UNIWERSYTET KARDYNAŁA STEFANA WYSZYŃSKIEGO
W WARSZAWIE

WYDZIAŁ MATEMATYCZNO-PRZYRODNICZY
SZKOŁA NAUK ŚCISŁYCH

Filip Krawczyk

Nr alb.: 119230

Kierunek studiów: Informatyka

Aplikacja mobilna do zarządzania biblioteką fiszek na system iOS

Praca licencjacka
Promotor:
dr inż. Artur Mikitiuk

Warszawa, 2025

Spis treści

Spis rysunków

Wstęp

Wprowadzenie

W czasach rosnących wymagań edukacyjnych i zawodowych efektywne metody przyswajania wiedzy są kluczowym elementem rozwoju osobistego. Samodzielna nauka to obszerna dziedzina, oferująca różne techniki dostosowane do indywidualnych potrzeb uczących się osób. Jednym z jej najskuteczniejszych narzędzi, jest powtarzanie interwałowe.

Jest to metoda nauczania, wykorzystująca zaplanowane odstępy w czasie pomiędzy kolejnymi sesjami nauki, aby zwiększyć efektywność przenoszenia informacji do pamięci długotrwałej. Kluczowym elementem tej metody jest obserwacja, że ludzki umysł lepiej przyswaja i zapamiętuje informacje, gdy są one powtarzane w pewnych odstępach.

Najpopularniejszym sposobem powtarzania interwałowego jest używanie fiszek. Są to dwustronne karty, na których awersie jest napisane słowo lub pytanie, a na rewersie jego definicja lub odpowiedź. Podczas nauki użytkownik przegląda fiszki, próbując przypomnieć sobie informacje z rewersu, a następnie weryfikuje poprawność swojej odpowiedzi. Karty, których zawartość została poprawnie zapamiętana, są odkładane do powtórzenia w dalszej przyszłości, podczas gdy te, które sprawiają trudność, są powtarzane częściej.

Motywacja

Wraz z rewolucją technologiczną wiele tradycyjnych narzędzi edukacyjnych przeszło do świata wirtualnego. Fiszki nie stanowią w tym wypadku wyjątku. Dzięki powszechnej dostępności smartfonów i innych urządzeń mobilnych, pojawiły się różne aplikacje oferujące wirtualne odpowiedniki papierowych kart do nauki, jednocześnie zawierające funkcjonalność niemożliwą do zaoferowania w ich analogowym odpowiedniku.

Mimo że istnieje wiele aplikacji do tworzenia fiszek w sklepie App Store, posiadają one wiele wad. Duża część z nich próbuje wymusić na użytkowniku kupno subskrypcji wersji premium, posiadają dużo reklam utrudniających korzystanie z aplikacji, albo ich zakup wymaga znacznego jednorazowego wydatku pieniężnego. Ponadto wiele z dostępnych rozwiązań charakteryzuje się nadmierną złożonością interfejsu, gdzie liczne funkcje i opcje zaciemniają podstawowy cel aplikacji.

Cel pracy

Głównym celem niniejszej pracy jest zaprojektowanie i implementacja minimalistycznej aplikacji do tworzenia oraz zarządzania katalogiem fiszek. Projekt koncentruje się na opracowaniu rozwiązania charakteryzującym się intuicyjnym, nieintruzywnym interfejsem użytkownika, który eliminuje zbędne elementy mogące zakłócać proces nauki.

Aplikacja musi mieć możliwość zapełnienia tekstem przodu oraz tyłu fiszki. Dodatkowo w przypadku materiałów, które ciężko jest zapisać w postaci tekstu, jak na przykład wzory matematyczne, powinna dawać możliwość załączać zdjęcia. Częścią aplikacji powinien być także tryb nauki, w którym program wylosuje kolejność fiszek do nauki, oraz po nauce zapamięta, które z nich zostały oznaczone jako nienauczone i na podstawie tego będzie dostosowywać kolejność w następnych sesjach nauczania.

Z technicznego punktu widzenia, aplikacja zostanie zaprojektowana dla platformy iOS wykorzystaniem zintegrowanego środowiska programistycznego Xcode. Implementacja będzie oparta na języku Swift oraz bibliotece SwiftUI, co pozwoli na stworzenie responsywnego interfejsu użytkownika. Do zarządzania i przechowywania danych zostanie wykorzystana nowa biblioteka od Apple o nazwie SwiftData, zapewniająca efektywną warstwę persystencji dla informacji o fiszkach i postępach użytkownika.

Struktura pracy

Praca składa się z pięciu rozdziałów. Pierwszy z nich przedstawia istniejące aplikacje w sklepie App Store o zbliżonej funkcjonalności. Drugi omawia narzędzia i technologie wykorzystane podczas tworzenia tej pracy. Rozdział trzeci zawiera diagramy UML służące do wizualizacji modelu systemu. W rozdziale czwartym przedstawiona została struktura interfejsu aplikacji. Rozdział piąty zawiera szczegółowe omówienie kodu z podziałem na poszczególne foldery.

Rozdział 1

Istniejące rozwiązania na rynku

1.1. Quizlet

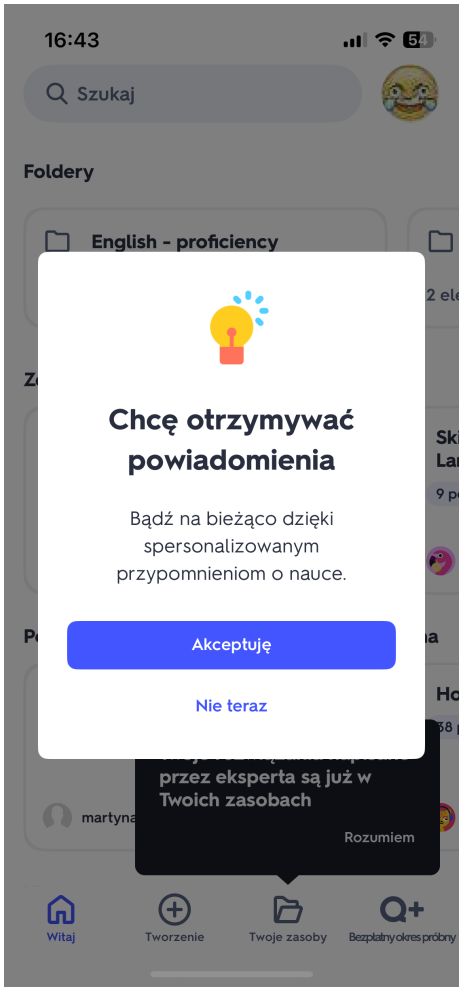
Rysunek 1.1: Quizlet: Pierwszy ekran

Rysunek 1.2: Quizlet: Drugi ekran

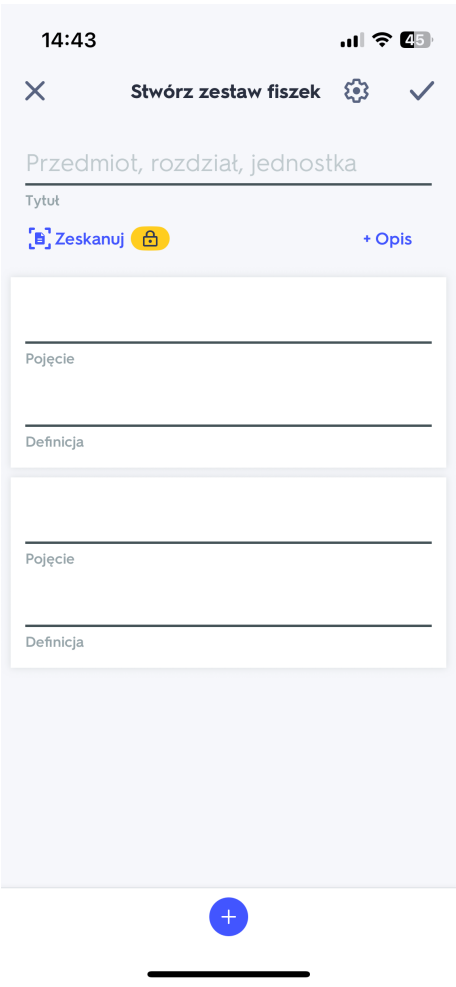
Quizlet [10] to amerykański serwis internetowy do nauki posiadający około 60 milionów użytkowników [9]. Umożliwia tworzenie zestawów fiszek oraz udostępnianie ich innym użytkownikom. Istnieje także opcja założenia konta dla nauczycieli, gdzie mogą oni utworzyć klasę i zapraszać do niej swoich uczniów. Ponadto wersja premium aplikacji dodaje dodatkowe funkcjonalności między innymi usuwanie reklam, dodawanie zdjęć do stworzonych fiszek oraz umożliwia zaawansowane formatowanie tekstu. Quizlet dostępny

jest w paru formatach: w postaci strony internetowej, a także jako aplikacja na platformy Android oraz iOS. Na potrzeby pracy omawiana zostanie wersja aplikacji na system iOS.

Do korzystania z aplikacji konieczne jest utworzenie lub zalogowanie się do istniejącego konta. Pierwszą rzeczą widoczną po zalogowaniu jest pełnoekranowy komunikat proponujący zakup subskrypcji o nazwie QuizletPlus. Po wyjściu z widoku subskrypcji w aplikacji pojawia się kolejne okno zalecające włączenie powiadomień. Aplikacja wyświetla dużą ilość elementów zanim pozwoli użytkownikowi na dostęp do swojej głównej funkcjonalności, co może zniechęcić część potencjalnych zainteresowanych do dalszego korzystania z tego serwisu.



Rysunek 1.3: Quizlet: Komunikat wyświetlany po zalogowaniu się

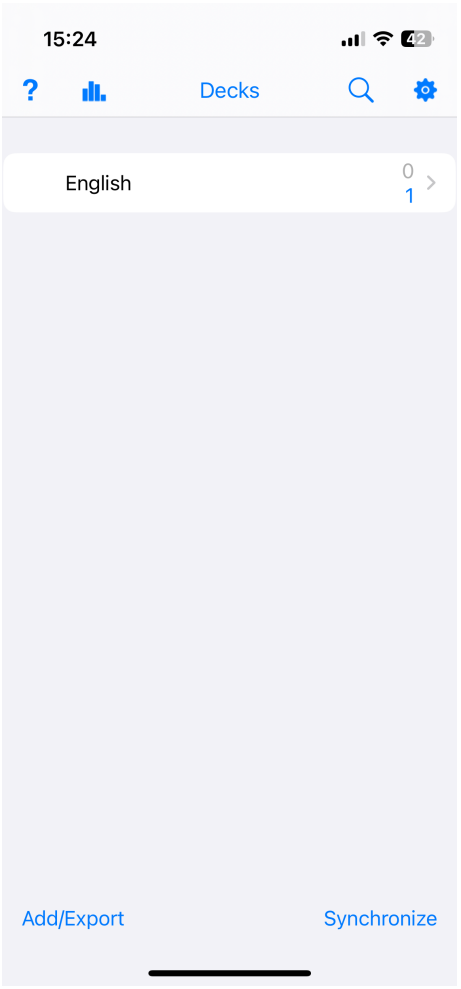


Rysunek 1.4: Quizlet: Ekran tworzenia zestawu fiszek

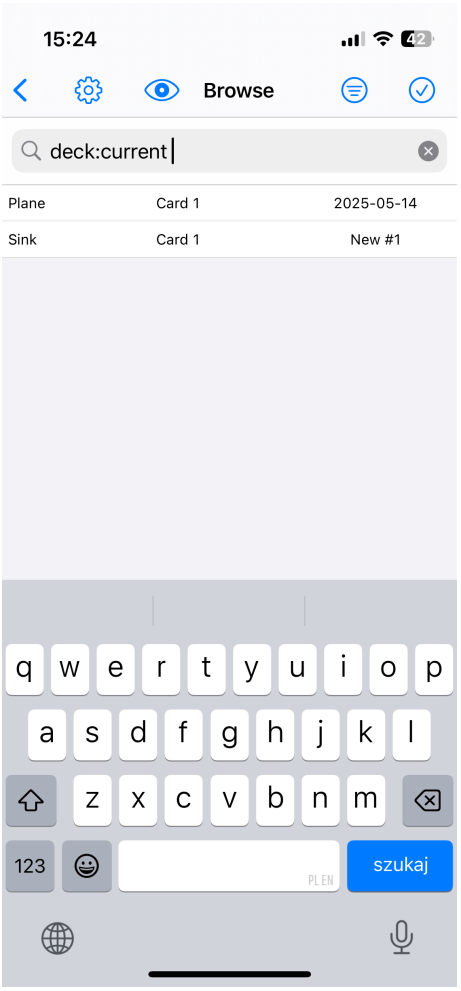
Struktura aplikacji jest podzielona na cztery sekcje używając guzików nawigacyjnych w dolnej części ekranu. Pierwsza z nich to „Witaj", gdzie znajduje się podsumowanie ostatnio używanych i proponowanych zestawów fiszek. Następnie znaleźć można sekcję „Tworzenie", gdzie użytkownik ma możliwość utworzyć zestaw fiszek, folder, albo klasę. Sekcja „Twoje zasoby" daje dostęp do wszystkich utworzonych do tej pory zestawów fiszek. Ostatnia z nich to "Bezpłatny okres próbny" dająca możliwość zakupu wersji premium aplikacji za pomocą subskrypcji QuizletPlus. Propozycja zakupu subskrypcji

wyświetla się nie tylko tutaj, lecz także w innych miejscach aplikacji. Przy przyciskach oferujących funkcjonalności dostępne wyłącznie w wersji premium wyświetla się dymek zachęcający użytkownika do zakupienia QuizletPlus.

1.2. AnkiMobile Flashcards



Rysunek 1.5: Anki: Ekran główny



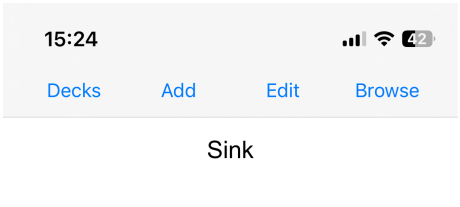
Rysunek 1.6: Anki: Lista fiszek wewnątrz zestawu fiszek

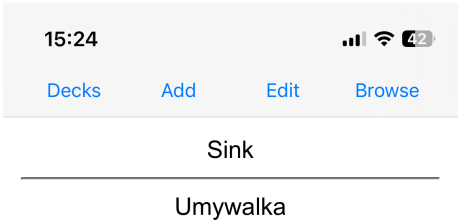
Anki [12] jest open-source programem do tworzenia fiszek współgrającym z darmowym portalem AnkiWeb, dzięki któremu można zapisywać je w chmurze. Program reklamuje się wykorzystaniem specjalnego algorytmu dobierania fiszek w trakcie nauki, aby optymalizować ich zapamiętywanie. Bazuje on na algorytmie SuperMemo 2 [2]. Darmowa wersja aplikacji dostępna jest na platformę Android, natomiast wersja na system iOS kosztuje 129,99 zł. Cena ta jest stosunkowo wysoka w porównaniu z innymi aplikacjami w tej kategorii.

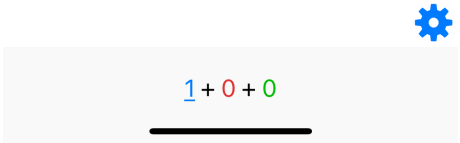
Po włączeniu aplikacja nie wymaga konta do tworzenia fiszek, użytkownik od razu dostaje dostęp do domyślnego ekranu aplikacji, gdzie widoczne są zestawy fiszek. Po

wejściu w dany zestaw wyświetlany zostaje ekran nauczania umożliwiający naukę lub dodawanie nowych fiszek. Każdą z fiszek można uzupełnić mediami o różnych formatach. Oprócz sformatowanego tekstu lub zdjęcia, treść może być przedstawiona w postaci nagrania audio, rysunku, działania matematycznego, bloku kodu lub nawet załącznika. Wersje aplikacji na systemy macOS, Windows oraz Linux dają możliwość rozszerzenia obszernej już funkcjonalności Anki za pomocą wtyczek, lecz niestety nie jest to możliwe w aplikacji mobilnej AnkiMobile.

Aplikacja charakteryzuje się dość surowym wyglądem, większość interfejsu użytkownika składa się z domyślnych elementów dostępnych w systemie iOS. W AnkiMobile ilość dostępnych funkcjonalności zdaje się grać większą rolę, niż intuicyjność obsługi. Przykładem tego może być zestaw fiszek — po wejściu do uprzednio dodanego zestawu użytkownik od razu umieszczany jest w trybie nauki, nawet jeśli zestaw nie posiada jeszcze żadnych elementów. Pusty zestaw gratuluje użytkownikowi pomyślnego skończenia nauki, co może powodować zdezorientowanie osób korzystających z aplikacji po raz pierwszy. Dodatkowo aplikacja nie posiada żadnego przewodnika lub wskazówek informujących użytkownika o sposobie dodawania nowych fiszek.





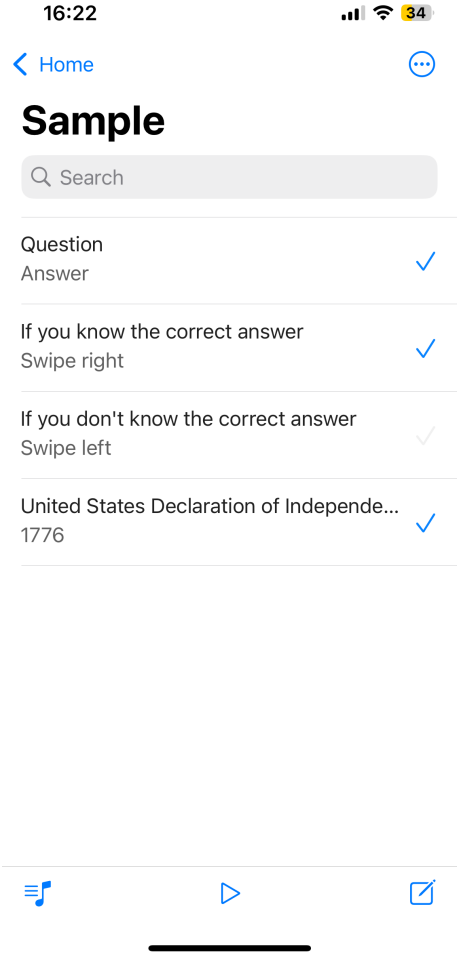


Rysunek 1.7: Anki: Widok fiszki

Rysunek 1.8: Anki: Widok fiszki po kliknięciu

1.3. FlashCards

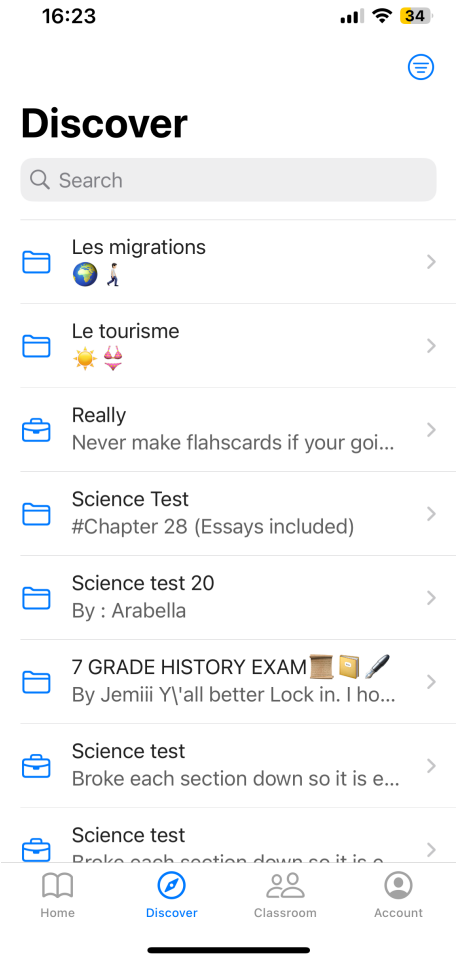
Rysunek 1.9: FlashCards: Ekran główny



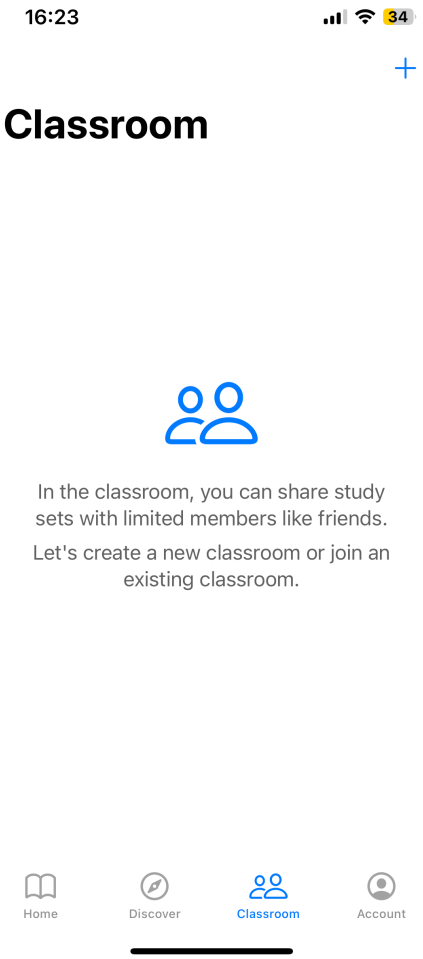
Rysunek 1.10: FlashCards: Widok przykładowego zestawu fiszek

FlashCards [13] to aplikacja, której autorem jest osoba posiadająca konto deweloperskie w sklepie App Store o nazwie Toshiki Motomura. Jako kolejna charakteryzuje się prostotą wyglądu stawiającym na użycie podstawowych elementów udostępnianych przez system iOS. Oprócz podstawowych funkcjonalności aplikacja posiada także możliwość dzielenia się swoimi zestawami fiszek z innymi użytkownikami.

Pierwszym ekranem widocznym po otwarciu aplikacji jest sekcja „Home” wyświetlająca wszystkie zestawy fiszek utworzone dotychczas przez użytkownika. Na start udostępniany jest domyślny zestaw z przykładowymi pojęciami. Następnie za pomocą dolnego paska nawigacji przejść można do sekcji „Discover”, gdzie jak już wcześniej wspomniano użytkownik znaleźć może zestawy fiszek utworzone przez inne osoby korzystające z aplikacji. Kolejną sekcją jest „Classroom” — udostępnia możliwość tworzenia prywatnych grup wewnątrz, których użytkownicy mają szansę dzielić się swoimi fiszkami. Ostatnią sekcją na pasku nawigacji jest „Account”, gdzie można znaleźć ustawienia aplikacji.



Rysunek 1.11: FlashCards: Sekcja z zestawami udostępnianymi przez innych użytkowników



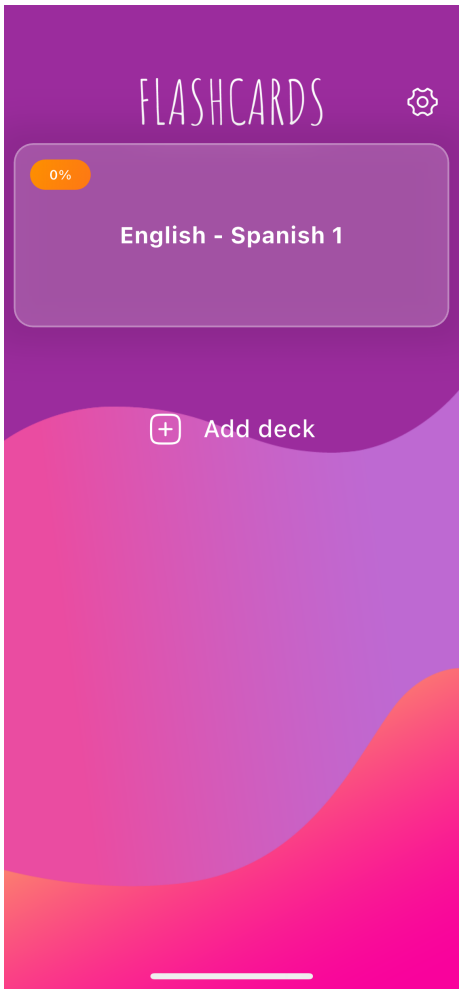
Rysunek 1.12: FlashCards: Sekcja pozwalająca na utworzenie klasy

1.4. Flashcards maker

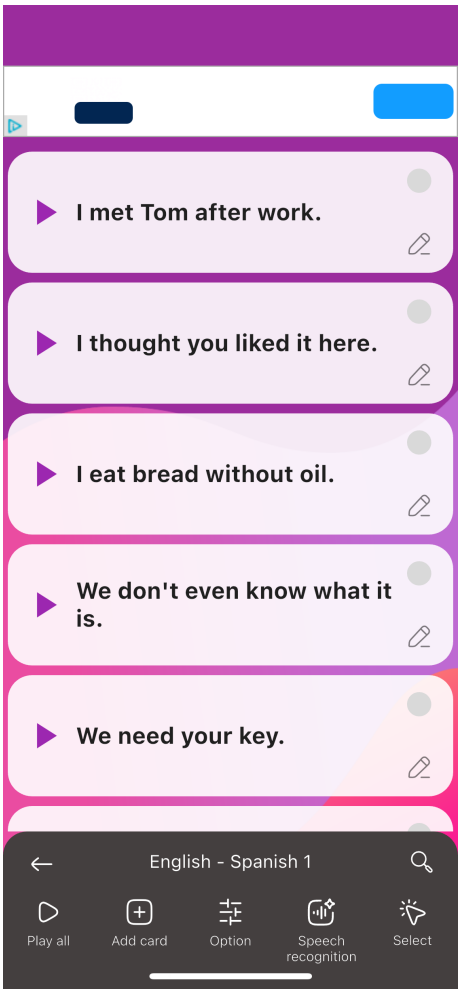
Flashcard maker [4] to kolejna aplikacja autorstwa niezależnego dewelopera —- w tym wypadku jest to Yoshinobu Ikeda. Wyróżnia się na tle innych opisanych w tym rozdziale aplikacji przede wszystkim swoim wyglądem — użycie jaskrawych kolorów oraz układ strony głównej i okienek sprawia, że nie pasuje ona do zwyczajnego stylu aplikacji iOS. Kolejną cechą charakterystyczną jest nastawienie na naukę języków. Podczas tworzenia zestawu fiszek aplikacja pyta o dwa języki, między którymi będzie ten zestaw tłumaczył.

Po otwarciu aplikacji Flashcard maker widoczna jest lista zestawów fiszek z przykładowym zestawem do nauki języka hiszpańskiego. Pod nim znajduje się guzik umożliwiający utworzenie nowego zestawu. Aplikacja posiada kilka trybów nauki: test wielokrotnego wyboru, test pisowni, oraz tryb samodzielnego sprawdzania. Pierwszy z nich polega na wyborze popranej odpowiedzi z 4 potencjalnych wybranych losowo z danego katalogu fiszek, natomiast drugi sprawdza pisownię użytkownika poprzez wpisanie frazy w innym języku litera po literze. Ostatni tryb działa analogicznie do trybu dostępnego w aplikacji

będącej tematem pracy — na ekranie ukazuje się karta z frazą, która odwraca się po kliknięciu pokazując jej tłumaczenie. Jeśli odpowiedź użytkownika była poprawna zaznacza on przycisk „✓”. Dodatkowo frazy pokazane na kartach czytane są automatycznie.



Rysunek 1.13: Flashcards maker: Ekran główny



Rysunek 1.14: Flashcards maker: Widok przykładowego zestawu fiszek

Rozdział 2

Zastosowane technologie

2.1. Xcode

Środowiskiem programistycznym wybranym do zarządzania projektem jest Xcode [8], czyli zestaw narzędzi deweloperskich do tworzenia aplikacji na platformy Apple. W ich skład wchodzi edytor kodu wewnątrz którego można kompilować, uruchamiać oraz debugować rozwijane aplikacje. Częścią oprogramowania Xcode jest również Simulator umożliwiający testowanie aplikacji na różnych urządzeniach Apple, takich jak iPhone, iPad lub AppleWatch bez potrzeby korzystania ich fizycznych odpowiedników. Xcode nie ma obecnie konkurencji na rynku — wybór tego zestawu narzędzi jest oczywisty, ponieważ jako jedyny pozwala na stworzenie aplikacji mobilnej na system iOS.

2.2. Git

Git [1] to system kontroli wersji umożliwiający śledzenie zmian w kodzie źródłowym, oraz współprace zespołową przy projektach programistycznych. Jedną z jego głównych funkcji jest zapisywanie kodu źródłowego w etapach, nazywanych tutaj commit-ami. Istnieje także możliwość rozłączania oprogramowania na gałęzie, na których rozwijane są różne funkcjonalności, z możliwością późniejszego ich połączenia z powrotem do głównej gałęzi. Dzięki systemowi Git można także przywrócić dany projekt do swojej poprzedniej wersji, albo mieć ogólny wgląd w jego historię. Podczas prac nad aplikacją mobilną będącą tematem aplikacji, Git zastosowany został w celu zachowania historii zmian.

2.3. Swift

Swift [5] jest językiem programowania stworzonym przez firmę Apple. Głównym celem jego kreacji było zastąpienie Objective-C jako języka używanego podczas tworzenia oprogramowania na platformy Apple. Jest to wysokopoziomowy, kompilowany język zawierający nowoczesne funkcje, takie jak inferencja typów, która ułatwia pisanie kodu oraz zmniejsza szanse na wystąpienie potencjalnych błędów. Język Swift nie wymaga pisania średników na końcach linijek a jego system modułów eliminuje potrzebę tworzenia plików nagłówkowych. Dodatkowo kodowanie ciągów znaków w tym języku bazowane jest na UTF-8, co ułatwia korzystanie ze znaków specjalnych w aplikacji. Pamięć w Swift jest zarządzana poprzez system ARC (ang. Automatic Reference Counting), który automatycznie usuwa niewykorzystywane zasoby z pamięci.

2.4. SwiftUI

SwiftUI [6] jest biblioteką rozwijaną przez firmę Apple zajmującą się tworzeniem interfejsów użytkownika, Jej zadaniem jest zastąpić bibliotekę UIKit, która do 2019 roku była głównym sposobem na tworzenie aplikacji na systemy iOS oraz iPadOS. Jednak cały czas jest w trakcie rozwoju co sprawia, że UIKit nadal utrzymuje się jako dominująca technologia w tej dziedzinie.

Tworzenie interfejsów za pomocą SwiftUI [7] ma charakter deklaratywny — analogiczny do innych technologii internetowych, takich jak na przykład React. Oznacza to, że deweloper w trakcie tworzenia wyglądu aplikacji opisuje to co ma się na ekranie wyświetlić, a nie w jaki sposób. Dzięki temu przy rozwoju interfejsu użytkownika występuje automatyczna synchronizacja widoku z danymi.

W środowisku SwiftUI każdy element graficzny interfejsu użytkownika musi implementować protokół `View` (tłum. widok). Z tego względu w tej pracy wszystkie komponenty wizualne aplikacji będą określane mianem widoków.

Makra zapewniają reaktywność z minimalną dodatkową konfiguracją: `@State` pozwala na przechowywanie zmiennej lokalnej, na której zmianę interfejs będzie reagował automatycznie, `@Binding` pozwala widokowi na otrzymanie dostępu do zmiennej wraz z możliwością jej modyfikacji. Takie podejście jest często stosowane przy polach tekstowych.

Kolejnym przydatnym makrem jest `@Observable` używane w celu wyodrębniania stanu aplikacji. Pozwala ono na utworzenie obiektu, który będzie automatycznie śledził zmiany każdych z jego atrybutów i aktualizował interfejs w przypadku ich nastąpienia. Jest to uproszczeniem w porównaniu do poprzedniego rozwiązania, gdzie taki obiekt musiałby dziedziczyć protokół `StateObject`. Ponadto każdy z jego atrybutów musiałby zostać oznaczony makrem `@Published`, by interfejs reagował na ich zmiany. Rozwiązanie to mniej efektywnie wykorzystuje zasoby systemu, ponieważ jednego parametru wywoływała ponowne renderowanie każdego elementu interfejsu korzystającego z tego obiektu. Natomiast używając makra `@Observable` renderowane są tylko te części interfejsu, gdzie faktycznie nastąpiła jakaś zmiana.

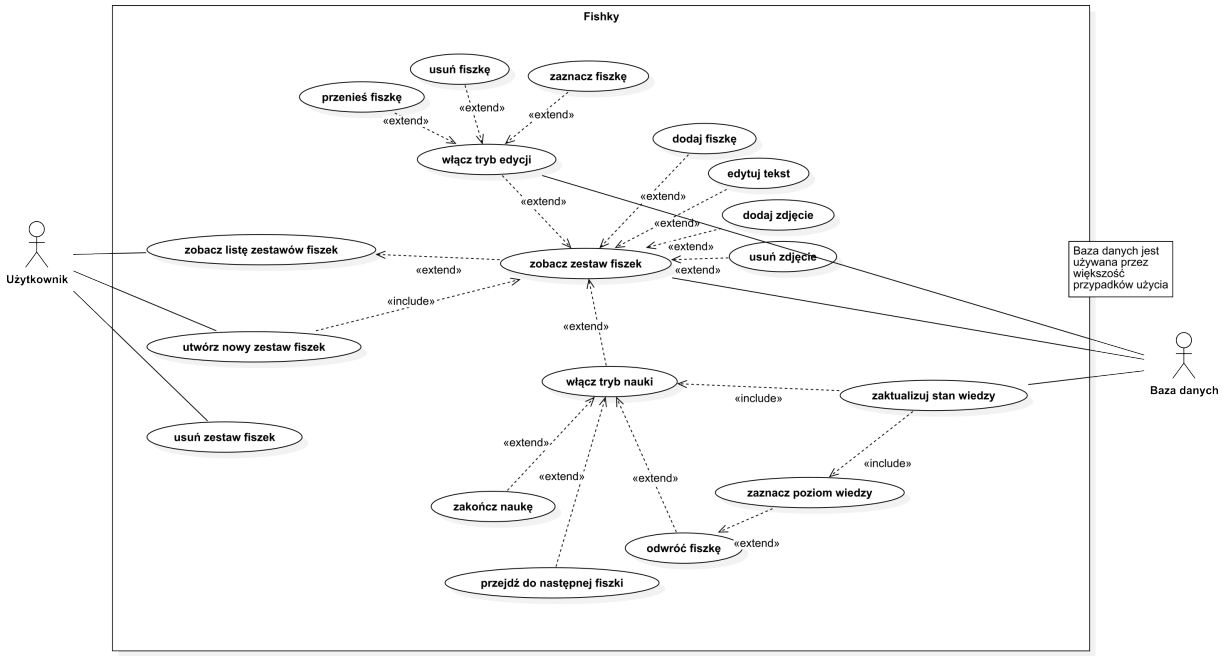
2.5. SwiftData

SwiftData to biblioteka udostępniona przez firmę Apple umożliwiająca modelowanie danych przechowywanych wewnątrz aplikacji. Architektura opiera się o wcześniejszą bibliotekę Apple spełniającą tą rolę: Core Data, która była wrapperem ORM wokół bazy danych SQLite. Główną zaletą Core Data była jej skuteczna integracja z istniejącymi w Objective-C typami. Mimo że istnieje możliwość użycia jej podczas budowania aplikacji w języku Swift, nie wspiera ona wielu nowoczesnych funkcji językowych, takich jak na przykład wsparcie generyków. Z tego powodu biblioteką użytą w trakcie pracy został pakiet SwiftData.

Rozdział 3

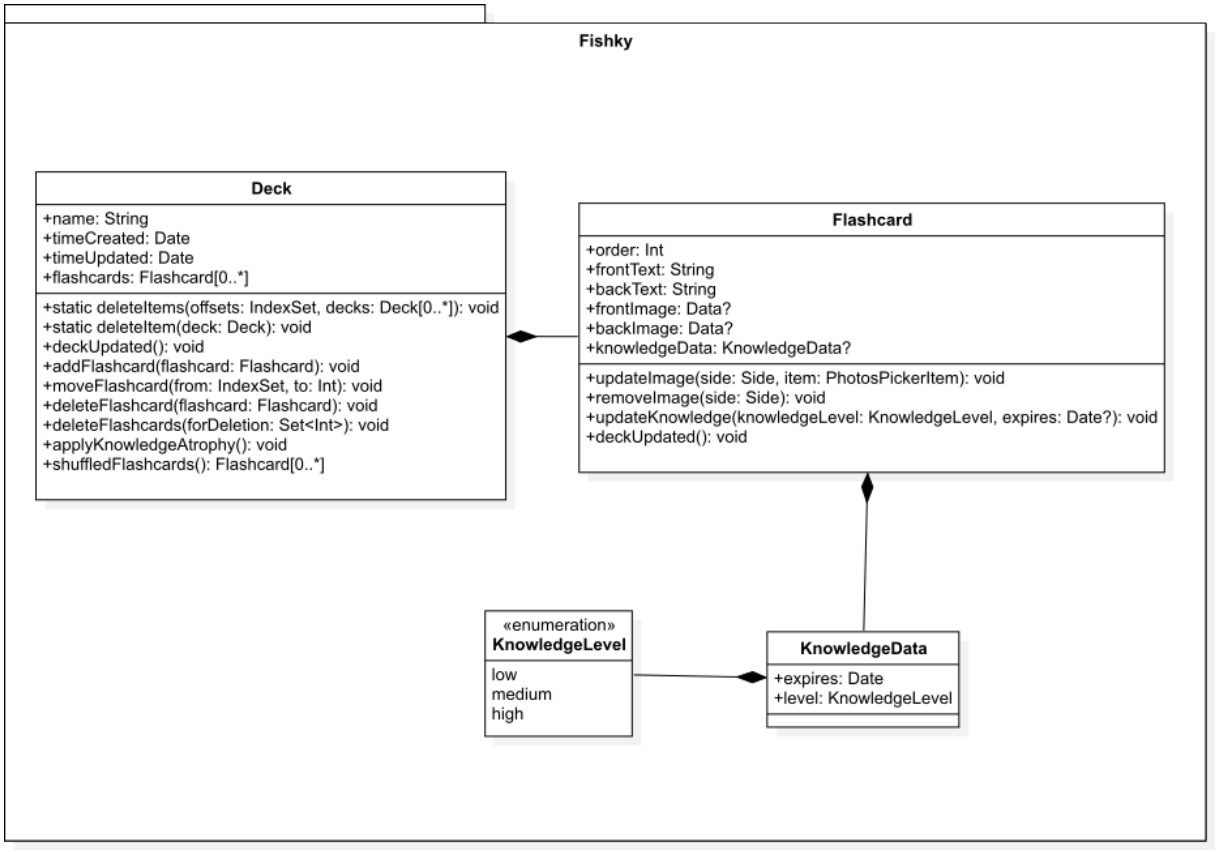
Diagramy UML

3.1. Diagram przypadków użycia



Rysunek 3.1: Diagram przypadków użycia

3.2. Diagram klas

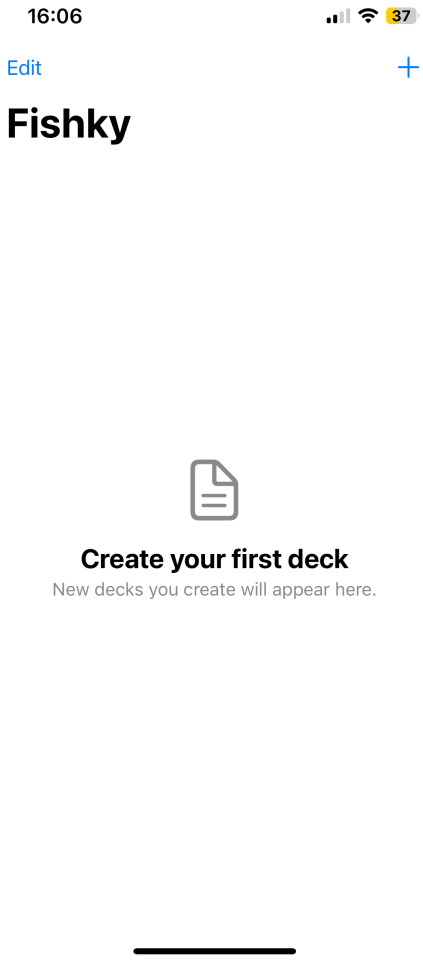


Rysunek 3.2: Diagram klas

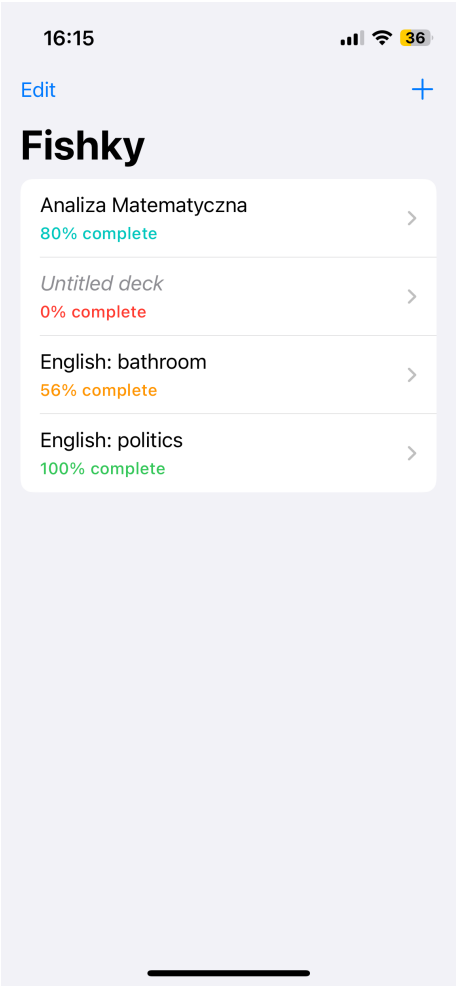
Rozdział 4

Struktura aplikacji

4.1. Ekran startowy



Rysunek 4.1: Ekran startowy bez żadnych zestawów fiszek



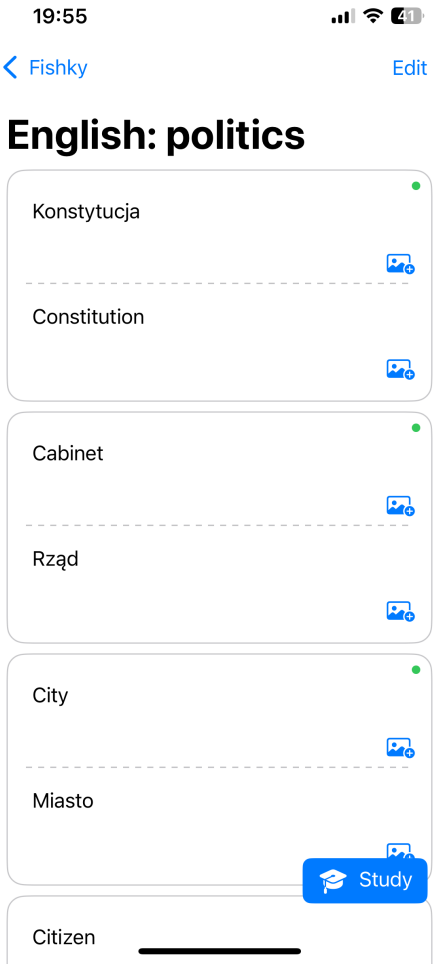
Rysunek 4.2: Ekran startowy z zestawami fiszek

Otwierając aplikacje użytkownik widzi ekran główny będący listą zestawów fiszek. W przypadku, gdy jest to pierwsze uruchomienie programu na środku ekranu znajduje się napis informujący go o tym, że w tym miejscu pojawią się utworzone przez użytkownika zestawy. Każdy zestaw na liście posiada swój poziom nauczenia, czyli procent obliczany na podstawie tego na ile fiszek użytkownik udzielił poprawnych odpowiedzi. Poziomy

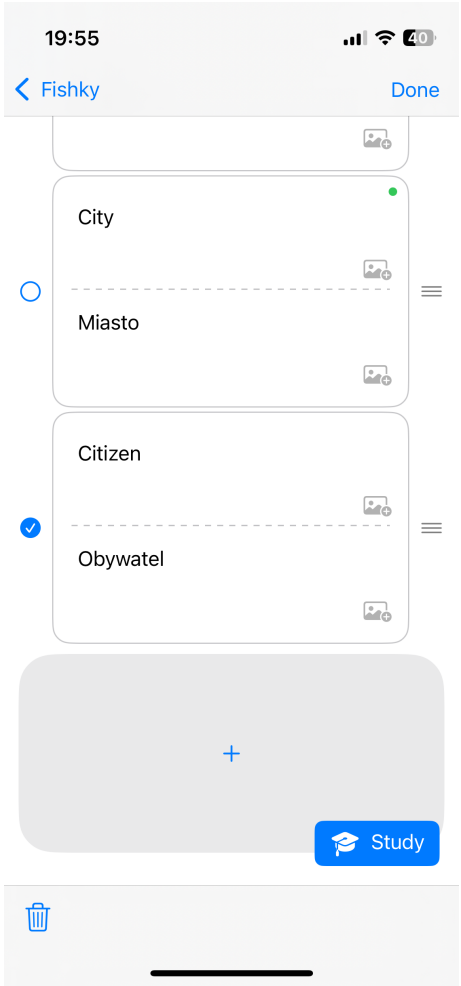
oznaczone są różnymi kolorami: od czerwonego, który oznacza zerowy procent wiedzy do zielonego, który informuje użytkownika o przekroczeniu progu 95 procent.

W górnej części aplikacji widnieje jej tytuł a tuż nad nim na pasku narzędzi znajduje się guzik „Edit", którego funkcją jest włączenie trybu edycji. Po drugiej stronie, w prawym górnym rogu ekranu znaleźć można guzik z ikoną plus umożliwiający dodanie nowego zestawu fiszek.

4.2. Ekran edycji zestawu fiszek



Rysunek 4.3: Zestaw fiszek



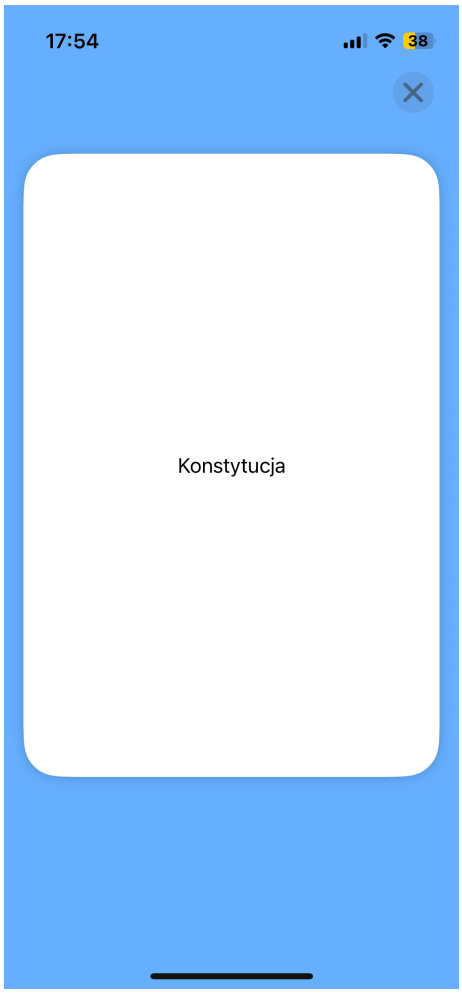
Rysunek 4.4: Zestaw fiszek w trybie edycji

Po utworzeniu nowego zestawu fiszek użytkownik jest przekierowywany na stronę jego edycji. Na samej górze znajduje się pasek narzędzi, po którego lewej stronie widoczny jest, zgodnie z konwencjami znanymi w systemie iOS, guzik cofania umożliwiający przejście do poprzedniego ekranu. Po drugiej stronie paska znajduje się przycisk po naciśnięciu, którego włącza się tryb edycji fiszek. Domyślną nazwą zestawu jest „Untitled deck" oznaczona szarym kolorem. Po kliknięciu na nią użytkownik ma możliwość zmiany tytułu, który przybiera wtedy kolor czarny. Poniżej nazwy znajduje się szary guzik z ikoną plusa, który

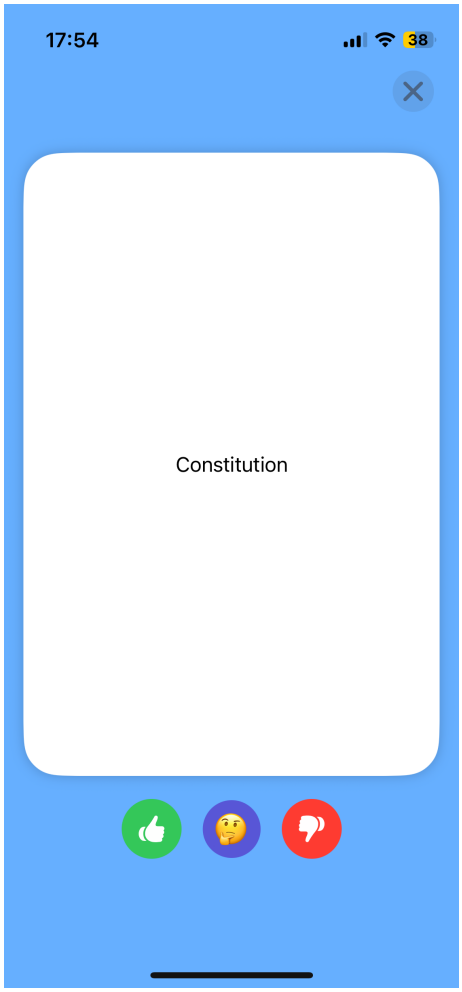
udostępnia funkcję dodania nowej fiszki do danego zestawu. W prawym dolnym rogu ekranu znajduje się także niebieski przycisk do aktywacji trybu nauki.

Każda fiszka posiada swój przód oraz tył znajdujące się na jednym kafelku. Oddzielone są przerywaną linią, by użytkownik miał możliwość jednoczesnej edycji obu stron jednocześnie. Na karcie znajdują się także dwa guziki umożliwiające dodanie zdjęcia do każdej ze stron. Po zakończeniu trybu nauki danego zestawu fiszek w prawym górnym rogu każdego kafelka pojawia się kropka, która swoim kolorem informuje użytkownika o poziomie wiedzy danej fiszki.

4.3. Tryb nauki



Rysunek 4.5: Tryb nauki



Rysunek 4.6: Tryb nauki po odwróceniu fiszki

Włączenie trybu nauki powoduje przejście do pełnoekranowego widoku z guzikiem umożliwiającym jego zamknięcie w prawym górnym rogu. Na środku ekranu znajduje się przednia strona fiszki zawierająca tekst wraz z opcjonalnym zdjęciem. Kliknięcie na nią powoduje rozpoczęcie animacji zamiany karty na jej drugą stronę, gdzie użytkownik może

zobaczyć tekst oraz potencjalne zdjęcie. Wraz z przejściem na drugą stronę fiszki pod nią pojawiają się trzy guziki pozwalające na wybranie poziomu wiedzy danej fiszki. Pierwsza z nich przyjmuje postać kciuka w dół — wiedza niewystarczająca, druga postać emotikony myślenia — wiedza niepełna, a ostatnia to kciuk w górę — odpowiedź użytkownika była poprawna. Sposób przejścia pomiędzy fiszkami jest analogiczny do przewijania listy elementów – należy wykonać gest przesunięcia po ekranie w górę lub w dół. Po przejściu przez wszystkie fiszki wyświetlona zostaje karta informująca o dotarciu do końca zestawu, w której dolnej sekcji widoczny jest duży przycisk umożliwiający zamknięcie trybu nauki.

4.4. Tryb edycji

W aplikacji zaimplementowane zostały funkcje edycji w dwóch wspomnianych już miejscach: lista zestawów fiszek oraz widok szczegółów danego zestawu. Na ekranie głównym daje on możliwość łatwego usuwania uprzednio dodanych zestawów fiszek. Natomiast w drugim przypadku posiada on większą liczbę funkcjonalności jakimi są zmiana kolejności fiszek, zaznaczenie oraz usuwanie wielu kart naraz.

Rozdział 5

Struktura projektu

Bazowa struktura projektu została wygenerowana przy pomocy zestawu narzędzi Xcode. Następnie w trakcie rozwijania projektu pliki zostały podzielone na odpowiednie foldery. Takie postępowanie jest zgodne z konwencją tworzenia projektów w języku Swift oraz sprawia, że poszczególne pliki są łatwe do zlokalizowania.

5.1. Folder `SwiftData`

Zapleczem całej aplikacji, jeśli chodzi o przechowywanie danych jest folder `SwiftData`. W nim zdefiniowane są modele pozwalające na zapisywanie wszystkich danych do pamięci trwałej. Każdy z modeli posiada odpowiednie dla siebie metody, pozwalające widokom na dokonywanie zmian bez bezpośredniego modyfikowania bazy danych. `SwiftData` umożliwia także automatyczne przesyłanie kopii zapasowej danych użytkownika na jego konto iCloud, co może być pomocne w przypadku, gdy aplikacja zainstalowana została na kilku urządzeniach.

```
@Model
final class Deck: Hashable, CustomStringConvertible, Identifiable {
    var name: String = ""
    var icon: String = ""
    var timeCreated: Date = Date.now
    var timeUpdated: Date = Date.now

    @Relationship(deleteRule: .cascade, inverse: \Flashcard.deck)
    var flashcards = [Flashcard]()

    init(name: String = "", icon: String = "") {
        self.name = name
        self.icon = icon
    }
```

Rysunek 5.1: Fragment klasy Deck

Definicje modeli znajdują się w plikach `Flashcard.swift` oraz `Deck.swift`. W pierwszym z nich przechowywany jest obiekt reprezentujący model fiszki w bazie danych, wraz z zestawem metod go modyfikujących. Dodatkowo znajduje się tam struct `KnowledgeData` wraz z enumem `KnowledgeLevel`, które razem reprezentują poziom wiedzy użytkownika dla danej fiszki: niski, średni, albo wysoki. Metody modyfikujące model fiszki pozwalają na zmianę załączonych zdjęć oraz wiedzy o fiszce, a także aktualizowanie informacji o

ostatnich zmianach w zestawie. Klasa `Flashcard` zawiera w sobie również getter koloru reprezentujący aktualną wiedzę fiszki, co wykorzystywane jest w widoku głównym. Modyfikacja tekstu przedniego oraz tylnego jest dokonywana poprzez przekazywanie `Binding` do zmiennej widokowi edytora tekstu.

```
@Model
final class Flashcard: Hashable, CustomStringConvertible, Ordered, Reorderable {
    enum Side {
        case front, back
    }

    @Attribute(originalName: "index")
    var order: Int

    var frontText: String
    var backText: String
    @Attribute(.externalStorage) var backImage: Data?
    @Attribute(.externalStorage) var frontImage: Data?

    var knowledgeData: KnowledgeData?

    var deck: Deck?
```

Rysunek 5.2: Fragment klasy Flashcard

W pliku `Deck.swift` znajduje się obiekt reprezentujący model zestawu fiszek, który jako swoje metody zawiera możliwości dodawania, usuwania oraz zmieniania kolejności fiszek. Zawarta jest tutaj również metoda nakładająca na zestaw fiszek efekt zaniku wiedzy wraz z upływem czasu. Efekt nakładany jest podczas otwierania danego zestawu w trybie nauki. Kolejną z metod zlokalizowanych w pliku `Deck.swift` jest sortowanie listy fiszek. Metoda nie działa w sposób kompletnie losowy — algorytm najpierw dzieli wszystkie fiszki na cztery sekcje w zależności od poziomu nauczenia, lub jego braku. W każdej z czterech kategorii odbywa się losowanie kolejności a następnie wszystkie fiszki łączone są z powrotem w jedną listę, dzięki temu fiszki słabo pamiętane przez użytkownika znajdują się na początku trybu nauczania. Inną ważną metodą w tym pliku zajmuje się obliczaniem procentu wiedzy w danym zestawie fiszek, czyli średniej ważonej z danych o wiedzy każdej fiszki. Obiekt `Deck` zawiera także statyczne metody służące do dodawania i usuwania zestawów fiszek.

W folderze `SwiftData` znajdują się także pliki `DataModel.swift` oraz `PreviewSampleData.swift`. Pierwszy z nich definiuje globalny obiekt `DataModel` będący singletonem, który przechowuje `ModelContainer`, czyli obiekt zarządzającym modelami bazy danych `SwiftData`. Drugi plik definiuje modyfikator podglądu widoków, aby ułatwić wyświetlanie danych z modeli `SwiftData` w widokach.

```
enum KnowledgeLevel: Int, Codable, CustomDebugStringConvertible {
    var debugDescription: String {
        switch self {
            case .low:
                return ".low"
            case .medium:
                return ".medium"
            case .high:
                return ".high"
        }
    }

    case low = 0
    case medium = 1
    case high = 2

    var oneLevelDown: KnowledgeLevel? {
        switch self {
        case .low:
            return nil
        case .medium:
            return .low
        case .high:
            return .medium
        }
    }
}

struct KnowledgeData: Codable, CustomDebugStringConvertible {
    var debugDescription: String {
        "KnowledgeData(\(expires), \(level))"
    }

    let expires: Date
    let level: KnowledgeLevel
}
```

Rysunek 5.3: Enum oraz struct pozwalające na zapisywanie stanu wiedzy

5.2. Plik `FishkyApp.swift`

`FishkyApp.swift` to plik startowy dla całej aplikacji, który definiuje widok bazowy. Ten widok z kolei tworzy zmienną zawierającą `DataModel`, oraz wywołuje widok główny.

5.3. Folder `Views`

W tym folderze umieszczone są wszystkie widoki SwiftUI. Widoki zostały podzielone na odpowiednie podfoldery w zależności od tego w jakim ekranie są wykorzystywane. Widoki uniwersalne, czyli mające możliwość użycia w wielu miejscach, zostały umieszczone w folderze o nazwie `Support`.

5.4. Folder `Views/Home`

Ten folder reprezentuje widok główny aplikacji. Znajdują się w nim trzy pliki reprezentujące etapy ciągu wywołań na tym ekranie. Plik `ContentView.swift` jest pierwszym z etapów — znajduje się w nim `NavigationSplitView`, który na większych ekranach będzie tworzył widok podzielony na dwie sekcje: listę elementów oraz widok szczegółów zaznaczonego elementu. Na mniejszych ekranach natomiast będzie wyświetlał samą listę elementów, przy czym kliknięcie na jeden z nich wywoła przejście do jego ekranu szczegółów. Tym ekranem zajmują się widoki w folderze `Views/Deck`. Widok listy zestawów fiszek znajduje się w pliku `DeckListView.swift` wywoływanym w `ContentView.swift`.

```
struct ContentView: View {
    @State var currentDeck: Deck?

    var body: some View {
        NavigationSplitView(columnVisibility: .constant(.doubleColumn)) {
            DeckListView(currentDeck: $currentDeck)
                .navigationTitle("Fishky")
        } detail: {
            if let currentDeck {
                DeckView(currentDeck)
            } else {
                Text("Select or create a deck")
            }
        }
    }
}
```

Rysunek 5.4: Widok ContentView

Wewnątrz niego znajduje się kod pobierający listę zestawów fiszek z bazy danych, używając makra `@Query`, a także funkcjonalność dodawania i usuwania ich. Za pomocą operatora `.toolbar` pasek narzędzi wypełniany jest dwoma przyciskami: pierwszy przełączający tryb edycji oraz drugi, z ikonką plusa, tworzący nowy zestaw fiszek. Znajduje się tutaj także logika gestów przesunięcia fiszki w celu jej usunięcia.

Część wizualna każdego z elementów jest zamieszczona w pliku `DeckListItem.swift`. Wyświetla on widok `NavigationLink`, czyli widok elementu używanego w celu przechodzenia do innej strony, a w nim tytuł zestawu fiszek wraz z procentem wiedzy.

5.5. Folder `Views/Deck`

W tym folderze znajdują się wszystkie pliki związane z widokiem zestawu fiszek. Bazowym widokiem jest `DeckView` z pliku `DeckView.swift`. Zarządza on elementami, takimi jak tytuł zestawu, guziki dodawania nowej fiszki, guziki włączenia trybu nauki, oraz dodatkowymi elementami konfiguracyjnymi.

Większa część logiki zawarta została w pliku `FlashcardListView.swift`. Ze względu na to widok ten został rozdzielony na dwie części: część wyświetlającą i część zarządzającą logiką widoku. Do obiektu zarządzającego logiką zostało wykorzystane makro `@Observable`, które umożliwia widokom SwiftUI monitorowanie zmian obiektu i aktualizowanie widoku po ich dokonaniu. Funkcjonalności zawarte w tym obiekcie to możliwość

zmiany kolejności oraz zaznaczenia fiszek w danym momencie. Część wyświetlająca w tym pliku ma za zadanie pokazywać widoki `AdaptiveList` oraz `ReorderableForEach`, które są zdefiniowane w innym folderze. Zapewniają one funkcjonalność listy, która dostosowuje się do wielkości ekranu oraz możliwość zmiany kolejności elementów. Wewnątrz listy przechodzącej po każdej z fiszek w danym zestawie wyświetlany zostaje kafelek edycji lub kafelek podglądu fiszki w zależności od stanu trybu edycji. W pierwszym wypadku tryb edycji jest wyłączony a w drugi włączony. Nie brzmi to intuicyjnie — dlaczego kafelek edycji pojawia się, gdy tryb jest wyłączony? Uzasadnieniem jest fakt, iż podczas przenoszenia fiszki w liście tworzyły się niechciane błędy powodujące wejście wewnętrznego ID danego elementu do jego pola tekstowego. Rozwiązaniem problemu okazała się restrykcja opcji zmiany kolejności tylko do trybu edycji, co pasuje do wzorców znanych w systemie iOS. Dodatkowo podczas zmiany kafelka edycji w kafelek podglądu zaimplementowana została animacja zapewniająca płynne przejście.

```
@Observable
class FlashcardListState {
    private(set) var deck: Deck

    var reorderedFlashcard: Flashcard?
    var reorderInProgress: Bool { reorderedFlashcard != nil }

    private(set) var selectedFlashcards: Set<Int> = []


    init(deck: Deck) {
        self.deck = deck
    }


    func isSelected(flashcard: Flashcard) -> Bool {
        selectedFlashcards.contains(flashcard.order)
    }

    func toggleSelection(_ flashcard: Flashcard) {
        let selectedIndex = selectedFlashcards.firstIndex(of: flashcard.order) ?? nil
        let isSelected = selectedIndex != nil

        if isSelected {
            selectedFlashcards.remove(at: selectedIndex!)
        } else {
            selectedFlashcards.insert(flashcard.order)
        }
    }

    func resetSelection() {
        selectedFlashcards = []
    }

    func deleteFlashcards() {
        withAnimation {
            logger.info("delete flashcards")
            deck.deleteFlashcards(selectedFlashcards)
            selectedFlashcards.removeAll()
        }
    }
    func deleteFlashcard(_ flashcard: Flashcard) {
        logger.info("delete flashcard")
        deck.deleteFlashcard(flashcard)
    }

    func moveFlashcard(from: IndexSet, to: Int) {
        deck.moveFlashcard(from: from, to: to)
    }
}
```

Rysunek 5.5: Obiekt stanu w pliku FlashcardListView.swift

Kafelek edycji zawarty jest w pliku `FlashcardEditTile.swift`. Posiada on dwa pola tekstowe oddzielone linią przerywaną — pod każdym z nich są załączone zdjęcia lub przyciski ich dodawania. Większość tego widoku zawarta jest w iinnych plikach aplikacji. Sekcja ciała widoku jest złożona z pionowej listy elementów otoczonej szarą ramką , która składa się z edytora tekstu z pliku znajdującego się w folderze `Support`. Kolejnym jego elementem jest widok `FlashcardPickerOrImage` z pliku `FlashcardImage.swift`, niżej znajduje się linia przerywana pliku, widok `FlashcardPickerOrImage` oraz edytor tekstu.

Widok zajmujący się wyświetlaniem zdjęć w pliku `FlashcardImage.swift` zawiera także funkcjonalność wyświetlaniem guzika dodawania zdjęcia. Taka struktura została wybrana, aby każdy etap zarządzania zdjęciami był zmieszczony w tym widoku, a do bazy danych zapisywane zostały tylko ostateczne zdjęcia w formacie binarnym. W tym pliku stan widoku został wydzielony do osobnego obiektu i tak samo jak poprzednio jest on definiowany przy pomocy makro `@Observable`. Ten stan przechowuje w sobie referencję do obiektu danej fiszki w bazie danych, informację o stronie fiszki (przód czy tył) w formie enum oraz obiekt `PhotosPickerItem`. Obiekt ten jest zwracany przez wbudowany widok `PhotosPicker` i przechowuje w sobie tymczasowo wybrane zdjęcie. W pliku `FlashcardImage.swift` obecne są również gettery upraszczające logikę samego widoku: jeden zwraca odpowiednie dane binarne zdjęcia w zależności od strony fiszki, drugi konwertuje dane binarne na reprezentację możliwą do użycia w interfejsie użytkownika. Znajdują się w nim także dwie metody do zarządzania zdjęciami w pamięci. Pierwsza z nich usuwa zdjęcia z fiszki, natomiast druga dodaje zdjęcia zamieniając je na format binarny. Zamiana ta zamknięta jest w bloku `Task`, ponieważ operacje związane z konwersją formatu zdjęcia na binarny są asynchroniczne. Wymagają więc wyodrębnienia jako osobne „zadanie". W sekcji wyświetlającej ten widok znajduje się instrukcja warunkowa, która zajmuje się wyświetlaniem dostępnego zdjęcia lub, w razie jego braku, wyświetlaniem przycisku jego dodawania.

Pozostałe pliki w tym folderze to `NewFlashcardButton.swift`, gdzie znajduje się definicja guzika dodawania nowych fiszek, oraz `FlashcardPreviewTile.swift`, który definiuje podgląd `FlashcardEditTile` z guzikami o szarej barwie. Kolor ten został użyty w celu podkreślenia brak możliwości interakcji.

5.6. Folder `Views/Study`

Po kliknięciu guzika włączania trybu nauki otwiera się pełnoekranowy widok nauki. Zarządza nim folder `Views/Study`. Cały widok jest podzielony na trzy części: lista fiszek wyświetlana w pliku `FullscreenStudyView.swift`, widok pojedynczej fiszki w pliku `FlashcardStudyView.swift`, oraz guziki do zaznaczania znajomości materiału naukowego w pliku `KnowledgeButtons.swift`.

W pliku `FullscreenStudyView.swift` widok jest podzielony na obiekt stanu i obiekt wyglądu. Obiekt stanu posiada pięć zmiennych: odnośnik do zestawu fiszek w bazie danych, listę fiszek dla danej sesji, listę zapisującą stan odwrócenia fiszek, listę zapisującą widoczność przycisków wiedzy przy fiszkach, oraz zmienną oznaczająca oczekiwanie na przygotowanie wszystkich innych zmiennych. Ostatnia z nich zmienia się podczas wywo-

ływania funkcji `initialize`, która bierze z bazy danych fiszki i miesza je bazując na stanie wiedzy. Oprócz tego w obiekcie stanu znajdują się również dwie metody. Jedna z nich zajmuje się animacją obracania danej fiszki, przy czym odpowiednio opóźnia animację pojawiania się guzików, aby zapewnić lepsze zgranie elementów interfejsu. Druga metoda zwraca widoczność guzików dla odpowiedniego indeksu elementu. W sekcji wyglądu widoku wyświetlany jest widok przesuwalny (`ScrollView`). Nałożone na niego odpowiednie modyfikatory zapewniają, iż przewijanie odbywa się po jednej fiszce naraz. Pod-widokami widoku przesuwalnego są widoki nauki fiszki, których częścią są widoki przycisków wiedzy. Nałożone na nie zostały specjalne modyfikatory zapewniające wypełnienie całej dostępnej na ekranie przestrzeni. Dodatkowo na samym jego końcu umieszczony jest widok końca nauki przypominający fiszkę, który zawiera w sobie przycisk do zakończenia trybu nauki.

```
struct KnowledgeButtons: View {
    @Bindable var flashcard: Flashcard
    @State var newKnowledge: KnowledgeLevel? = nil

    var body: some View {
        return HStack {
            Button {
                updateKnowledge(.high, for: flashcard)
            } label: {
                Image(systemName: "hand.thumbsup.fill")
                    .font(.system(.title2))
                    .padding(.all, 5)
            }.buttonStyle(.borderedProminent)
                .buttonBorderShape(.circle)
                .tint(newKnowledge != nil && newKnowledge != .high ? .gray : .green)
            Button {
                updateKnowledge(.medium, for: flashcard)
            } label: {
                Text("🤔")
                    .font(.system(.title))
            }.buttonStyle(.borderedProminent)
                .buttonBorderShape(.circle)
                .tint(newKnowledge != nil && newKnowledge != .medium ? .gray : .indigo)
            Button {
                updateKnowledge(.low, for: flashcard)
            } label: {
                Image(systemName: "hand.thumbsdown.fill")
                    .font(.system(.title2))
                    .padding(.all, 5)
            }.buttonStyle(.borderedProminent)
                .buttonBorderShape(.circle)
                .tint(newKnowledge != nil && newKnowledge != .low ? .gray : .red)
        }.sensoryFeedback(.impact, trigger: newKnowledge)
    }
```

Rysunek 5.6: Fragment widoku KnowledgeButtons

Widok nauki fiszki znajduje się w pliku `FlashcardStudyView.swift`. Zawiera on w sobie dwa widoki: widok pojedynczej karty, oraz widok pełnej fiszki. Widok karty dostaje jako argumenty treść danej strony fiszki, dane ewentualnego zdjęcia, oraz stopień obrócenia. Na kartę zostaje nałożony efekt obrotu trójwymiarowego zgodny z podanymi stopniami. Sama karta składa się z zaokrąglonego prostokąta z nałożonym na niego tekstem i ewentualnym zdjęciem. Dodatkowo nałożone zostały na nią modyfikatory wyglądu — kolor biały, proporcje 2:3 oraz cień. Wewnątrz widoku pełnej fiszki utrzymywany jest stan rotacji oraz stopień obrotu zarówno przedniej jak i tylnej strony. Obie strony karty reprezentowane są poprzez oddzielne nałożone na siebie widoki widoczne tylko, gdy ich stopień obrócenia na to pozwala. Po kliknięciu w kartę najpierw wywołuje się animacja

obrotu według jej osi pionowej. To sprawia, że karta przestaje być widoczna na ekranie — prawie jakby się „schowała". Po zakończeniu obrotu wywoływany jest kolejny obrót, tym razem karty o przeciwnej stronie fiszki od obecnie niewidocznej, co sprawia, iż pojawia się ona na ekranie. Płynność obu animacji daje złudzenie istnienia jednej karty wypełnionej treścią po obu stronach. Cały system obrotu kart bazowany jest na artykule Jonathana Rasmussona [3] udostępnionym na platformie GitHub.

Przyciski zawarte w pliku `KnowledgeButtons.swift` są złożone z trzech guzików w kształcie koła, z kolorami zależnymi od stanu guzika — wybrany zachowuje swój kolor, podczas gdy reszta staje się szara.

5.7. Folder `Views/Support/`

Widok edytora tekstu zawarty jest w pliku `TextEditorView.swift`. Biblioteka `SwiftUI` posiada wbudowany edytor tekstu `TextEditor`, jednak nie mógł on być zastosowany samodzielnie w tej aplikacji — powodował on błędy układu struktury strony. Przykładem mogą być dwa pola tekstowe reprezentujące treści po obu stronach fiszki. Podczas zmiany wysokości jednego pola drugie z nich automatycznie dostosowywało się do takiej samej wysokości. Efektem tego był mało estetyczny wygląd fiszek. Przy rozwiązaniu tego problemu pomocny okazał się użytkownik portalu Stack Overflow [11]. Zaproponowane rozwiązanie zakładało nałożenie na siebie `TextEditor`, niewidzialnego widoku normalnego tekstu oraz `GeometryReader`, którego zadaniem jest odczytywanie i egzekwowanie wysokości docelowej widoku tekstu.

Urządzenia mają różne rozmiary — na mniejszych z nich ułożenie elementów w listę może być bardziej sensowne, a na innych w siatkę. Do tego właśnie celu został utworzony widok zawarty w pliku `AdaptiveList.swift`. Widok ten ma możliwość dostać dowolną listę widoków i na podstawie klasy rozmiaru urządzenia wyświetli elementy w pasującym do niej układzie.

W pliku `ReorderableForEach.swift` znajduje się widok umożliwiający zmianę kolejności elementów. Jest on dość mocno inspirowany artykułem Daniela Saidi „Enabling drag reordering in lazy SwiftUI grids and stacks"[14]. Widok wykorzystuje modyfikatory `onDrag` oraz `onDrop`, aby wykrywać podnoszenie, upuszczanie, oraz zmianę pozycji elementów przenoszonych. Dodatkowo zawarty w pliku został rownież delegat upuszczania i przenoszenia elementu, które z wykorzystaniem wyżej wspomnianych modyfikatorów zarządzają stanem przenoszenia.

Podsumowanie

Celem pracy było stworzenie minimalistycznej aplikacji do nauki za pomocą fiszek. Rezultatem jest aplikacja na system iOS spełniająca wszystkie określone na początku pracy wymogi. Pozwala na zapełnianie tekstem obu stron fiszek, a także dodawanie zdjęć.

Aplikacja wyróżnia się na tle konkurencji minimalistycznym interfejsem bez zbędnej funkcjonalności zaciemniającej jej cel dydaktyczny. Intuicyjny układ zapewnia prostotę w użytku, a użytkownik jest na bieżąco informowany o jego postępach w nauce. Ponadto tryb nauczania z inteligentnym układaniem fiszek dostosowuje się do aktualnego poziomu wiedzy użytkownika w celu optymalizowania efektów nauki.

Aplikacja w obecnej wersji stanowi solidną podstawę do dalszego rozwoju. W perspektywie planowanych modyfikacji jest zapewnienie użytkownikowi możliwości robienia zdjęć wewnątrz aplikacji oraz przycinania ich. Dzięki dużym możliwością biblioteki SwiftUI aplikacja może zostać zaadaptowana do działania na komputerach z systemem macOS. Następnym krokiem mogłaby być też adaptacja na kolejne platformy ekosystemu Apple, aby możliwość nauki była dostępna na systemach watchOS oraz tvOS. W przypadku aplikacji działającej na tylu platformach potrzebna, by była także funkcjonalność synchronizacji aplikacji z chmurą iCloud, co jest jedną z funkcjonalności wbudowanych w bibliotekę SwiftData. Dalszy rozwój aplikacji jest możliwy, aktualny stan jest satysfakcjonujący i spełnia wszystkie założone oczekiwania.

Bibliografia

[1] Scott Chacon and Ben Straub. Git book. https://git-scm.com/book/en/v2, 2025.

[2] cqg2 (https://github.com/cqg2) Damien Elmes (https://github.com/dae), Expertium (https://github.com/Expertium). What spaced repetition algorithm does Anki use? https://faqs.ankiweb.net/what-spaced-repetition-algorithm.html, 2024.

[3] Jonathan Rasmusson (https://github.com/jrasmusson). Flipping a Card – GitHub. https://github.com/jrasmusson/swiftui/tree/main/Animations/CardFlip, 2025.

[4] Yoshinobu Ikeda. Flashcards - study flash cards on the App Store. "https://apps.apple.com/pl/app/flashcards-study-flash-cards/id1462174312", 2025.

[5] Apple Inc. Swift - Apple Developer. https://developer.apple.com/swift/, 2025.

[6] Apple Inc. Swift - Apple Developer. https://developer.apple.com/xcode/swiftui/, 2025.

[7] Apple Inc. Swift - Apple Developer. https://developer.apple.com/documentation/SwiftUI, 2025.

[8] Apple Inc. Xcode - apple developer. https://developer.apple.com/xcode/, 2025.

[9] Quizlet Inc. About quizlet. https://quizlet.com/mission, 2025.

[10] Quizlet Inc. Quizlet: Study with Flashcards on the App Store. https://apps.apple.com/pl/app/flashcards-study-flash-cards/id1462174312, 2025.

[11] jnpdx. swift - Controlling size of TextEditor in SwiftUI - Stack Overflow. https://stackoverflow.com/a/69002976, 2025.

[12] Ankitects Pty Ltd. AnkiMobile Flashcards on the App Store. https://apps.apple.com/us/app/ankimobile-flashcards/id373493387, 2025.

[13] Toshiki Motomura. Flashcards - study flash cards on the App Store. https://apps.apple.com/pl/app/flashcards-study-flash-cards/id1462174312, 2025.

[14] Daniel Saidi. Enabling drag reordering in lazy SwiftUI grids and stacks. https://danielsaidi.com/blog/2023/08/30/enabling-drag-reordering-in-swiftui-lazy-grids-and-stacks, 2025.