia 2020 roku, rynek urządzeń mobilnych jest zdominowany poprzez dwa wiodące systemy operacyjne – Android (75,16 %) oraz iOS (23,37 %). Dokładniejsza statystyka zawierająca zestawienie mobilnych systemów operacyjnych została przedstawiona jest na Rysunku 1.



Rysunek 1. Statystyka mobilnych systemów operacyjnych według serwisu statcounter.com

3.1.1 Android

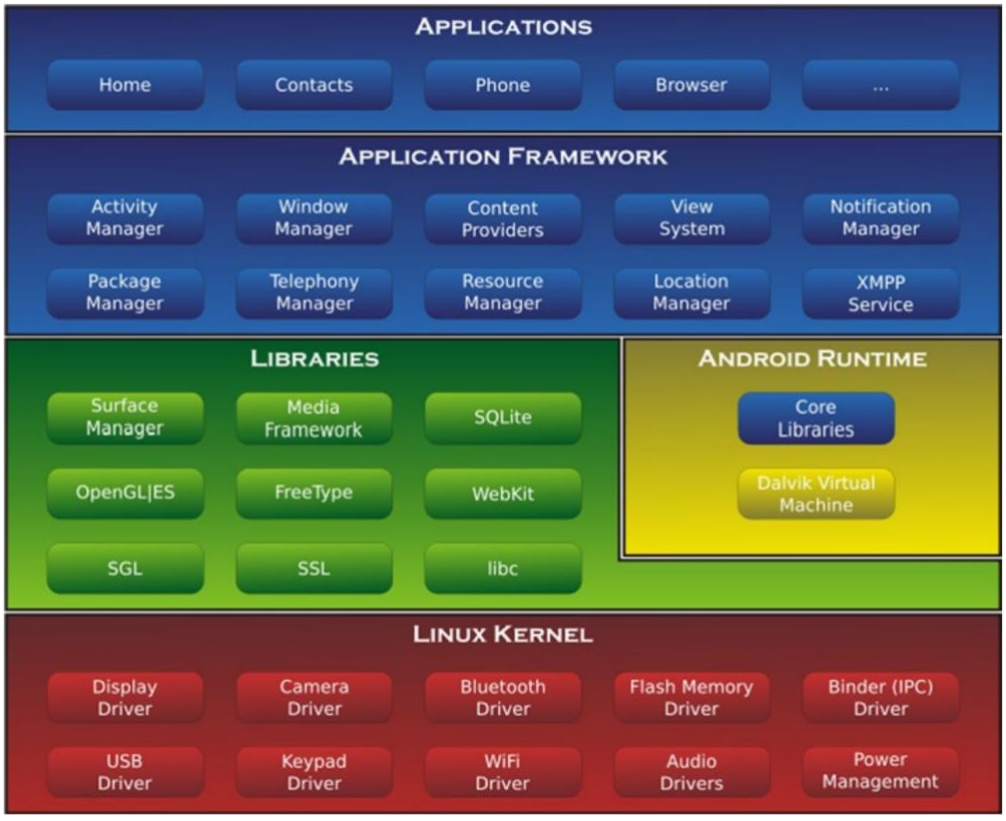
System Android został zaprezentowany w 2007 roku przez firmę Google. Platforma potrafi działać na wielu niezależnych od siebie urządzeniach mobilnych [3].

Jest dostępny na licencji *Open Source Apache Licence*. Rezultatem czego jest brak ograniczeń co do wymaganych środowisk dla tworzenia dedykowanych aplikacji. Do stworzenia programu wymagany jest jedynie zestaw narzędzi Android SDK (ang. *Software Development Kit*), dostępny do pobrania za darmo z strony producenta systemu.

Architektura systemu Android jest warstwowa, zostało to przedstawione na Rysunku 2. Jest zbudowany na jądrze Linux, zawierającym sterowniki dla komponentów technicznych takich jak kamera, wyświetlacz czy bluetooth. Kolejną warstwą są biblioteki odpowiedzialne między innymi za trwałość danych bądź renderowanie obrazu na ekranie urządzenia. Kolejnym etapem jest API, umożliwiające korzystanie z natywnych funkcji urządzenia aplikacjom znajdującym się w najwyższej warstwie [3].

Natywną możliwością tworzenia aplikacji dedykowanych dla środowiska Android jest technologia Java oraz Kotlin. Obie technologie są wspiera przez oficjalną dokumentację dedykowaną dla programistów [12].

Dystrybucja cyfrowych produktów odbywa się za pośrednictwem sklepu Google, znanym z mniejszych restrykcji niż alternatywa konkurencyjnego systemu iOS. Opłaty związane z kontem developerskim również są mniejsze, wynoszą one 25$ jednorazowej wpłaty rejestracyjnej.



Rysunek 2. Architektura systemu Android

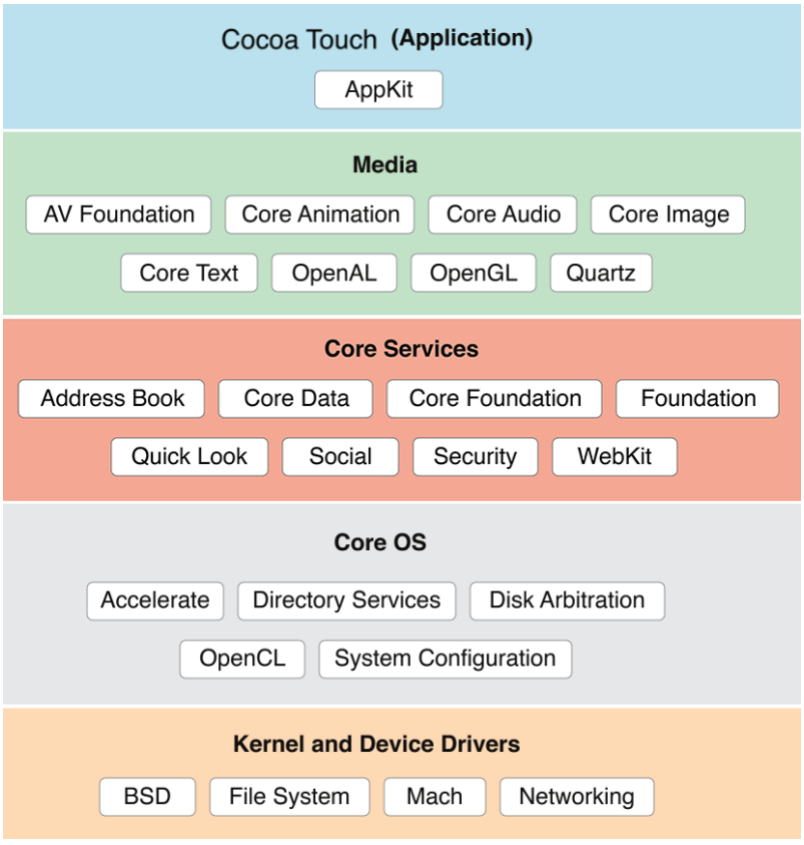
3.1.2 iOS

iOS to system stworzony przez firmę Apple w Styczniu 2007 roku na urządzenia iPhone oraz iPad Touch. Każda nowa wersja urządzenia iPhone zawiera nowe aktualizacje dla systemu podnoszące jego wersje znane jako *major*. iPhone to urządzenie niewielkiego rozmiaru mieszczące się w kieszeni spodni użytkownika zawierające wielodotykowy ekran, kamery oraz urządzenia Audio [3].

Platformę charakteryzuję zamknięte SDK (ang. *Software Development Kit*), czego rezultatem jest brak możliwości tworzenia oprogramowania na iOS za pomocą innego systemu operacyjnego niż macOS. MacOS jest zamkniętym na modyfikacje innych firm produktem firmy Apple, przeznaczonym jedynie na dedykowane urządzenia z serii MacBook Pro, iMac czy MacAir.

Dystrybucja aplikacji mobilnych odbywa się za pośrednictwem sklepu AppStore. Cechuję się wysoką jakością kontroli produktów przed wprowadzeniem ich w obieg. Każdy potencjalny program jest badany oraz oceniany przez wykwalikowanego pracownika zanim zostanie dopuszczony do procesu dystrybucji. Utworzenie konta umożlwiającego dostęp do owej procedury nie jest darmowe, na dzień tworzenia pracy owa opłata wynosi 99$ / 299$ (Enterprise) rocznie.

Architektura systemu iOS została przedstawiona na Rysunku 3. Natywne aplikacje tworzone są w technologii Swift oraz środowisku programistycznym Xcode. Wchodzą one w interakcje z najwyższą warstwą *Cocoa Touch*. Za renderowanie komponentów odpowiada warstwa *Media*, natomiast wszystkie serwisy dostarczane są do aplikacji z *Core Services*. Najniższą, fundamentalną warstwą jest *Core OS*, odpowiada ona za poprawną konfigurację całego systemu.



Rysunek 3. Architektura systemu iOS

3.1.3 Linux

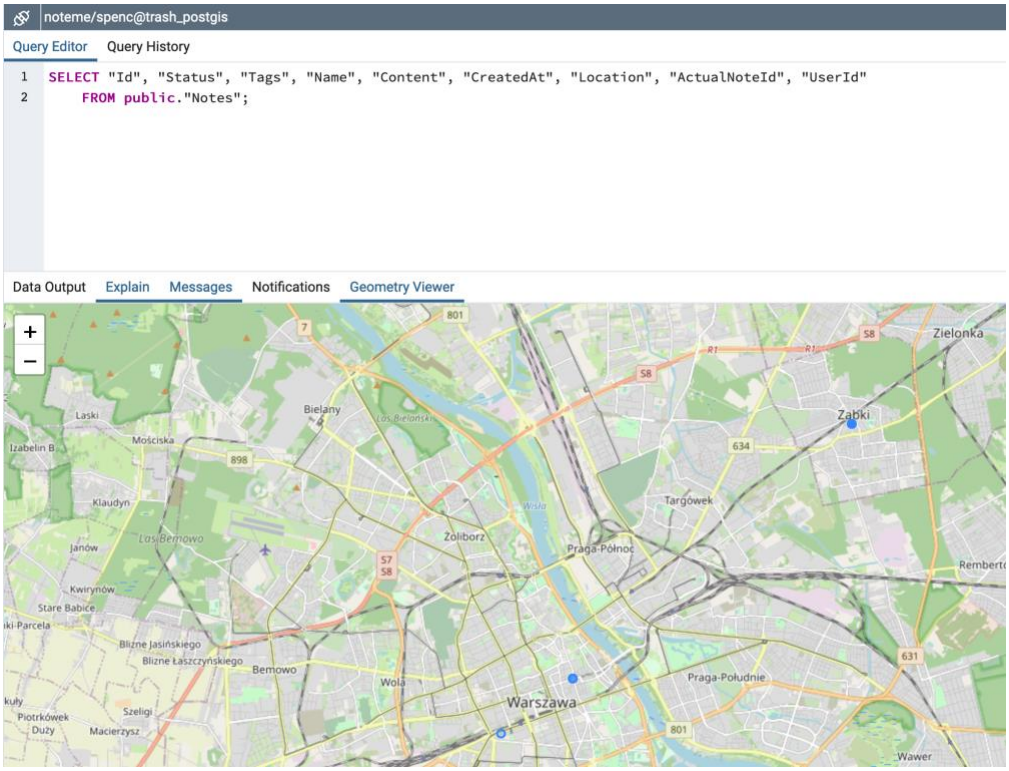
Linux to nazwa określająca rodzinę uniksopodobnych systemów operacyjnych uwarunkowanych na podobnym do siebie jądrze. Głównym jej atutem jest otwarty dostępny dla społeczności kod, w większości implementacji oznaczonym licencją *Open Source Apache Licence*. Najpopularniejszą jego dystrybucją obecnie jest system Android.

Został stworzony przez słynnego oraz uznanego w środowisko programistę Linusa Torvalds. 25 sierpnia 1991 poinformował on użytkowników pewnej informatycznej grupy dyskusyjnej, że tworzy niewielki, darmowy system operacyjny w celach hobbystycznych. Pierwszą jego dystrybucją wydaną w 16 lipca 1993 roku był Slackware Linux, a po kolejnych dwóch miesiącach światło dzienne ujrzała jedna z obecnie najpopularniejszych dystrybucji – Debian [5].

3.2. PostgreSQL

Za warstwę trwałości danych systemu odpowiedzialna jest relacyjna baza danych PostgreSQL. Stworzona została przez Kalifornijski Uniwersytet w Berkley w 1994 roku. Oprogramowanie podlega licencji PostgreSQL License która jest liberalną licencją *Open Source*, podobną do BSD lub MIT [4].

Silnik PostgreSQL posiada wiele dodatkowych rozszerzeń. Jednym z nich jest PostGIS, umożlwiający zapisywanie oraz wizualizację danych geograficznych bez konieczności dodawania kolejnych komponentów warstwy aplikacji stworzonego systemu. Rysunek 4 przedstawia wizualizację danych geograficznych za pośrednictwem oficjalnego narzędzia wspierającego przeglądanie danych bazy PostgreSQL – PgAdmin 4.



Rysunek 4. Podgląd zapisanej lokalizacji użytkownika dzięki rozszerzeniu PostGIS

3.3. Git

Historia wersji kodu prototypu została systematycznie zapisywana za pomocą narzędzia GIT. Narzędzie GIT zostało pierwszy raz opublikowane 7 Kwietnia 2005 roku przez Linusa Torvalds-a. Jest to rozproszony system kontroli wersji przechowujący zestaw migawek (ang. *Snapshot*) każdej zatwierdzonej postaci kodu [6].

Niewątpliwymi zaletami narzędzia są:

- Tworzenie gałęzi zawierających odrębne funkcjonalności,
- Wsparcie dla istniejących protokół sieciowych https / ssh,
- Dostęp do darmowych, zdalnych repozytoriów,
- Każda rewizja jest odzwierciedleniem całego projektu,
- Ilość narzędzi oferujących czytelną nakładkę graficzną,
- Dostępność wiedzy,
- Darmowa licencja.

3.4. Flutter

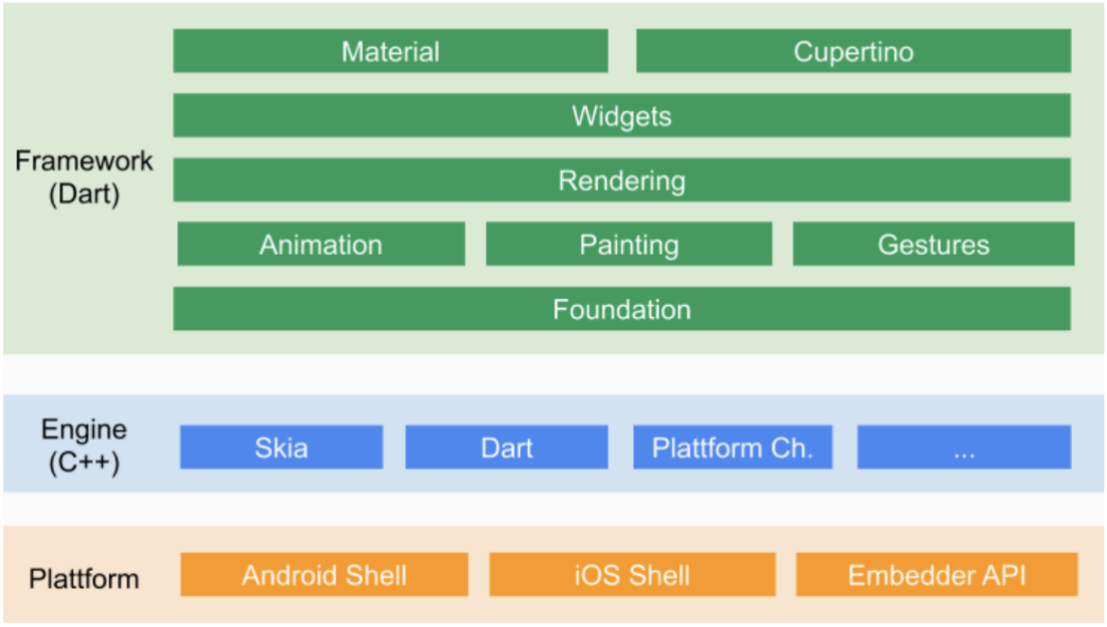
Flutter jest to zestaw narzędzi programistycznych przeznaczonych do budowania bardzo wydajnych oraz wysoko jakościowych aplikacji dla środowisk webowych, stacjonarnych (w trakcie pisania pracy możliwość dostępna tylko w wersji *preview*) jak i przenośnych za pośrednictwem wspólnej bazy kodowej [7]. Projekty tworzone są za pośrednictwem języka Dart, który strukturą bardzo przypomina język TypeScript.

Pierwsza jego stabilna wersja została opublikowana w Grudniu 2018 roku przez firmę Google [7]. Od tego czasu rozwiązanie przybiera na popularności. Zdalne, publiczne repozytorium Github projektu w czasie pisania pracy posiada ponad 91 tysięcy „polubień", co w porównaniu do jego konkurencji czyni go najpopularniejszym narzędziem programistycznym dedykowanym dla tworzenia wieloplatformowych aplikacji.

Kilka cech wyróżniających go spośród konkurencji:

1. Brak jawnie zdefiniowanych akcesorów dostępu takich jak *public* czy *private*. Dostępem do klasy czy pola kieruję jego nazwa. Nazwa bytu rozpoczynająca się od znaku „_" jest to równoznaczna z przypisaniem prywatnego akcesora dostępu.
2. Brak mechanizmu refleksji wpływa pozytywnie na wydajność rozwiązania. Efektem ubocznym tej cechy jest brak możliwości tworzenia zaawansowanych konstrukcji programistycznych podczas działania programu. Większość wzorców działających w konkurencyjnych rozwiązaniach jest za pośrednictwem refleksji jest realizowana w Flutterze poprzez generowanie kodu w trakcie jego pisania.
3. Mechanizm „Hot reload", umożlwiający wdrażanie zmian podczas tworzenia aplikacji na urządzenia docelowe bez konieczności ponownej kompilacji rozwiązania, co znaczne przyspiesza proces jej powstania.
4. Wszystko jest widgetem (ang. „everything is a widget"). Widget to podstawowa kontrolka interfejsu graficznego Fluttera. W prostym rozwiązaniu, dzieli się na dwie wersje: stanową (ang. *Statefule*) np. animacja oraz bezstanową (ang. *Statless*) np. tekst czy przycisk. Cały graficzny interfejs jest zbudowany hierarchicznie, każdy widget może posiadać zagnieżdżony, kolejny widget.

5. Język Dart pozwala na kompilację Fluttera AoT (ang. *Ahead of Time*) do natywnego kodu systemu kompatybilnego z Android NDK lub iOS LLVM. Rezultatem czego wszystkie komponenty graficzne są częścią technologii Flutter, a nie natywną funkcją systemu jak w przypadku rozwiązań konkurencyjnych. Pozwala to również na uzyskanie większej wydajności, gdyż w procesie nie uczestniczy maszyna interpretująca.



Rysunek 5. Architektura Fluttera

Wysoka wydajność technologii Flutter jest rezultatem braku interpretacji kodu podczas działania aplikacji między nią, a systemem operacyjnym jak ma to miejsce w niektórych konkurencyjnych rozwiązaniach. Przedstawiona na Rysunku 5 architektura ukazuję relację między wydajnym silnikiem C++, a warstwą aplikacji, która w bezpośredni sposób odwołuję się do warstwy NDK.

3.5. C#

C# jest to obiektowy język programowania zaprojektowany w latach 1998-2001 dla firmy Microsoft. Kompilowany jest do Common Intermediate Language (CIL), który jest kodem pośrednim, wykonywanym przez środowiska uruchomieniowe: .Net Framework, .Net Core , Mono lub DotGNU.

Pierwsza wersja Mono 1.0 ukazała się 30 stycznia 2004 roku. Umożliwiała kompilacje oraz uruchamianie programów napisanych językiem C# w systemach operacyjnych Linux, Windows czy macOS. Jednym z głównych założeń Mono jest otwarte środowisko oraz niezależne działanie od systemu operacyjnego. Projekt aktualnie jest rozwijany przez społeczność. Licencja MIT pozwala użytkownikom na tworzenie bloków kodu po uprzednim jego zatwierdzeniu do publicznego repozytorium GIT, dostępnego dzięki platformie github.com. Według oficjalnej dokumentacji projektu niektóre funkcję znane z środowiska .Net Framework mogą być niekompletne bądź działać inaczej niż w pierwowzorze [12]. Z technologii korzystają takie narzędzia jak Xamarin.Native czy Xamarin.Forms.

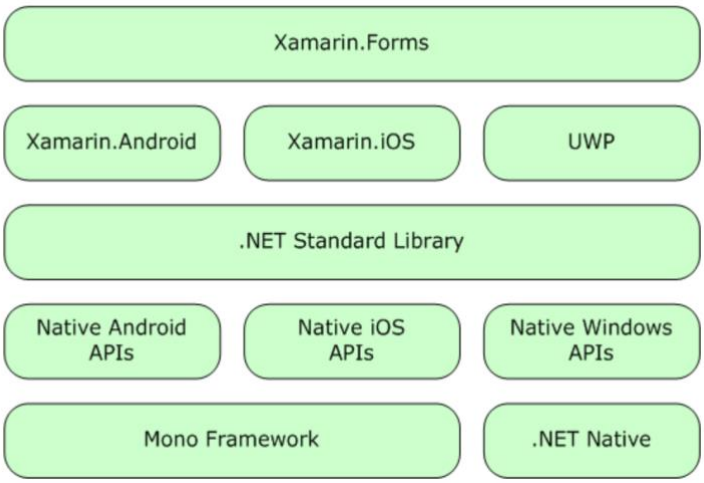
Jedna z cech C# wyróżniającą go na tle innych technologii jest LINQ (*Language Intregrated Query*). Jest to zestaw metod umożliwiający wykonywanie operacji filtrowania czy sortowania zbiorów danych w pamięci systemu operacyjnego przez intuicyjny dla użytkownika sposób.

3.5.1 Xamarin

Xamarin został stworzony przez firmę Xamarin znajdującą się w Stanach Zjednoczonych, stanie Kalifornia, mieście San Francisco. Założono ją w Maju 2011 roku, dzięki grupie osób pracującej nad rozwojem technologii Mono. Wraz z biegiem czasu firmą zainteresowała się korporacja – Microsoft, która w 2016 roku wykupiła całą firmę Xamarin, przypisując jej produktom licencje typu *Open Source* [21].

Xamarin jest podzielony na dwie technologię:

6. Xamarin.Native – jest to port natywnych bibliotek javy oraz swift w stosunku 1:1 dla technologii mono. Posiada on szersze możliwości implementacji wizualnych efektów kosztem możliwości dzielenia kodu tylko dla logiki biznesowej. Dostępna od początku projektu Xamarin.
7. Xamarin.Forms – powstała w 2014 roku technologia umożliwiająca dzielenie kodu interfejsu użytkownika oraz logiki biznesowej między obsługiwanymi platformami. Warstwa graficzna jest tworzona za pośrednictwem kodu XAML, lecz jej możliwości w stosunku do podejścia Xamarin.Native są mocno ograniczone. Tworzenie zaawansowanego interfejsu graficznego za pomocą tego podejścia jest bardziej skomplikowane. W aplikacji mobilnej będącej rezultatem analizy pracy, dzięki zastosowaniu technologii Xamarin.Forms udało się pokryć 99% dzielenia kodu między systemem Android oraz iOS. Architektura Xamarin.Forms została przedstawiona na Rysunku 6.



Rysunek 6. Architektura Xamarin.Forms

3.5.2 .Net Core

.Net Core jest ogólnie dostępną platformą programistyczną *Open Source* umożliwiającą tworzenie programów przeznaczonych dla systemów operacyjnych jak Windows, Linux czy macOS. Zapewnia wsparcie dla nowoczesnych paradygmatów tworzenia oprogramowania jak programowanie asynchronicznie czy kompatybilność dla kontenerów. Pierwsza wersja została opublikowana 27 Lipca 2016 roku [9].

Asp.net Core jest technologią platformy .Net Core przeznaczoną do tworzenia aplikacji serwerowych odpowiadającą za przetwarzanie żądań http / https / amqp i innych. Jest kompilowana do jednego piku wykonawczego obsługiwanego przez .Net Core Runtime na serwerach Kestrel czy IIS.

Zaletami platformy Asp.Net Core są:

1. Wydajność potwierdzona przez wiele niezależnych benchmarków, m.in. Age of Ascent,
2. Ilość narzędzi w postaci bibliotek programistycznych,
3. Ilość materiałów edukacyjnych dostępnych w sieci,
4. Wieloplatformowość,
5. Mechanizm refleksji pozwalający na budowę zaawansowanych konstrukcji programistycznych w trakcie działania programu,
6. Licencja typu *Open Source*,
7. Rozbudowana społeczność.

3.5.3 .Net Standard

.Net Standard jest formalną specyfikacją środowiska .Net dostępną na wszystkie jej implementację. Jej motywacją jest stworzenie większej jednolitości w ekosystemie .Net. Zapewnia szerszy zestaw bibliotek niż specyfikacja ECMA 335 [9].

Większość bibliotek programistycznych dostępnych na platformę NuGet jest aktualnie tworzona w technologii .Net Standard. Dzięki temu mogą być one konsumowane niezależnie od wybranego narzędzia ekosystemu .Net (Xamarin, .Net Framework, .Net Core).

3.6. IDE

IDE czyli „Integrated Development Enviroment” określa zintegrowane środowisko programistyczne. Jest to pakiet programów niezbędnych do tworzenia oraz rozwoju aplikacji, który często dostarcza szybki dostęp do ważnych, często używanych funkcji oraz automatyzuję powtarzalne wykonane zadania. Zintegrowane środowisko programistyczne zawiera w zależności od zakresu użycia kilka komponentów, takich jak np. edytor tekstowy, kompilator czy debbuger.

3.6.1 Visual Studio Code

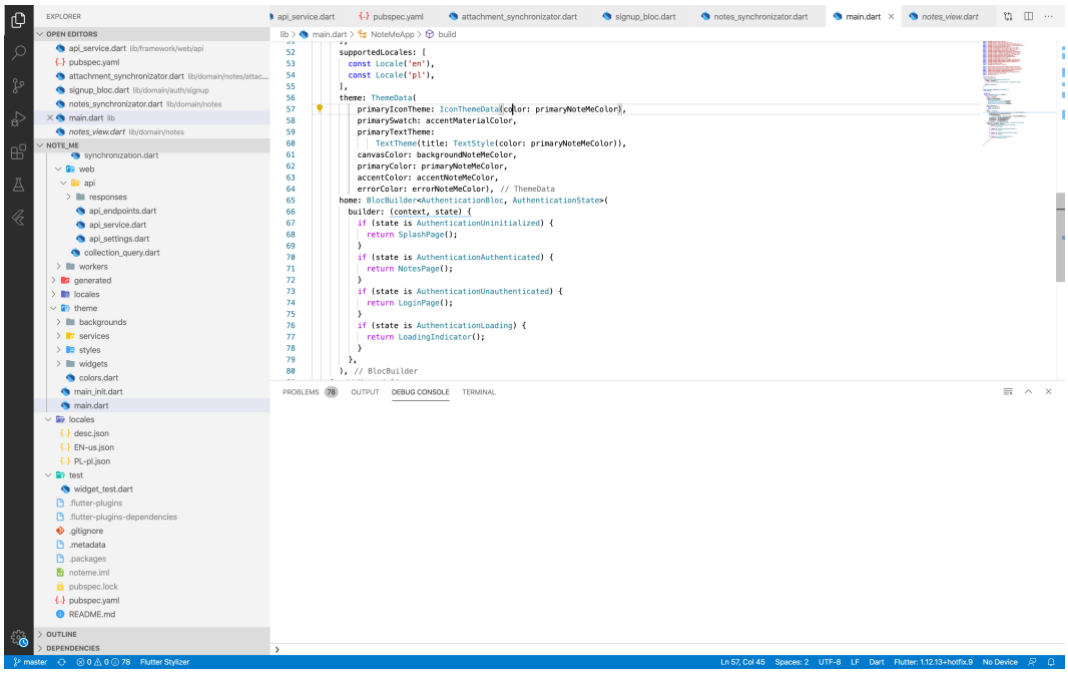
Visual Studio Code jest darmowym, stacjonarnym edytorem kodów źródłowych dla wielu środowisk stworzonych przez firmę Microsoft dystrybuowanym na licencji MIT o otwartym kodzie źródłowym. Posiada wsparcie dla debugowania kodu, kolorowania składni, wersjonowania za pomocą wbudowanej nakładki dla narzędzia GIT oraz działa na najpopularniejszych stacjonarnych systemach operacyjnych Windows, macOS czy Linux.

Zaletami środowiska Visual Studio Code są:

1. Darmowa licencja,

2. Wsparcie dla wielu systemów operacyjnych,
3. Rozbudowany Market oferujący system rozszerzeń,
4. Duża społeczność,
5. Stabilność,
6. Wydajność.

Dzięki bardzo rozbudowanej części marketu wyposażonej w dużą ilość darmowych wtyczek, Visual Studio Code posiada wsparcie dla technologii Flutter. Rysunek 7 przedstawia przykładowe użycie IDE w środowisku Flutter.

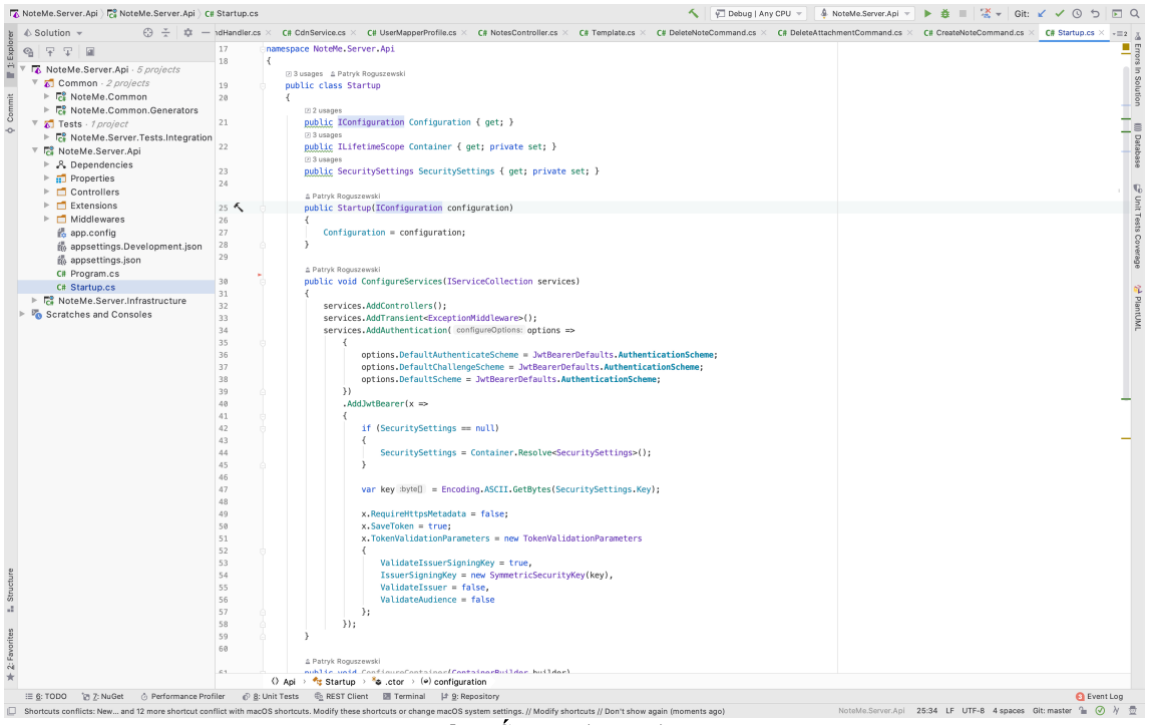


Rysunek 7. Środowisko Visual Studio Code

3.6.2 Rider

Rider jest komercyjnym środowiskiem firmy JetBrains oferującym wsparcie ekosystemu .Net dla takich systemów operacyjnych Linux, macOS czy Windows. Mimo płatnej licencji zdobył on uznanie wśród społeczności programistycznej, gdyż cechuję się szerszym wachlarzem możliwości, większą wydajnością oraz funkcjami umożliwiającymi szybszą implementację kodu. Rysunek 8 przedstawia przykładowe użycie IDE podczas jego użycia.

Posiada wbudowany dodatek ReSharper wspierający programistów podczas tworzenia kodu czyniąc ten proces krótszym oraz wskazuje miejsce które można zmodyfikować w celu zyskania wydajności.



Rysunek 8. Środowisko Rider

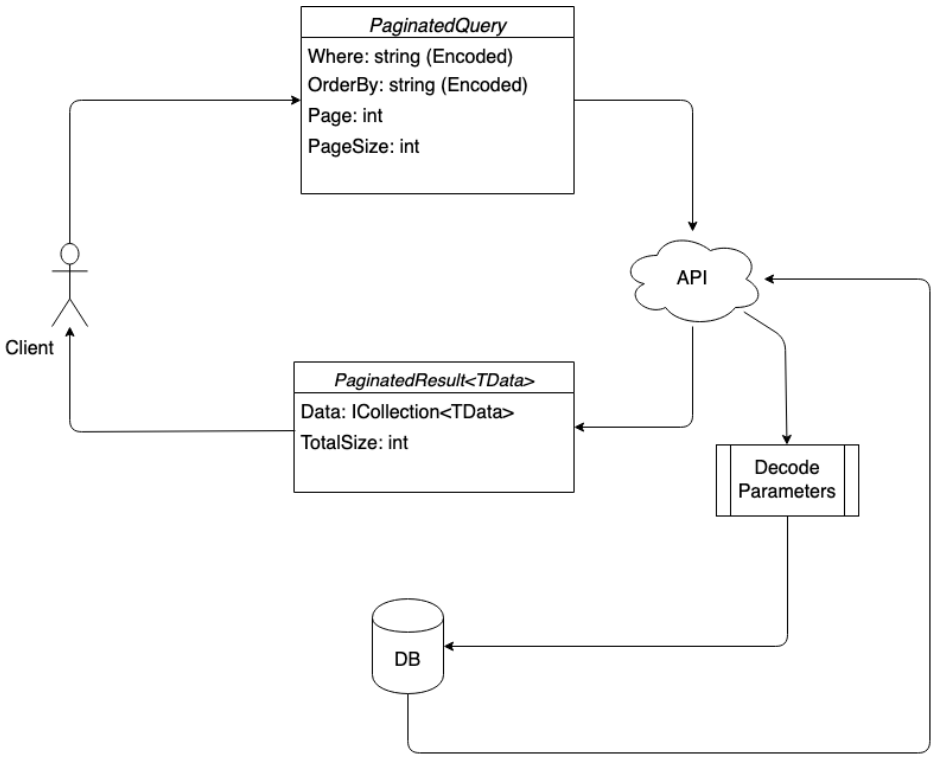
4. Propozycja nowego wzorca projektowego Paginator

Jednym z pobocznych rezultatów przeprowadzanej analizy jest koncepcja wzorca projektowego Paginator, zaimplantowanego w trzech częściach rozwiązania.

4.1. Koncepcja

Paginator rozwiązuje problem przesyłania nadmiernej ilości danych między częścią kliencką, a serwerową systemu za pośrednictwem wymiany odpowiednich kontraktów. Określa on standard komunikacyjny dotyczącej transportu kolekcji danych.

Większość konkurencyjnych rozwiązań jest oparta o przekazywanie jawnych parametrów bądź kodu SQL bezpośrednio wykonywanego na bazie danych. Powszechnym zjawiskiem jest tworzenie własnego silnika stronicowania oraz transformacji danych w każdym nowym rozwiązaniu. Brak standaryzacji tej kwestii wydłuża czas wdrażania nowych osób do projektu oraz niesie z sobą ryzyko wystąpienia błędów takich jak filtrowanie za pomocą użycia znaków specjalnych.



Rysunek 9. Schemat wzorca Paginator

Rysunek 9 obrazuje działanie mechanizmu filtrowania, sortowania oraz paginacji pobieranych danych. Parametry *Where* oraz *OrderBy* podlegają szyfrowaniu, rezultatem czego wzrasta poziom bezpieczeństwa podczas przesyłania kontraktu oraz wzrastają jego możliwości przekazywania znaków specjalnych.

Odpowiedzialności parametrów kontraktu PaginatedQuery:

1. *Where* – Filtrowanie danych,
2. *OrderBy* – Sortowanie danych, powinna uwzględniać ich kierunek (*desc*, *asc*),
3. *Page* – Numer strony,
4. *PageSize* – Pożądany rozmiar kontraktu danych.

Wymagania dotyczące użycia wzorca:

1. Struktura klient – serwer,
2. Możliwość wymiany kontraktu między klientem, a serwerem,
3. Wspólny sposób szyfrowania parametrów dla części klient oraz serwer,
4. Warstwa trwałości danych.

Czynności wykonywane za pomocą wzorca:

1. Implementacja wspólnego sposób szyfrowania parametrów dla struktury klient – serwer,
2. Klient tworzy kontrakt z zaszyfrowanymi polami *Where* oraz *OrderBy*. Następnie wypełnia pola dotyczące ilości danych (*PageSize)* wraz z ich lokalizacją (*Page),*
3. Klient przekazuje utworzony kontrakt do części serwerowej,
4. Część serwerowa przyjmuje kliencki kontrakt,
5. Kontrakt przekazywany jest do mechanizmu deszyfrującego,
6. Odszyfrowane parametry przekazane są do warstwy trwałości danych w celu pobrania kolekcji,
7. Przekształcona przez parametry kolekcja przekazywana jest z warstwy trwałości do api wraz z całkowitą ilością dostępnych danych (*TotalSize)* według parametru *Where*,
8. Api tworzy kontrakt przypisując przekształconą kolekcję do parametru *Data* oraz liczbę całkowicie dostępnych danych do parametru *TotalSize*,
9. Api zwraca kontrakt do klienta.

Potencjalne zagrożenia:

1. Parametr *Where* przed wysłaniem do bazy danych powinien zostać sprawdzony przed podatnością typu *Sql Injection*,
2. Parametr PageSize powinien posiadać maksymalny limit chroniący przed pobraniem zbyt dużej ilości danych.

Zalety użycia wzorca:

1. Ograniczenie ilości rozmiaru danych przesyłanych między strukturą klient / serwer,
2. Ograniczenie zużycia pamięci serwerowej spowodowanej dużą ilością danych,
3. Możliwość umieszczania w kontrakcie znaków specjalnych,
4. Standaryzacja procesu pobierania kolekcji przez część kliencką,
5. Łatwy poziom implementacji.

4.2. Przypadek użycia

Wzorzec Paginator może zostać użyty w sytuacji, gdy użytkownik systemu zdecyduję się pobrać z serwera tylko 50 ostatnich rekordów danych stworzonych przed 2020 rokiem. Założeniem przypadku jest komunikacja między klientem, a serwerem w postaci https, za pośrednictwem *Query String* oraz *JSON*. Aby zrealizować przypadek klient musi wykonać następującą listę czynności:

1. Klient wybiera BASE64 jako algorytm szyfrowania w częściach klienckich oraz serwerowych,
2. Szyfruje pola oraz przypisuje im wartości:
    a. *OrderBy* – CreatedAt_desc - Q3JlYXRlZEF0X2Rlc2M=,
    b. *Where* – CreatedAt > ”01.01.2020” - Q3JlYXRlZEF0IDwgIjAxLjAxLjIwMjAi,
    c. *PageSize* – 50,
    d. *Page* – 0.
3. Wysyła kontrakt protokołem https do serwera, w postaci *Query String*,
4. Serwer odbiera kontrakt,
5. Serwer przekształca parametry zaszyfrowane:
    a. *OrderBy* - Q3JlYXRlZEF0X2Rlc2M= - CreatedAt_desc,
    b. *Where* - Q3JlYXRlZEF0IDwgIjAxLjAxLjIwMjAi - CreatedAt > ”01.01.2020”.
6. Przekuje parametry do bazy danych odbierając:
    a. Przekształconą kolekcję,
    b. Liczbę dostępnych danych podlegających przekształceniu parametru Where.
7. Wypełnia pola kontraktu zwrotnego:
    a. Data – przypisuje przekształconą kolekcję,
    b. TotalSize – przypisuję liczbę dostępnych danych.
8. Zwraca wypełniony kontakt w postaci obiektu JSON.

4.3. Implementacja

Przykładowej implantacji wzorca w projekcie dokonano za pomocą technologii .Net oraz Flutter. W tym rozdziale przedstawiono kod źródłowy rozwiązania części klienckiej (Xamarin), serwerowej (.Net Core) oraz wspólnej (.Net Standard).

W części wspólnej umieszczono bazowy kontrakt przedstawiony na Kodzie 1 wraz metody rozszerzające umożliwiające adaptację parametrów przekształcających do formy LINQ.

Kod 1. Bazowa klasa kontraktu Query dla wzorca Paginator.

```
public abstract class EncodedPaginationQueryBase : IPaginationQuery
{
    private string _where = "";
```

```
private string _orderBy = "";

public virtual string Where
{
    get => _where;
    set => _where = Base64UrlEncoder.Decode(value);
}

public virtual string OrderBy
{
    get => _orderBy;
    set => _orderBy = Base64UrlEncoder.Decode(value);
}

public virtual int Page { get; set; }
public virtual int PageSize { get; set; } = PaginationSettings.DefaultPageSize;

public EncodedPaginationQueryBase(
    string where = null,
    string orderBy = null,
    int page = default,
    int pageSize = default)
{
    _where = where;
    _orderBy = orderBy;

    Page = page;
    PageSize = pageSize;
}

public virtual string ToUri()
{
    var sb = new StringBuilder("?");

    if (!string.IsNullOrEmpty(_where))
        sb.Append($"{nameof(Where)}={Base64UrlEncoder.Encode(_where)}&");

    if (!string.IsNullOrEmpty(_orderBy))
        sb.Append($"{nameof(OrderBy)}={Base64UrlEncoder.Encode(_orderBy)}&");

    if (Page != default(int))
        sb.Append($"{nameof(Page)}={Page}&");

    if (PageSize != default(int))
        sb.Append($"{nameof(PageSize)}={PageSize}&");

    sb.Remove(sb.Length - 1, 1);

    return sb.ToString();
}

public virtual EncodedPaginationQueryBase SetNormalOrderBy(string value)
{
    this._orderBy = value;
    return this;
}

public virtual EncodedPaginationQueryBase SetNormalWhere(string value)
{
    this._where = value;
    return this;
}
```

Za pomocą mechanizmu dziedziczenia każda klasa dziedzicząca po bazowym kontrakcie posiada wbudowaną możliwość szyfrowania parametrów podczas przekształcania modelu do formy http *Query String* dzięki której można przekazać parametry do warstwy serwerowej. Przykładowa postać wygenerowanego zapytania została przedstawiona w Kodzie 2. Dzięki deszyfrowaniu parametru w getterze klasy, mechanizm bindowania automatycznie odszyfrowuje informację z parametru *Query String* do odpowiedniej klasy. Kontrakt bazowy znajduję się w wspólnej części .Net Standard.

Kod 2. Przykładowa postać wygenerowanego zapytania o dane

```
https://some.api/news?Where=Q3JlYXRlZEF0IDwgIjAxLjAxLjIwMjAi&OrderBy
=Q3JlYXRlZEF0X2Rlc2M=&Page=0&PageSize
```

Kod 3. Konsumpcja kontraktu Query przez API

```
[HttpGet]
public async Task<IActionResult> GetAllAsync([FromQuery] GetNotesQuery query)
{
    var notes = await DispatchQueryAsync<GetNotesQuery,
PaginationDto<NoteDto>>(query);
    return Ok(notes);
}
```

Kod 3 przedstawia poprawny odbiór kontraktu przez część serwerową, który zostaje przesłany do odpowiednego handlera. Kod 4 ukazuję proces adaptacji parametrów kontraktu do postaci LINQ, tak by Kod 5 mógł wykonać operację obsłużenia na kontekście bazodanowym technologii ORM EntityFramework.

Kod 4. Adaptacja parametrów kontaktu Query do postaci LINQ

```
public static class QueryableExtension
{
    public static IQueryable<TModel> FilterBy<TModel>(this IQueryable<TModel>
queryable,
        IPaginationQuery paginationQuery)
    {
        return queryable.Where(paginationQuery)
                  .TransformBy(paginationQuery);
    }

    public static IQueryable<TModel> Where<TModel>(this IQueryable<TModel>
queryable,
        IPaginationQuery paginationQuery)
    {
        if (!string.IsNullOrEmpty(paginationQuery.Where))
        {
            queryable = WhereExpressionFactory.CreateFromString(queryable,
paginationQuery.Where);
        }

        return queryable;
    }

    public static IQueryable<TModel> TransformBy<TModel>(this IQueryable<TModel>
queryable,
        IPaginationQuery paginationQuery)
    {
        var pageSize = paginationQuery.PageSize > PaginationSettings.MaxPageSize
            ? PaginationSettings.MaxPageSize
```

```
            : paginationQuery.PageSize;

        if (!string.IsNullOrEmpty(paginationQuery.OrderBy))
        {
            queryable = SortExpressionFactory.SortBy(queryable,
paginationQuery.OrderBy);
        }

        queryable = queryable
            .Skip(pageSize * paginationQuery.Page)
            .Take(pageSize);

        return queryable;
    }
}
```

Kod 5. Obsługa kontraktu oraz stworzenie odpowiedzi

```
public async Task<PaginationDto<NoteDto>> HandleAsync(GetNotesQuery query)
{
    var queryable = _context.Notes
        .Where(x => x.UserId == query.RequestBy)
        .Where(query);

    var count = await queryable.CountAsync();
    var result = await queryable.FilterBy(query).ToListAsync();
    var pagination = new PaginationDto<Note>
    {
        Data = result,
        TotalSize = count
    };

    return _mapper.Map<PaginationDto<NoteDto>>(pagination);
}
```

5. Prototyp aplikacji

Na potrzeby wykonania poprawnej analizy wpływu wzorców projektowych na tworzenie wieloplatformowych aplikacji mobilnych, stworzono prototyp umożliwiający zapisywanie danych w postaci notatek tekstowych wraz z wykorzystaniem najpopularniejszych wzorców zależnych od wykorzystanej technologii.

5.1. Wymagania biznesowe

Mobilna aplikacja NoteMe! Ma zapewniać użytkownikowi swobodne zapisywanie informacji w formie cyfrowych notatek, przechowywanych na zdalnym serwerze.

1. Użytkownik musi mieć autoryzowany dostęp do swoich notatek, niezależny od aktywowanego urządzenia,
2. Każda notatka powinna mieć treść, datę stworzenia, lokalizację, tagi oraz tytuł,
3. Notatka może, lecz nie musi zawierać załącznik w formie zdjęcia,
4. Użytkownik powinien móc założyć własne konto, wymaganymi danymi dla każdego konta są email oraz hasło. Hasło powinno być trzymane w bezpiecznej postaci na zdalnych serwerach,
5. Każdy użytkownik może tworzyć wiele notatek,
6. Notatki powinny być dostępne tylko dla jego właściciela.

5.2. Wymagania niefunkcjonalne

Aplikacja NoteMe! spełnia poniższe wymagania niefunkcjonalne:

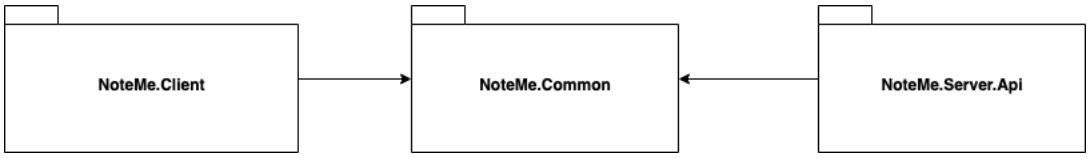
1. Dostępność 24h / 7 dni w tygodniu,
2. Dwujęzyczność w językach Angielskim oraz Polskim,
3. Dostęp oraz spełnienie wymagań biznesowych w trybie offline,
4. Synchronizacja danych po przywróceniu połączenia z Internetem oraz automatycznym przejściu do trybu online,
5. Zalogowany użytkownik ma dostęp do aplikacji do momentu ręcznego wylogowania lub czasu wygaśnięcia jego sesji zalogowania,
6. Aplikacja ma dostęp do aparatu oraz lokalizacji urządzenia,
7. Strona główną w zależności od ważności sesji zalogowania jest:
    a. Brak aktywnej sesji – ekran logowania z możliwością przejścia do ekranu rejestracji,
    b. Aktywna sesja – ekran stosu notatek, posortowanych według ich daty stworzenia wraz z przejściem do ekranu edycji / stworzenia notatki.

5.3. Architektura systemu

Prototyp aplikacji został stworzony w czterech różnych technologiach (Xamarin, Asp.net Core, .Net Standard oraz Xamarin) w architekturze Klient – Serwer.

W części C# rozwiązanie zostało podzielone na trzy solucje:

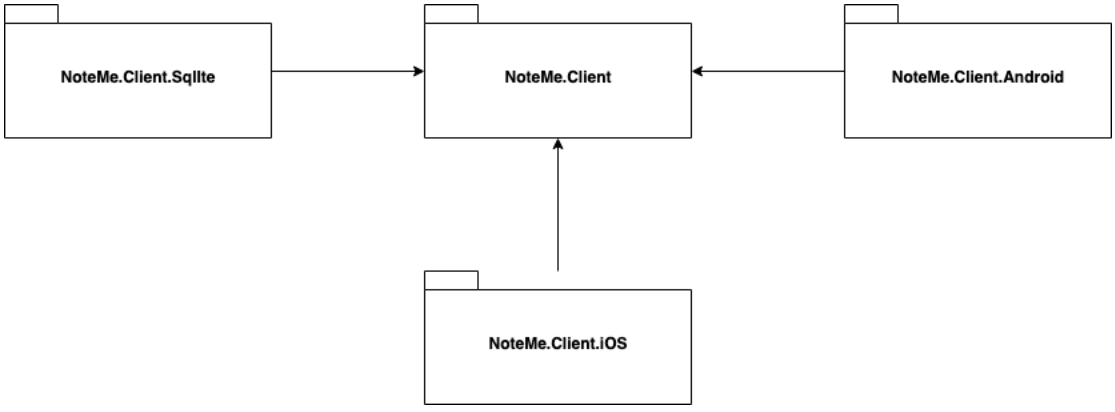
1. `NoteMe.Common`
2. `NoteMe.Client`
3. `NoteMe.Server.Api`



Rysunek 10. Schemat zależności technologii C#.

Według przedstawionego na Rysunku. 10 schematu zależności solucje `NoteMe.Client` (Xamarin) oraz `NoteMe.Server.Api` (Asp.net Core) korzystają z klas oraz metod zawartych w projekcie `NoteMe.Common` (.Net Standard). Rezultatem tego jest brak powtórzonej implementacji (zasada DRY ang. *Don't Repeat Yourself*) między strukturą klient – serwer. `NoteMe.Common` nie zawiera referencji do zewnętrznych bibliotek NuGet.

`NoteMe.Client` zawiera implementację interfejsu użytkownika oraz obsługi dla funkcji natywnych urządzenia jak dostępu do kamery czy pobranie aktualnej lokalizacji.



Rysunek 11. Schemat zależności projektu klienckiego C#.

Projekt kliencki w technologii C# zawiera cztery solucje:

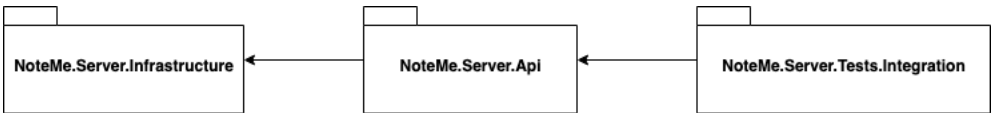
1. `NoteMe.Client` – projekt wspólny zawierający kod dla logiki biznesowej, interfejsu użytkownika oraz dostępu do natywnych funkcji systemu operacyjnego. Zestaw zewnętrznych bibliotek programistycznych NuGet użytych w rozwiązaniu został przedstawiony w Tabeli 1.

2. `NoteMe.Client.Sqlite` – środowisko uruchomieniowe dla lokalnej warstwy trwałości danych w postaci bazy Sqlite umożliwiającej zarządzenie strukturą poprzez wykonywanie migracji kodu C# dzięki technologii Entity Framework.
3. `NoteMe.Client.iOS` – punkt uruchomieniowy dla systemu operacyjnego iOS. Zawiera również definicję listy uprawnień oraz kod inicjalizujący lokalną bazę danych.
4. `NoteMe.Client.Android` – punkt uruchomieniowy dla systemu operacyjnego Android. Zawiera również definicję listy uprawnień oraz kod inicjalizujący lokalną bazę danych.

Tabela 1. Zestaw zewnętrznych bibliotek NuGet projektu NoteMe.Client.

Nazwa	Opis	Wersja
`Autofac`	Biblioteka umożliwiająca wstrzykiwanie zależności między powstałymi klasami.	4.9.4
`FluentValidation`	Biblioteka realizująca tworzenie reguł walidacyjnych wraz z możliwością ich weryfikacji.	8.5.1
`Automapper`	Mapper umożliwiający przekształcanie obiektów domenowych typu POCO.	9.0.0
`Plugin.Multilingual`	Biblioteka umożliwiająca implantację wielojęzyczności w kontekście aplikacji mobilnych.	1.0.2
`Xamarin.Essentials`	Warstwa abstrakcji dla natywnych funkcji mobilnych systemów operacyjnych takich jak dostęp do aparatu czy geolokalizacji.	1.2.0
`Npgsql.EntityFrameworkCore`	ORM firmy Microsoft służący za abstrakcję dla warstwy trwałości - Sqlite.	2.1.0

`NoteMe.Server.Api` zawiera implementację dostępu do głównej warstwy trwałości – bazy danych PostgreSql wraz z logiką biznesową pozwalająca na odpowiednie przekształcenie informacji przez ich utrwaleniem.



Rysunek 12. Schemat zależności projektu serwerowego C#.

Projekt serwerowy w technologii C# zawiera trzy solucje:

1. `NoteMe.Server.Infrastructure` – projekt zawierający logikę biznesową oraz implementację warstwy trwałości danych. Zestaw zewnętrznych bibliotek programistycznych NuGet użytych w rozwiązaniu został przedstawiony w Tabeli 2.
2. `NoteMe.Server.Api` – punkt uruchomieniowy API oraz warstwa aplikacji odpowiedzialna za kontakt z częścią kliencką. Obsługuje żądania http / https.

3. `NoteMe.Server.Tests.Integrations` – zestaw testów typu E2E (ang. *End to end*) umożlwiający uruchomienie instancji części serwerowej w pamięci systemu operacyjnego z możliwością tworzenia żądań sieciowych za pomocą kodu C#.

Tabela 2. Zestaw zewnętrznych bibliotek NuGet projektu NoteMe.Server.Api.

Nazwa	Opis	Wersja
Autofac	Biblioteka umożliwiająca wstrzykiwanie zależności między powstałymi klasami.	4.9.4
Automapper	Mapper umożlwiający przekształcanie obiektów domenowych typu POCO.	9.0.0
Npgsql.EntityFrameworkCore	ORM firmy Microsoft służący za abstrakcję dla warstwy trwałości - PostgreSQL.	3.0.1

Pozostała część kliencka została zaimplantowana w technologii firmy Google – Flutter. Posiada dwa punkty uruchomieniowe zależne od systemu operacyjnego (Android / iOS). Tabela 3. przedstawia zestaw zewnętrznych, użytych bibliotek programistycznych pub.dev.

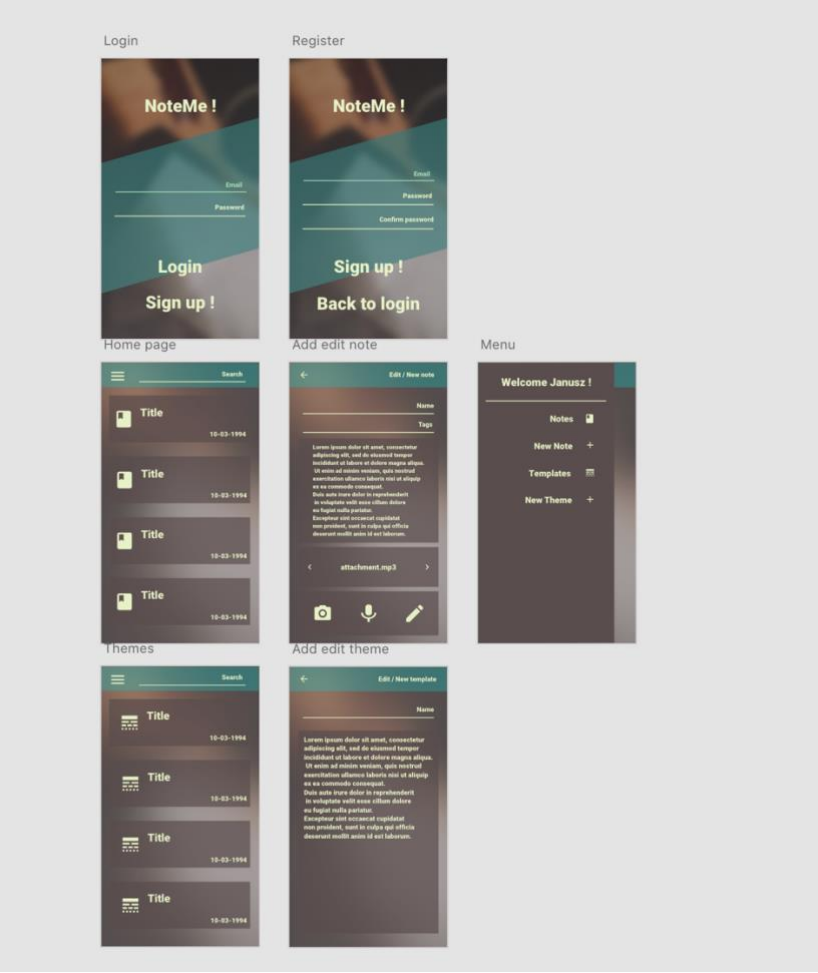
Tabela 3. Zestaw zewnętrznych bibliotek programistycznych pub.dev dla projektu klienckiej w technologii Flutter.

Nazwa	Opis	Wersja
get_it	Biblioteka umożliwiająca wstrzykiwanie zależności między powstałymi klasami poprzez generowanie kodu za pomocą adnotacji.	3.1.0
http	Zestaw narzędzi pozwalających na wykonywanie zapytań http / https.	0.12.0+4
json_annotation	Generator umożlwiający tworzenie kodu dla klas będących modelem dla kontraktu danych między klientem, a serwerem za pomocą adnotacji.	3.0.1
shared_preferences	Abstrakcja dla warstwy trwałości danych o małym rozmiarze.	0.5.6+3
fluttertoast	Narzędzie umożliwiające wyświetlanie niewielkich powiadomień.	4.0.0
path	Biblioteka umożliwiająca operację dotyczących ścieżek lokalizacji pliku.	1.6.4
sqflite	Sterownik bazy danych Sqlite.	1.3.0
flutter_bloc	Narzędzie wspomagające implantację wzorca projektowego BLOC.	3.2.0
bloc	Narzędzie wspomagające implantację wzorca projektowego BLOC.	3.0.0
uuid	Generator guid.	2.0.4

equatable	Warstwa abstrakcji umożliwiającą nadpisanie implantacji dla operacji porównania klas.	1.1.0
intl	Biblioteka umożliwiająca implantację wielojęzyczności w kontekście aplikacji mobilnych.	0.16.0
geolocator	Wsparcie dla pobierania aktualnej geolokalizacji użytkownika.	5.3.1
image_picker	Biblioteka rozszerzająca możliwości używania aparatu.	0.6.5+2

5.4. Projektowanie rozwiązania

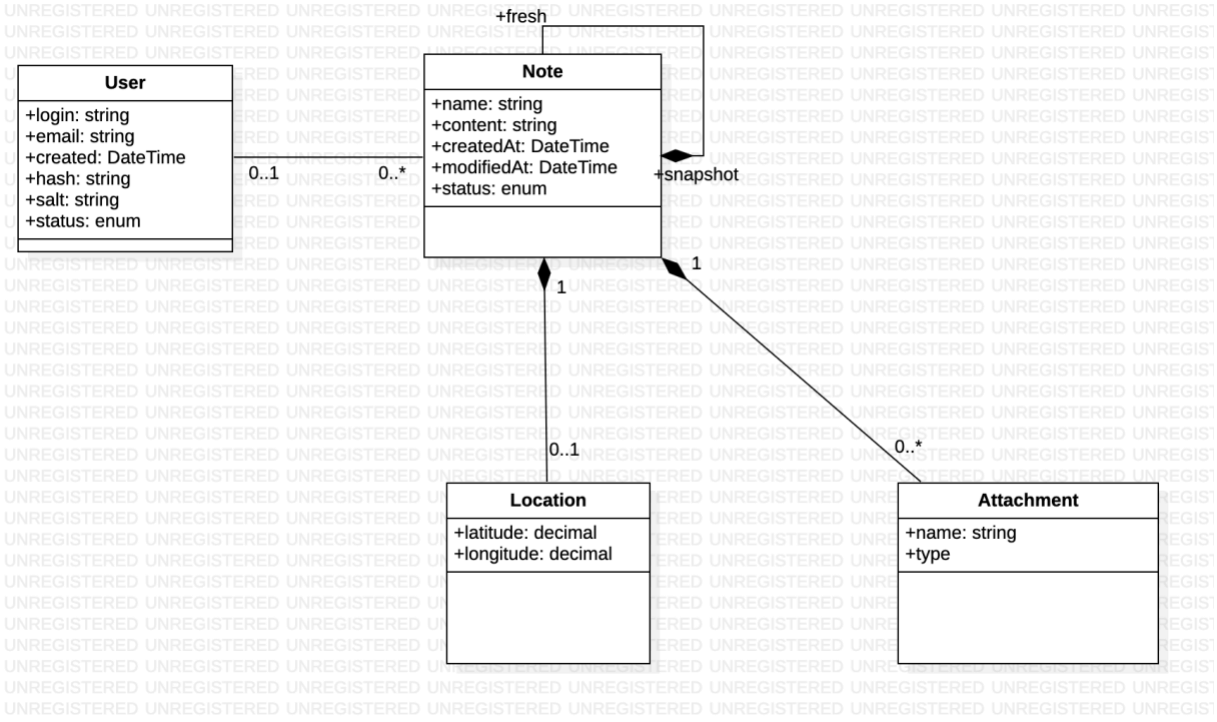
Rozwiązanie zostało zaprojektowane za pomocą szkiców mockup oraz diagramów klas UML, będących wizualizacją dla zależności biznesowych.



Rysunek 13. Mockupy prototypu aplikacji

Mockupy przedstawione na Rysunku 13. nie zostały w całości zaimplementowane przez prototyp aplikacji. Ograniczenia technologicznie nie pozwoliły na realizację zielonych podświetleń kontrolek tekstowych czy nadaniu przezroczystości dla górnego paska akcji. Pozycje „Templates” oraz „New Theme” również zostały pominięte, gdyż nie miały wpływu na wynik analizy pracy.

Przedstawiony poniżej diagram klas UML wizualizuję relację zależności klas domenowych zaimplementowanych przez prototyp aplikacji.



Rysunek 14. Diagram UML dla prototypu aplikacji.

6. Analiza wykorzystanych wzorców projektowych

W stworzonym prototypie aplikacji zaimplementowano 8 różnych wzorców projektowych utrzymując standard reguł SOLID. Poniższy rozdział opisuje wpływ zastosowanych praktyk wraz z przypadkami ich użycia na działanie stworzonego prototypu aplikacji.

6.1. SOLID

Reguły SOLID zawierają wskazówki o tym jak rozmieszczać funkcje oraz struktury danych w klasach wraz definicją zachodzących między nimi relacji. Słowo „klasa" nie obejmuję jedynie tworzenia oprogramowania wyłącznie w technologiach obiektowych. Jest ono tylko metodą grupowania funkcji i danych, niezależnie od potocznie nadanej im nazwy. Reguły SOLID odnoszą się do dowolnej metody programowania [11].

S – SRP (ang. *Single Resposibility Principle*) – reguła jednej odpowiedzialności. Nawet na najlepszą strukturę systemu oprogramowania intensywnie wpływa społeczna struktura używającej go organizacji, dlatego każdy moduł produktu powinien mieć tylko i wyłącznie jeden powód do zmiany [11]. Kod 6 przedstawia zastosowanie reguły SRP na przykładzie konfiguracji encji bazodanowej. Odpowiedzialnością klasy `AttachmentEntityConfiguration` jest tylko i wyłącznie określenie docelowego schematu tabeli załączników dla notatek. Potencjalny powód modyfikacji zawartości klasy może być związany tylko i wyłącznie z potrzebą zmiany struktury warstwy trwałości tabeli `Attachemts`, rezultatem czego zmiana nie wpływa na inne konteksty domenowe.

Kod 6. Przykład użycia SRP w projekcie.

```
public class AttachmentEntityConfiguration : IEntityTypeConfiguration<Attachment>
{
    public void Configure(EntityTypeBuilder<Attachment> builder)
    {
        builder.HasKey(x => x.Id);

        builder.HasOne(x => x.Note)
            .WithMany(x => x.Attachments)
            .HasForeignKey(x => x.NoteId);
    }
}
```

O – OCP (ang. *Open-Closed Principle*) – reguła otwarty – zamknięty. Oprogramowanie ma pozwalać na łatwe wprowadzanie zmian poprzez dodawanie nowego kodu, a nie jego modyfikację [11]. Kod 7 przedstawia zasadę OCP na przykładzie implementacji nowych operacji wykonywanych po stronie serwerowej części systemu. By zaimplementować nową operację dotyczącą kontekstu autoryzacji należy stworzyć model typu `ICommandProvider` oraz implementację jego handlera `ICommandHandler`. Cały kod dotyczący logiki biznesowej powinien znaleźć się w ciele metody `HandleAsync`. Rezultatem wprowadzenia tego standardu jest brak możliwości ingerencji w istniejący kod innej logiki biznesowej.

Kod 7. Przykład użycia OCP w projekcie.

```
public interface ICommandHandler<TCommand>
    where TCommand : ICommandProvider
{
    Task HandleAsync(TCommand command);
}
```

```
public class AuthCommandHandler : ICommandHandler<LoginCommand>
{
    public async Task HandleAsync(LoginCommand command)
    {
        var user = await _context.Users.FirstOrDefaultAsync(x => x.Email.ToLower()
== command.Email.ToLower());
        if (user == null)
        {
            throw new ServerException(ErrorCodes.InvalidCredentials);
        }

        var hash = _securityService.GetHash(command.Password, user.Salt);
        if (hash != user.Hash)
        {
            throw new ServerException(ErrorCodes.InvalidCredentials);
        }

        var dto = _mapper.Map<UserDto>(user);
        var jwt = _securityService.GetJwt(dto, command.Id);

        _cacheService.Set(jwt);
    }
}
```

L - LSP (ang. *Liskov Substitution Principle*) – reguła podstawiania. Chcąc zbudować system z wymiennych części, trzeba sprawić, by dostosowały się one do kontaktu, który w przyszłości pozwoli je na zastąpienie innymi [11]. Kod 8 przedstawia użycie zasady LSP na przykładzie synchronizacji danych. Interfejs `ISynchronizationHandler` informuję o wymaganej metodzie `HandleAsync` która jest odpowiedzialna za przeprowadzenie synchronizacji danych o odpowiednim typie. Jego implementację nie wymagają dodatkowych argumentów niezdefiniowanych w interfejsie. Wymiana części kodu odbywa się poprzez implementację kolejnego interfejsu.

Kod 8. Przykład użycia LSP w projekcie.

```
public interface ISynchronizationHandler<TEntity> : ISynchronizationHandler
    where TEntity : IIdProvider, ISynchronizationProvider
{
    Task HandleAsync(Synchronization synchronization, NoteMeContext context,
CancellationToken cts);
}

public class NoteSynchronizationHandler : ISynchronizationHandler<Note>
{
    public async Task HandleAsync(Domain.Synchronization.Synchronization
synchronization, NoteMeContext context, CancellationToken cts)
    {
        await UpdateAllNotesAsync(context, cts);
        await FetchAllResultsAsync(synchronization, context, cts);
        await SendAllNotesAsync(context, cts);
    }
}


public class AttachmentSynchronizationHandler : ISynchronizationHandler<Attachment>
{
    public async Task HandleAsync(
        Domain.Synchronization.Synchronization synchronization,
        NoteMeContext context,
        CancellationToken cts)
    {
```

```
            await FetchAllAttachmentsAsync(synchronization, context, cts);
            await DownloadAllAttachmentsAsync(context, cts);
            await CreateAllAttachmentsAsync(synchronization, context, cts);
            await UploadAllAttachmentsAsync(context, cts);
        }
}
```

I – ISP (ang. *Interface .Segregation Principle*) – reguła podziału interfesjów. Ta reguła nakazuję programistom by unikali tworzenia zależności od niepotrzebnych elementów [11]. Każdy interfejs jest oddzielony atomowo, rezultatem czego posiada najmniejszą możliwą odpowiedzialność. Interfejsy powinny być łączone tylko w wypadku zaistnienia takiej potrzeby. Kod 9 przedstawia strukturę interfejsów `IIdProvider` oraz `ISynchronizationProvider`, z której wynika zależność, że pierwszy dziedziczy po drugim. Dzięki takiemu rozbiciu zyskano większą możliwość dla tworzenia generycznych klas, gdyż każdy element `ISynchrionizationProvider` może być przetworzony poprzez operację na obiekcie `IIdProvider`.

Kod 9. Przykład użycia ISP w projekcie.

```
public interface ISynchronizationProvider : IIdProvider
{
    DateTime? LastSynchronization { get; set; }
    bool NeedSynchronization { get; set; }

    SynchronizationStatusEnum StatusSynchronization { get; set; }
}

public interface IIdProvider
{
    Guid Id { get; set; }
}
```

D – DIP (ang. *Dependecy Inversion Principle*) – reguła odwracania zależności. Kod implementujący reguły biznesowe nie powinien zależeć od kodu implementującego niskopoziomowe szczegóły [11]. Kod 10 przedstawia zależność kodu biznesowego jedynie od abstrakcji dla narzędzia pozwalającego na niskopoziomowe operacje na plikach tekstowych. Wstrzyknięcie implementacji zostało przeprowadzone za pomocą mechanizmu IoC.

Kod 10. Przykład użycia DIP w projekcie.

```
public interface ICdnService
{
    Task SaveFileAsync(IFormFile formFile);
    Task<String> GetFilePathAsync(Guid id);
}

private readonly ICdnService _cdnService;

public AttachmentsController(
    FileExtensionContentTypeProvider fileExtensionContentTypeProvider,
    ICdnService cdnService,
    IQueryDispatcher queryDispatcher,
    ICacheService memoryCacheService,
    ICommandDispatcher commandDispatcher) : base(queryDispatcher,
memoryCacheService, commandDispatcher)
{
    _cdnService = cdnService;
}
```

Reguły SOLID wpłynęły na kształt oraz czas implementacji projektu. Tabela 4 zawiera podsumowanie dotyczące poszczególnych reguł w postaci spisanych wniosków.

Tabela 4. Wpływ reguł SOLID na kształt oraz działanie projektu.

Reguła	Wniosek z zastosowania
SRP	Zwiększyła przejrzystość kodu powodując zdecydowanie mniejszy rozmiar klas. Rezultatem zastosowania reguły jest mniejsza ilość bloku kodu w pliku pozwalająca szybciej zdiagnozować problem.
OCP	Wpłynęła na czas implementacji kodu zmniejszając go. Tworzenie nowych funkcjonalności w oparciu o rozszerzaniu instancji klas o kolejne interfejsy jest prostszym zabiegiem niż modyfikacja gotowych rozwiązań. Zwiększyła się również stabilność systemu, gdyż implementacja nowej funkcjonalności nie narusza struktury istniejącego już kodu.
LSP	Znalazła zastosowanie w silniku synchronizacji danych między częścią serwerową, a kliencką, pozwalając ustandaryzować wymianę informacji przez procesy zachodzące w tle aplikacji. Pozwala również na bezkolizyjną wymianę sposobu synchronizacji danych poprzez nowe implementacje interfejsów.
ISP	Pozwoliła na stworzenie mechanizmów generycznych takich jak zapisywanie stanu synchronizacji produktu czy rozpoznawanie tożsamości encji w warstwie trwałości systemu. Rezultatem zastosowania reguły jest mniejsza ilość kodu oraz duże możliwości rozwojem niskim nakładem kosztu liczonego w czasie potrzebnym do implementacji.
DIP	Zmniejszyła zależności między implementacjami klas. Rezultatem czego jest większa ilość interfejsów zaimplementowanych tylko przez jedną klasę. Zwiększyło to liczbę linijek stworzonego kodu, co wpłynęło na czas implementacji wydłużając go.

6.2. Część wspólna

Wymienione w tym rozdziale wzorce projektowe zostały zaimplementowane przez obie części systemu kliencką, jak i serwerową. Rezultatem czego ich realizacja odniosła się do więcej niż jednej technologii.

6.2.1 Wstrzykiwanie zależności – przypadek użycia, wnioski

Wstrzykiwanie zależności jest implementacją paradygmatu Odwrócenia Sterowania (ang. *IoC – Inversion Of Control).* Polega na dostarczeniu do komponentów zależności klas przez technologię, dzięki czemu programista jest zwolniony z obowiązku tworzenia ich instancji [18].

Wzorzec zrealizowano na dwa sposoby zależne od środowiska programistycznego. Głównym powodem rozdzielania implementacji jest ograniczenie technologicznie w postaci braku mechanizmu refleksji w środowisku Flutter.

Dzięki refleksji dostępnej w języku C#, klasy tworzone są dynamicznie w czasie działania programu, po wcześniejszym zdefiniowaniu sposobu ich konstrukcji oraz przypisaniu pod żądaną abstrakcję. Za implementację wzorca wstrzykiwania zależności odpowiedzialna jest zewnętrzna

biblioteka `Autofac` wspominania w Tabeli 1. Definicja sposobu stworzenia instancji klasy oraz przypisania jej do odpowiedniego interfejsu odbywa się za pomocą implementacji modułu. Moduł zawiera możliwość nadpisania metody `Load` do której jako parametr przekazywany jest obiekt typu `ContainerBuilder`, który umożliwia przekazanie sposobu konstrukcji klasy. Kod 11 przedstawia przykład implementacji modułu CQRS, w którym można zauważyć, że interfejsy `ICommandHandler` oraz `IQueryHandler` rejestrowane są automatycznie po ich implementacji, miało to wpływ na czas implementacji rozwiązania skracając go. Na załączonym kodzie widać również sposobność do określenia czasu życia dla danej instancji, dzięki czemu w jednym miejscu znajduję się kod do zarządzania trwałością stworzonych klas.

Kod 11. Przykład implementacji wstrzykiwania zależności w technologii C#.

```
public class CqrsModule : Autofac.Module
{
    protected override void Load(ContainerBuilder builder)
    {
        var assembly = typeof(CqrsModule)
            .GetTypeInfo()
            .Assembly;

        builder.RegisterAssemblyTypes(assembly)
            .AsClosedTypesOf(typeof(ICommandHandler<>))
            .InstancePerLifetimeScope();

        builder.RegisterType<CommandDispatcher>()
            .As<ICommandDispatcher>()
            .SingleInstance();

        builder.RegisterAssemblyTypes(assembly)
            .AsClosedTypesOf(typeof(IQueryHandler<,>))
            .InstancePerLifetimeScope();

        builder.RegisterType<QueryDispatcher>()
            .As<IQueryDispatcher>()
            .SingleInstance();

        builder.RegisterType<GenericCommandHandler>()
            .As<IGenericCommandHandler>()
            .InstancePerLifetimeScope();
    }
}
```

Wstrzyknięcie klas odbywa się automatycznie, poprzez podanie pożądanego interfejsu w parametrze konstruktora po uprzednim jego zdefiniowaniu w module. Odpowiedzialnością technologii jest stworzenie instancji klas tak, by wszystkie zależności zostały do niej wstrzyknięte.

Środowisko Flutterowe nie oferuję tworzenia instancji klas za pomocą mechanizmu refleksji. Wzorzec wstrzykiwania zależności został zrealizowany dzięki zewnętrznej biblioteki `get_it` wspomnianej w Tabeli 3. Narzędzie umożliwia nadawanie odpowiednich atrybutów (`@injectable`, `@lazySingleton`) dla klas, zależnie od długości życia instancji. Klasy oznaczone wspomnianymi atrybutami są brane pod uwagę podczas procesu generowania kodu dla pliku `injection.dart` który zawiera metody umożliwiające tworzenie danej instancji. Proces generowania jest uruchamiany za pośrednictwem komendy - `flutter pub run build_runner watch --delete-conflicting-outputs`. Kod 12 przedstawia część klasy zawierającej wygenerowany kod dzięki zewnętrznej bibliotece `get_it`. Dostęp do instancji tak wygenerowanej klasy można uzyskać poprzez wywołanie metody `getIt<TClass>` gdzie `TClass` jest typem pożądanej instancji.

Kod 12. Część klasy posiadającej wygenerowany kod kontenera IoC.

```
void $initGetIt({String environment}) {
  getIt
    ..registerFactory<SynchronizationRepository>(
        () => SynchronizationRepository(getIt<NoteMeDatabaseFactory>()))
    ..registerFactory<MainSynchronizator>(() => MainSynchronizator(
          getIt<SynchronizationRepository>(),
          getIt<NotesSynchronizator>(),
          getIt<AttachmentsSynchronizator>(),
        ))
    ..registerLazySingleton<MessageBus>(() => MessageBus())
```

Wzorzec wstrzykiwania zależności miał bardzo istotny wpływ na implementację projektu. Przeniesienie odpowiedzialności tworzenia obiektów na technologię skróciło czas programowania aplikacji oraz uczyniło kod znacznie czytelniejszym.

Tabela 5. Wpływ wzorca wstrzykiwania zależności na projekt

Cecha	Wniosek
Próg wejścia	Wysoki. Zrozumienie istoty działania wzorca oraz sposobu określenia długości życia stworzonej instancji wymaga czasu. Na podstawie doświadczeń nabytych podczas implementacji projektu stwierdzono duży stopień trudności dotyczący zrozumienia wzorca.
Wydajność	Potencjalnie użycie refleksji wpływa negatywnie na wydajność rozwiązania. Nie odnotowano jednak zauważalnego spadku wydajności w działaniu prototypu.
Błędogenność	Umiarkowana. Źle określenie długości życia może wpłynąć negatywnie na poprawne działanie systemu. Podczas tworzenia implementacji odnotowano przypadek złego przypisania długości życia instancji w postaci *Singleton* zamiast domyślnego na żądanie. Skutkowało to błędem w postaci braku możliwości aktualizacji danych.
Użyteczność	Duża. Wzorzec został użyty w większości zaimplementowanych klas.
Rozszerzalność	Nie odnotowano przypadku rozszerzenia wzorca.
Sposób implementacji	Zależny od wybranej technologii. Użyte zostały zewnętrzne biblioteki.

6.2.2 Singleton – przypadek użycia, wnioski

Wzorzec Singleton zapewnia, że klasa będzie miała tylko jedną instancję (można utworzyć jeden obiekt tej klasy), a jednocześnie udostępnia globalny, jednolity sposób uzyskiwania i odwoływania się do tego obiektu z różnych fragmentów kodu (innych klas) [19]. Został zrealizowany w projekcie za pomocą mechanizmu wstrzykiwania zależności.

Kod 13 przedstawia rejestrację zależności głównego dystrybutora zadań dotyczących pobierania informacji. Nie ma potrzeby by `QueryDispatcher` był tworzony za każdym żądaniem do WebApi, gdyż nie posiada on żadnej unikalnej tożsamości ani informacji zależnej od kontekstu użytkownika. Rezultatem czego jest mniejsza alokacja pamięci na wskutek tworzenia tylko i wyłącznie jednej instancji obiektu.

Kod 13. Rejestracja zależności jako Singleton w kodzie C#

```
builder.RegisterType<QueryDispatcher>()
    .As<IQueryDispatcher>()
    .SingleInstance();
```

Kod 14 przedstawia rejestracji zależności fabryki dla połączenia do bazy danych. Obiekt ten również nie posiada informacji zależnych od kontekstu użytkownika. Jego odpowiedzialnością jest tworzenie połączenia do warstwy trwałości jaką jest baza danych SQLite.

Kod 14. Przykład rejestracji zależności jako Singleton w technologii Flutter.

```
@lazySingleton
class NoteMeDatabaseFactory {
  Future<Database> create() async {
    final Future<Database> database = openDatabase(
      join(await getDatabasesPath(), 'notes8.db'),
      onCreate: (db, version) async {
        await db.execute(
            "CREATE TABLE $notesTable (id TEXT PRIMARY KEY, name TEXT, content
TEXT, tags TEXT, createdAt DATETIME, modifiedAt DATETIME, latitude REAL, longitude
REAL, lastSynchronization DATETIME, statusSynchronization INTEGER, status
INTEGER);");
        await db.execute(
            "CREATE TABLE $synchroTable (id TEXT PRIMARY KEY, lastSynchronization
DATETIME, statusSynchronization INTEGER); ");
        await db.execute(
            "CREATE TABLE $attachmentsTable (id TEXT PRIMARY KEY, name TEXT,
createdAt DATETIME, lastSynchronization DATETIME, statusSynchronization INTEGER,
path TEXT, noteId TEXT); ");
      },
      version: 6,
    );

    return database;
  }
}
```

Singleton został użyty w wszystkich częściach prezentowanego rozwiązania. Tabela 6 przedstawia szczegółowe informacje na temat wpływu wzorca na implementację projektu.

Tabela 6. Wpływ wzorca Singleton na projekt

Cecha	Wniosek
Próg wejścia	Niski. Zrozumienie istoty wzorca oraz jego implementacja przebiegły sprawnie bez niespodziewanych sytuacji.
Wydajność	Nie odnotowano zauważalnego wpływu na wydajność.
Błędogenność	Umiarkowana. Zła implementacje wzorca uniemożliwiła aktualizacje danych po stronie aplikacji mobilnych.
Użyteczność	Umiarkowana. Wzorzec został zaimplantowany w wszystkich częściach systemu, lecz nie odnotowano potrzeby wymaganego użycia wzorca.
Rozszerzalność	Został rozszerzony dzięki wstrzykiwaniu zależności.
Sposób implementacji	Zależny od wybranej technologii oraz wzorca wstrzykiwania zależności.

6.2.3 Paginator – wnioski

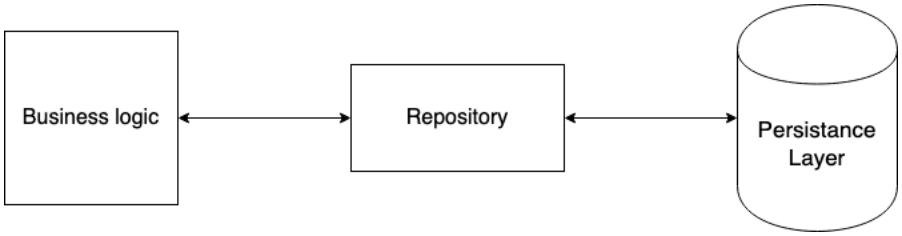
Wzorzec Paginator jest odpowiedzialny za wymianę kolekcji danych pomiędzy częścią serwerową, a kliencką systemu. Został dokładnie opisany w rozdziale 4. Tabela 7 przedstawia szczegółowy wpływ zastosowania wzorca na projekt.

Tabela 7. Wpływ wzorca Paginator na projekt

Cecha	Wniosek
Próg wejścia	Niski. Zrozumienie istoty wzorca oraz jego implementacja przebiegły sprawnie bez niespodziewanych sytuacji.
Wydajność	Pozytywny. Brak implementacji wzorca skutkował pobraniem zbyt dużej ilości danych co wpłynęło na czas trwania operacji.
Błędogenność	Nie odnotowano błędu związanego z implementacją wzorca.
Użyteczność	Wysoka. Wzorzec jest używany podczas pobierania kolekcji danych z serwera za pomocą klienta.
Rozszerzalność	Został rozszerzony dzięki wzorcu CQRS.
Sposób implementacji	Nie odnotowano zmian implementacji mimo różnych technologii.

6.2.4 Repozytorium – przypadek użycia, rozszerzalność, wydajność

Wzorzec repozytorium stanowi separację pomiędzy domeną, a warstwą trwałości. Dostarcza interfejs umożlwiający operacje dotyczące przechowywania oraz pobierania danych z ich źródeł. Warstwa abstrakcji nie powinna być związana z konkretną technologią, zależność ta powinna mieć miejsce dopiero na poziomie implementacji [9]. Rysunek 15 obrazuje przepływ danych z użyciem wzorca Repozytorium.



Rysunek 15. Diagram wzorca repozytorium.

Implementacja wzorca różniła się od użytej technologii. W części C# implementacja została przeprowadzona za pomocą wspomnianej w Tabeli 1 zewnętrznej biblioteki `EntityFramework` natomiast w środowisku Flutter użyto autorskiej implementacji.

Dostarczona zewnętrza biblioteka `EntityFramework` wspiera 24 silniki baz danych oraz posiada wiele funkcji umożliwiających przeprowadzenie skomplikowanych operacji bezpośrednio na warstwie trwałości systemu [9]. Oprócz ORM pełni ona również funkcję repozytorium dla zaimplementowanego projektu zarówno dla części klienckiej jak i serwerowej. Kod 15 przedstawia klasę odpowiedzialną za repozytorium części klienckiej, tworzenie dodatkowej abstrakcji ukrywającej ten kontekst bazodanowy przed wyższymi komponentami systemu wiąże się z nakładem dodatkowej, zbędnej pracy. Jako że `EntityFramework` umożliwia pełną kontrolę przed wstawieniem bądź modyfikowaniem danych oraz zapewnia wsparcie dla dużej ilości silników bazodanowych wyczerpuje w pełni założenia warstwy repozytorium.

Kod 15. Implementacja repozytorium w części klienckiej C#

```
public class NoteMeContext : DbContext
{
    private readonly SqliteSettings _settings;

    public DbSet<Attachment> Attachments { get; set; }
    public DbSet<Note> Notes { get; set; }
    public DbSet<Synchronization> Synchronizations { get; set; }

    public NoteMeContext(SqliteSettings settings)
    {
        _settings = settings;

        Database.EnsureCreated();
    }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlite($"Filename={_settings.Path}");

        base.OnConfiguring(optionsBuilder);
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.ApplyConfiguration(new NoteEntityConfiguration());
        modelBuilder.ApplyConfiguration(new AttachmentEntityConfiguration());
        modelBuilder.ApplyConfiguration(new SynchronizationEntityConfiguration());

        base.OnModelCreating(modelBuilder);
    }
}
```

Środowisko Flutter nie posiada tak rozbudowanej technologii ORM, więc zaimplementowano wzorzec repozytorium autorską metodą. Kod 16 prezentuje sposób implementacji bazowej, abstrakcyjnej klasy repozytorium zawierającej sposób wydobywania danych z warstwy trwałości - bazy danych Sqlite. Repozytoria używane przez logikę biznesową, dziedziczą po bazowym, lecz różnią się w stosunku do używanego typu. Potencjalna wymiana warstwy trwałości wiążę się z modyfikacja repozytorium bazowego.

Kod 16. Przykładowa implementacja wzorca Repozytorium w technologii Flutter

```
abstract class RepositoryBase<TModel extends SynchonizationProvider> {
  final NoteMeDatabaseFactory _databaseFactory;
  String table;

  RepositoryBase(this._databaseFactory);

  Future<void> insert(TModel item) async {
    final db = await _databaseFactory.create();
    db.insert(table, item.toJson(),
        conflictAlgorithm: ConflictAlgorithm.replace);
  }

  Future<void> insertMany(List<TModel> items) async {
    final db = await _databaseFactory.create();

    for (var item in items) {
      db.insert(table, item.toJson(),
          conflictAlgorithm: ConflictAlgorithm.replace);
    }
  }

  Future<void> update(TModel item) async {
    final db = await _databaseFactory.create();

    await db.update(
      table,
      item.toJson(),
      where: "id = ?",
      whereArgs: [item.id],
    );
  }

  Future<TModel> fetchById(String id) async {
    final Database db = await _databaseFactory.create();
    final List<Map<String, dynamic>> maps =
        await db.query(table, where: "id = ?", whereArgs: [id]);

    if (maps.length == 0) {
      return null;
```

```
    }

    final map = maps[0];
    return createFromMap(map);
  }

  Future<List<TModel>> fetch() async {
    final Database db = await _databaseFactory.create();
    final List<Map<String, dynamic>> maps = await db.query(table);

    return List.generate(maps.length, (i) {
      return createFromMap(maps[i]);
    });
  }

  Future clear() async {
    final Database db = await _databaseFactory.create();
    await db.delete(table);
  }

  TModel createFromMap(Map<String, dynamic> map);
}
```

Wzorzec repozytorium zapewnia dodatkową abstrakcję pomiędzy logiką biznesową, a warstwą techniczną. Dzięki czemu sposób wymiany warstwy przechowywania danych nie wiążę się z przebudową całego systemu. Tabela 8 przedstawia szczegółowy wpływ zastosowania wzorca na projekt.

Tabela 8. Wpływ wzorca Repozytorium na projekt

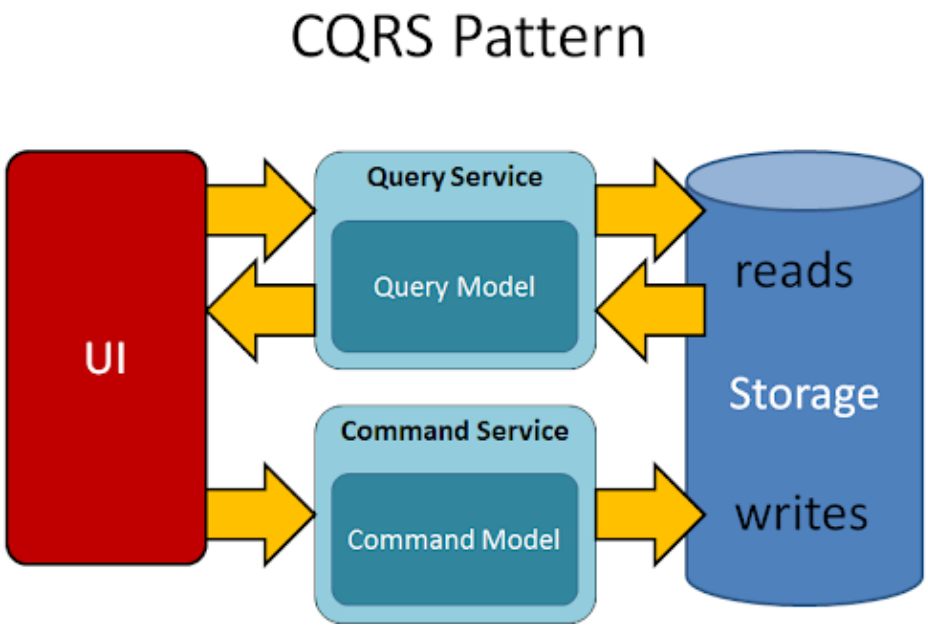
Cecha	Wniosek
Próg wejścia	Niski. Zrozumienie istoty wzorca oraz jego implementacja przebiegły sprawnie bez niespodziewanych sytuacji.
Wydajność	Nie odnotowano zauważalnego wpływu na wydajność.
Błędogenność	Nie odnotowano błędu związanego z implementacją wzorca.
Użyteczność	Niska. Wymiana warstwy trwałości jest rzadkim zjawiskiem oraz nie wystąpiła w czasie implementacji projektu.
Rozszerzalność	Nie odnotowano rozszerzenia wzorca.
Sposób implementacji	Różny. Technologia C# znacznie lepiej wspiera wzorzec poprzez dostęp do gotowego już rozwiązania. Część Flutterowa wymaga napisania własnej implementacji.

6.3. Część serwerowa

Część serwerowa aplikacji odpowiedzialna jest za operację dotyczące transformacji danych przed ich zapisaniem do warstwy trwałości. Wymienione w tym rozdziale wzorce w praktyce ograniczają się tylko do jednej technologii – Asp.net Core.

6.3.1 CQRS – przypadek użycia, wnioski

Wzorzec CQRS (ang. *Command Query Responsibility Segregation*) został opublikowany przez Grega Young-a oraz Udi Dahan-a. Polega na podziale metod operujących z warstwą trwałości na podstawie ich rodzaju, dokładniej na metody zapisu oraz odczytu [17]. Metody zapisu, obsługą komendy (ang. *Command*) zapisując zawarte w nich informację do bazy danych, z definicji nie zwracając żadnych wyników. Metody odczytu przeznaczone są do obsługi kwerendy (ang. *Query*), która jest zapytaniem o konkretne dane, w porównaniu do obsługi komend, metody odczytu zobligowane są do zwrócenia danych. Rysunek 16 przedstawia interakcję aplikacji z bazą danych poprzez wydzielone metody zapisu oraz odczytu.



Rysunek 16. Schemat wzorca CQRS

W części serwerowej zaimplementowano CQRS bez pomocy bibliotek trzecich. Kod 17 przedstawia zestaw operacji dotyczących pobrania kolekcji załączników dla notatek za pomocą użycia wzorca. Pierwszym etapem jest przyjęcie przez `Controller` żądania dotyczącego pobrania informacji w postaci `Query`, które następnie przekazywane jest do `QueryDispatcher`. `Dispatcher` pobiera z kontenera zależności odpowiednią implementację generycznego interfejsu na podstawie typu kwerendy. Obsługa oraz zwrot informacji następuję w klasie `AttachmentQueryHandler` której odpowiedzialnością jest poprawne obsłużenie kwerendy. Mechanika dotycząca obsługi komend odbywa się w podobny sposób, jedyną znaczącą różnicą jest brak informacji zwrotnych. Interfejsy dotyczące obsługi komend implementują metody typu `void`.

Kod 17. Kod implementujący obsługę kwerendy pobrania załączników

```
[Authorize]
[HttpPost]
public async Task<IActionResult> CreateAsync([FromBody] CreateAttachmentCommand
command)
{
    await DispatchAsync(command);
    var cached = GetDto<AttachmentDto>(command.Id);
    return Created("api/attachments", cached);
}

public interface IQueryDispatcher
{
    Task<TResult> DispatchAsync<TQuery, TResult>(TQuery query)
        where TQuery : IQueryProvider;
}

public class QueryDispatcher : IQueryDispatcher
{
    private readonly IComponentContext _componentContext;

    public QueryDispatcher(
        IComponentContext componentContext)
    {
        _componentContext = componentContext;
    }

    public Task<TResult> DispatchAsync<TQuery, TResult>(TQuery query) where TQuery
: IQueryProvider
    {
        var handler = _componentContext.Resolve<IQueryHandler<TQuery, TResult>>();
        return handler.HandleAsync(query);
    }
}

public interface IQueryHandler<in TQuery, TResult>
    where TQuery : IQueryProvider
{
    Task<TResult> HandleAsync(TQuery query);
}

public class AttachmentQueryHandler : IQueryHandler<GetAttachmentQuery,
PaginationDto<AttachmentDto>>
{
    private readonly INoteMeMapper _mapper;
    private readonly NoteMeContext _context;

    public AttachmentQueryHandler(
        INoteMeMapper mapper,
        NoteMeContext context)
    {
        _mapper = mapper;
        _context = context;
    }

    public async Task<PaginationDto<AttachmentDto>> HandleAsync(GetAttachmentQuery
query)
    {
        var queryable = _context.Attachments
            .Where(x => x.Note.UserId == query.RequestBy)
            .Where(query);

        var count = await queryable.CountAsync();
        var result = await queryable.TransformBy(query).ToListAsync();
```

```
            var pagination = new PaginationDto<Attachment>
            {
                Data = result,
                TotalCount = count
            };

            return _mapper.Map<PaginationDto<AttachmentDto>>(pagination);
        }
    }
```

Wzorzec CQRS znacznie wpłyną na architekturę części serwerowej czyniąc ją bardziej przejrzystą oraz mniej błędogenną. Tabela 9 przedstawia zestawienie wniosków wynikając z zastosowania wzorca CQRS w projekcie.

Tabela 9. Wpływ wzorca CQRS na projekt.

Cecha	Wniosek
Próg wejścia	Średni. Zrozumienie istoty wzorca oraz odejście od tradycyjnej formy serwisów obsługujących logikę biznesową zajęło większą ilość czasu niż bezpośrednie użycie tradycyjnej metody.
Wydajność	Nie odnotowano zauważalnego wpływu na wydajność.
Błędogenność	Nie odnotowano błędu związanego z implementacją wzorca.
Użyteczność	Duża. Wszelkie operacje dotyczące warstwy trwałości systemu są obsługiwane za pomocą wzorca CQRS.
Rozszerzalność	Wzorzec został rozszerzony poprzez Paginator oraz Wstrzykiwanie zależności.
Sposób implementacji	Własny.

6.3.2 DTO – przypadek użycia, wnioski

DTO (ang. *Data Transfer Object*) to kontener dla danych przenoszonych między różnymi warstwami lub procesami systemu. Nie zawiera żadnej logiki biznesowej, jest jedynie kontraktem pomiędzy dwoma częściami systemu. Należy do grupy wzorców dystrybucji [20].

Wzorzec zrealizowano w projekcie za pomocą wspomnianej w Tabeli 1 zewnętrznej biblioteki `Automapper`, umożliwiającej stworzenie oraz późniejsze wykonanie konfiguracji dotyczącej mapowania obiektów modelowych. Kod 18 prezentuje sposób wprowadzenia mapowania między dwoma modelami `User` oraz `UserDto`. Za całą konfigurację odpowiedzialna jest klasa `UserMapperProfile`, dzięki generycznej metodzie `CreateMap` ustawiono relacje mapowania między kontraktami. Model User posiada dane wrażliwe w postaci właściwości `Hash` oraz `Salt`. Dzięki zastosowaniu wzorca uzyskano możliwość przekazania tylko wymaganych danych w ustandaryzowany sposób.

Kod 18. Implementacja wzorca DTO w projekcie

```
public class UserMapperProfile : NoteMeMapperProfile
{
    public UserMapperProfile()
    {
        CreateMap<User, UserDto>();
```

```
            CreateMap<UserRegisterCommand, UserDto>();
            CreateMap<UserRegisterCommand, User>();
        }
    }

    public class User : IIdProvider,
        IStatusProvider,
        INameProvider,
        ICreatedAtProvider,
        IModifiedAtProvider
    {
        public Guid Id { get; set; }
        public StatusEnum Status { get; set; }
        public string Name { get; set; }
        public string Email { get; set; }
        public DateTime CreatedAt { get; set; }
        public DateTime ModifiedAt { get; set; }
        public string Hash { get; set; }
        public string Salt { get; set; }

        public ICollection<Template> Templates { get; set; } = new HashSet<Template>();
        public ICollection<Note> Notes { get; set; } = new HashSet<Note>();
    }

    public class UserDto : IDtoProvider,
        IIdProvider,
        INameProvider,
        IStatusProvider
    {
        public Guid Id { get; set; }
        public string Name { get; set; }
        public string Email { get; set; }
        public StatusEnum Status { get; set; }
    }
```

Wzorzec DTO wpłynął głównie na komunikację między klientem, a serwerem, czyniąc ją bezpieczną oraz uzyskując większą kontrolę. Tabela 10 zawiera zestawienie wniosków opisujących użyteczność wzorca pod kątem cech tworzenia oprogramowania.

Tabela 10. Wpływ wzorca DTO na projekt

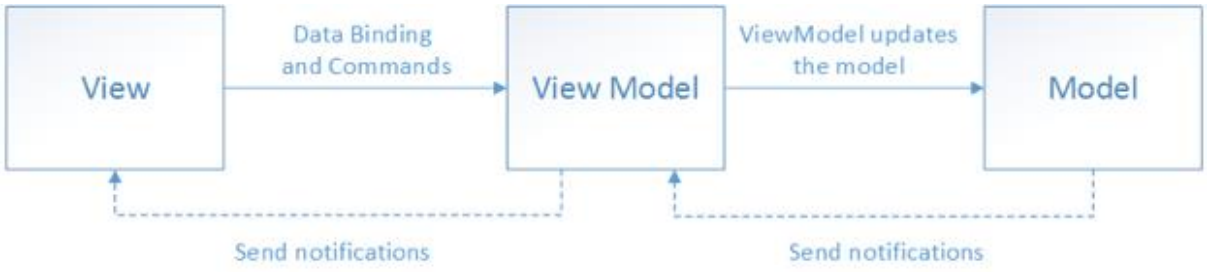
Cecha	Wniosek
Próg wejścia	Niski. Stworzenie dodatkowych klas modelowych oraz skonfigurowanie sposobu mapowania nie zajęło dużo czasu.
Wydajność	Nie odnotowano zauważalnego wpływu na wydajność.
Błędogenność	Nie odnotowano błędu związanego z implementacją wzorca.
Użyteczność	Duża. Wszelkie operację kontaktu między klientem, a serwerem odbywają się za pośrednictwem obiektów DTO.
Rozszerzalność	Wzorzec został rozszerzony poprzez Paginator.
Sposób implementacji	Po stronie serwerowej powiązany z zewnętrzną biblioteką.

6.4. Część kliencka

W części klienckiej zostały zaimplementowane 2 wzorce zależne od wybranej technologii. W środowisku Flutter cała architektura aplikacji została oparta o wzór BLOC (ang. *Bussiness Logic Componment*) natomiast rozwiązanie Xamarinowe wykorzystuję wzorzec MVVM (ang. *Model-View-ViewModel*).

6.4.1 MVVM (Model-View-ViewModel) – przypadek użycia, wnioski

Część kliencka oparta na środowisku Xamarin, została zrealizowana na podstawie wzorca strukturalnego Model-View-ViewModel MVVM. Wzorzec ten pomaga w czystym oddzieleniu logiki biznesowej od warstwy prezentacji aplikacji. Rozwiązuje to wiele problemów programistycznych związanych np. z utrzymaniem, tworzeniem testów jednostkowych czy współdzieleniem kodu między różnymi komponentami [9]. Rysunek 17 przedstawia interakcję warstw zawartych wzorcu. Widok (ang. *View)* jest odpowiedzialny za prezentacje informacji dla użytkownika końcowego. Wszelkie operacje przekształcenia tych informacji zachodzą w warstwie *View Model,* która zawiera również elementy logiki biznesowej. Ostatnia część *Model* jest odwzorowaniem warstwy trwałości informacji.



Rysunek 17. Schemat wzorca MVVM

Kod 19 przedstawia zestaw właściwości ViewModelu związanego z procesem logowania użytkownika do aplikacji. Każda operacja dotycząca przypisania tych wartości wywołuje zdarzenie `SetProperty` które dociera do warstwy prezentacji aplikacji, a następnie jest obsłużone w przedstawionym przez Kod 20 mechanizmie powiązania (ang. *Binding*). Dzięki tej operacji użytkownik zostaje natychmiast poinformowany o zdarzeniach logiki biznesowej przez warstwę prezentacji.

Kod 19. Właściwości warstwy ViewModel

```
public string Email
{
    get => _email;
    set => SetPropertyAndValidate(ref _email, value);
}

public string Password
{
    get => _password;
    set => SetPropertyAndValidate(ref _password, value);
}
```

Kod 20. Bindowanie propercji w warstwie widoku

```
<StackLayout Grid.Row="1" VerticalOptions="CenterAndExpand">
    <ActivityIndicator IsRunning="{Binding IsBusy}"></ActivityIndicator>
    <Entry Placeholder="{ extensions:Translate Email }" Text="{Binding
Email}"></Entry>
    <Entry Placeholder="{ extensions:Translate Password }" IsPassword="True"
Text="{Binding Password}" ></Entry>
    <Label Style="{StaticResource ErrorLabelStyle}" Text="{Binding Error}"
></Label>
</StackLayout>
```

Wszelkie operacje dotyczące zmiany stanu modelu przeprowadzane są przez wywołanie odpowiednich funkcji w warstwie ViewModel. Daje to możliwość przetestowania logiki poprzez wymianę implementacji interfejsów odpowiedzialnych za zapisywanie do warstwy trwałości za pośrednictwem mechanizmu mockowania danych.

Każdy widok posiada swój własny ViewModel, który może odwoływać się do dowolnych modeli bądź komponentów odpowiedzianych za transformacje danych. Rezultatem czego jest możliwość współdzielenia kodu między różnymi ekranami aplikacji w postaci kompozycji.

Architektura zbudowanej aplikacji jest ściśle powiązana z wzorcem MVVM. Każdy został zaimplementowany z jego pomocą. Tabela 11 zawiera zestawienie wniosków wynikających z użycia w solucji.

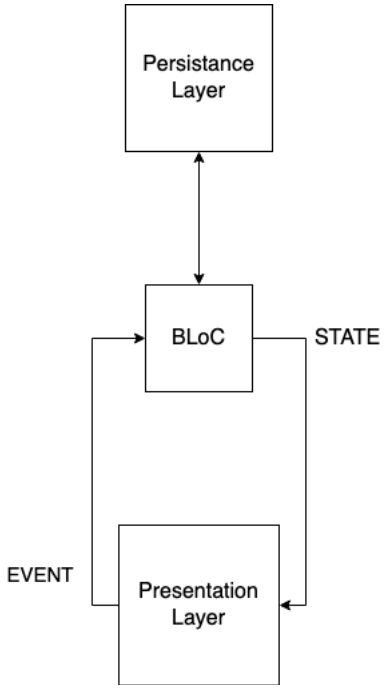
Tabela 11. Wpływ wzorca MVVM na projekt.

Cecha	Wniosek
Próg wejścia	Średni. Zrozumienie istoty wzorca oraz mechanizmu powiązania propercji między widokiem, a jego abstrakcją nie jest trywialnym problemem.
Wydajność	Nie odnotowano zauważalnego wpływu na wydajność.
Błędogenność	Nie odnotowano błędu związanego z implementacją wzorca.
Użyteczność	Duża. Wszystkie widoki aplikacji mobilnej korzystają z implementacji wzorca. Powstały kod jest przejrzysty oraz dobrze ustrukturowany. Klasy są mniejsze dzięki rozdzieleniu odpowiedzialności na trzy warstwy - *View, ViewModel, Model*
Rozszerzalność	Nie odnotowano rozszerzenia wzorca w projekcie.
Sposób implementacji	Własny, bez pomocy użycia bibliotek zewnętrznych.

6.4.2 BLOC (Business Logic Component) – przypadek użycia, wnioski

Bloc (ang. *Business Logic Component*) to wzorzec projektowy umożlwiający separację kodu warstwy prezentacji od logiki biznesowej. Sprawia, że kod staję się wydajny, łatwy do testowania oraz dostępny do wielokrotnego użytku. Główną ideą wzorca jest zarządzanie aplikacją za pomocą kontroli nad jej stanami poprzez odpowiednie zdarzenia [15]. Rysunek 18 przedstawia schemat przepływu informacji pomiędzy warstwami aplikacji.

Został on zaprezentowany przez Paolo Soaresa i Conga Hui-a podczas konferencji *DartConf* w 2018 roku, po czym stał się jednym z najpopularniejszych wzorców projektowych używanych podczas tworzenia wieloplatformowych aplikacji mobilnych opartych na platformie Flutter [16].



Rysunek 18. Schemat wzorca projektowego BLOC

Zaimplementowany został za pomocą wspomnianych w Tabeli 3 zewnętrznych bibliotek `bloc` oraz `flutter_bloc`. Kod 21 prezentuje `bloc` odpowiedzialny za przetwarzanie danych dotyczących procesu logowania. LogicBloc dziedziczy po klasie generycznej klasie `Bloc` która przyjmuje bazowe, abstrakcyjne typy `LoginEvent` oraz `LoginState`. Stan początkowy został określony za pomocą metody `initialState`. Funkcja `mapEventToState` za pomocą słów kluczowych `async*` oraz `yield` przekazuje do strumienia przeznaczonego dla warstwy prezentacji odpowiedni stan nie przerywając jej działania.

Kod 21. Bloc odpowiedzialny za logikę biznesową logowania.

```
@injectable
class LoginBloc extends Bloc<LoginEvent, LoginState> {
  final ApiService _apiService;

  LoginBloc(this._apiService);

  LoginState get initialState => LoginInitial();

  @override
  Stream<LoginState> mapEventToState(LoginEvent event) async* {
    if (event is LoginButtonPressed) {
      yield LoginLoading();

      try {
        final response = await _apiService.post(LoginEndpoint, event);
        if (!response.isCorrect) {
```

```
          yield LoginFailure(error: response.body.toString());
          return;
        }
        final user = Jwt.fromJson(response.json);

        yield LoginSuccess(user);
      } catch (error) {
        yield LoginFailure(error: error.toString());
      }
    }
  }
}
```

Wstrzyknięcie oraz zarządzanie instancją obiektu `Bloc` zostało przedstawione za pomocą Kodu 22. `BlocProvider` dzięki mechanizmowi `IoC` tworzy instancję `LoginBloc` zwracając ją poprzez przesłonięcie metody `create: (context),` przywiązując do odpowiedniego contextu. Dzięki dowiązaniu `Bloc` do `context`, wewnątrz widgetu uzyskano dostęp do instancji `Bloc` poprzez metodę `BlocProvider.of`. Metoda `builder: (context, state)` zostaje wywoływana za każdym zwróceniem stanu przez `Bloc`, co umożliwia przedstawioną przez Kod 23, reakcję w postaci przebudowy istniejącego widgetu (warstwy prezentacji) na potrzebę określoną przez odpowiedni stan.

Bloc zaprezentowany w Kod 21, zawiera przekazywanie informacji za pomocą strumieni. Długość ich życia zarządzana jest dzięki użyciu klasy `BlocProvider`. Kiedy nie jest on już dłużej potrzebny w aplikacji odpowiedzialnością *providera* jest zamknięcie wszystkich strumieni, co zapobiega wyciekom pamięci.

Kod 22. Stworzenie BLOC oraz przekazanie jego instancji do BlocProvider

```
  @override
  Widget build(BuildContext context) {
    return Scaffold(
        body: BlocProvider(create: (context) {
      return getIt<LoginBloc>();
    }, child: BlocBuilder<LoginBloc, LoginState>(builder: (context, state) {
      if (state is LoginSuccess) {
        BlocProvider.of<AuthenticationBloc>(context).add(AppStarted());
      }

      return UnloggedBackgroundBox(
          child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          TitleNoteMeText(text: 'NoteMe !'),
          SizedBox(height: 100),
          LoginForm(),
        ],
      ));
    })));
  }
```

Kod 23. Reakcja warstwy prezentacji na odpowiednie stany BLOC

```
return BlocListener<LoginBloc, LoginState>(
    listener: (context, state) {

      if (state is LoginFailure) {
        Scaffold.of(context).showSnackBar(
          SnackBar(
            content: Text('${locale.login.invalid}'),
            backgroundColor: Colors.red,
          ),
        );
      }
    },
    child: BlocBuilder<LoginBloc, LoginState>(
      builder: (context, state) {

        if (state is LoginSuccess) {
          BlocProvider.of<AuthenticationBloc>(context)
              .add(LoggedIn(state.user));
        }
```

Dzięki użyciu wzorca BLOC uzyskano separację między logiką biznesową oraz warstwą prezentacji nadając rozwiązaniu odpowiednią strukturę. Tabela 12 przedstawia zestawienie wniosków dotyczących wpływu wzorca BLOC na prototyp.

Tabela 12. Wpływ wzorca BLOC na projekt.

Cecha	Wniosek
Próg wejścia	Średni. Motyw sterowania warstwą aplikacji za pomocą zdarzeń oraz stanów nie był trywialny do własnej implementacji. Zewnętrzna biblioteka zapewniła dużo materiałów wprowadzających oraz skróciła czas wdrożenia wzorca.
Wydajność	Nie odnotowano zauważalnego wpływu na wydajność.
Błędogenność	Nie odnotowano błędu związanego z implementacją wzorca. Próba własnej implementacji wzorca mogłaby skończyć się wyciekiem pamięci spowodowanym poprzez zaniedbanie zamykania strumieni.
Użyteczność	Duża. Cała struktura warstwy prezentacji została zaimplementowana za pomocą wzorca. Tworzenie nowych funkcji wiążę się z dodawaniem nowych stanów oraz zdarzeń.
Rozszerzalność	Funkcjonalność tworzenia bloc została rozszerzona za pomocą wzorca wstrzykiwania zależności.
Sposób implementacji	Biblioteki zewnętrzne `bloc` oraz `Flutter_bloc` znacznie przyspieszyły implementację wzorca.

7. Podsumowanie

Przeprowadzona analiza potwierdziła użyteczność wzorców projektowych podczas tworzenia wieloplatformowych aplikacji mobilnych. Podsumowując zalety uzyskane dzięki zastosowaniu dobrych praktyk programistycznych, stwierdzono:

1. Wzorce projektowe nie zmniejszyły wydajności prototypu aplikacji,
2. Użyteczność większości wzorców dotyczyła całej architektury, rezultatem czego został zwiększony poziom standaryzacji projektu,
3. Niską błędogenność,
4. Wsparcie dla implementacji wzorców w postaci dobrze udokumentowanych bibliotek zewnętrznych,
5. Większą testowalność logiki biznesowej dotyczącej warstwy prezentacji aplikacji mobilnej,
6. Mniejszą potrzebę modyfikacji kodu, w celu stworzenia nowej funkcjonalności,

Negatywy wpływ stosowania wzorców projektowych stwierdzony na podstawie wniosków z Rozdziału 6 dotyczy głównie przeważającego średniego progu wejścia. Czas przeznaczony na początkową implementację wzorca oraz zapoznanie się z jego schematem wydłużył proces powstawania bazowych struktur prototypu.

Prototyp aplikacji mobilnej NoteMe, może zostać w łatwy sposób rozszerzony o pobieranie kolekcji danych z web api czy implementację nowych widoków w postaci tworzenia szablonów notatek dzięki wprowadzonej architekturze.

Wykaz rysunków

Wykaz tabel

Wykaz kodu

Prace cytowane

[1]. C. Alexander „A pattern language”, ISBN-13: 9780195019193, Wyd. Oxford University Press, 1977.

[2]. PWN, https://encyklopedia.pwn.pl/haslo/system-operacyjny;3982217.html – 2020.04.23

[3]. I. Krajci, D. Cummings, *Android on x86*, ISBN-13 (pbk): 978-1-4302-6130-8, Wyd. Apress open, 2013

[4]. PostgreSQL, https://www.postgresql.org/docs - 2020.04.23

[5]. EuroLinux, https://pl.euro-linux.com/blog/od-jadra-do-setek-dystrybucji-skad-wzial-sie-linux/ - 2020.04.23

[6]. Git-scm, https://git-scm.com/book/pl/v2/Pierwsze-kroki-Podstawy-Git - 2020.04.24

[7]. Flutter docs, https://flutter.dev/docs/resources/technical-overview - 2020.04.25

[8]. Elevatex, https://elevatex.de/how-flutter-works-under-the-hood-and-why-it-is-game-changing - 2020.04.25

[9]. Dokumentacja - Microsoft https://docs.microsoft.com/pl-pl - 2020.04.25

[10]. Słownik komputerowy - https://www.slownik-komputerowy.pl - 2020.04.25

[11]. R. C. Martin *Czysta Architektura*, ISBN 978-83-283-4225-5, Wyd. Helion, 2018.

[12]. Dokumentacja systemu Android - https://developer.android.com/docs - 2020.04.29

[13]. Dokumentacja .Net Mono - https://www.mono-project.com/docs/about-mono/dotnet-integration/ - 2020.04.29

[14]. CodeProject - https://www.codeproject.com – 2020.04.30

[15]. Bloc - https://bloclibrary.dev/ - 2020.04.30

[16]. FlutterDevs - http://flutterdevs.com – 2020.04.30

[17]. BulldogJob - https://bulldogjob.pl/articles/122-cqrs-i-event-sourcing-czyli-latwa-droga-do-skalowalnosci-naszych-systemow_ – 2020.05.09

[18]. TypeOfWeb - https://typeofweb.com/wzorce-projektowe-dependency-injection/ - 2020.05.09

[19]. PJATK – http://edu.pjwstk.edu.pl/wyklady/zap/scb/W5/W5.htm – 2020.05.09

[20]. UJ - http://users.uj.edu.pl/~ciesla/wzorce/wzorce_15.pdf – 2020.05.09

[21]. Altextsoft - https://www.altexsoft.com/blog/mobile/pros-and-cons-of-xamarin-vs-native/ - 2020.05.09