Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Kierunek: **INA**
Specjalność: **-**

Praca Dyplomowa Inżynierska

Serwis do przeprowadzania rozgrywek Rummikub

Adam Bednarz

Opiekun pracy
dr inż. Jakub Lemiesz

Wrocław 2022

Streszczenie

Tutaj tekst streszczenia po polsku.

Abstract

Tutaj treść streszczenia po angielsku.

Spis treści

Wstęp

Celem pracy jest zaprojektowanie i implementacja serwisu informatycznego, który umożliwia przeprowadzanie gier Rummikub.

Założenia funkcjonalne serwisu:

- autoryzacja użytkowników,
- zarządzanie instancjami gier,
- korzystanie z serwisu za pomocą strony internetowej lub aplikacji na telefon,
- rozgrywka z botem.

Obecnie istnieją już takie rozwiązania, które dostarczają nam podobne funkcjonalności. Jednakże motywacją tej pracy jest bliższe zapoznanie się z procesem tworzenia takiego serwisu, jak również analiza złożoności gry Rummikub. Czynnikiem wyróżniającym się stworzonego serwisu jest umożliwienie graczom uczestniczenie w danej rozgrywce niezależnie od tego czy korzystają z aplikacji na telefon (iOS, Android), czy też ze strony internetowej w przeglądarce.

Praca składa się z czterech rozdziałów. W rozdziale pierwszym omówiono grę Rummikub, jej zasady oraz przebieg rozgrywki. W rozdziale drugim zobrazowano szczegółowy projekt serwisu. Do graficznego przedstawienia wykorzystano notacje UML. W rozdziale trzecim opisane zostały użyte technologie, które zostały użyte w projekcie informatycznym wraz z diagramami UML. W rozdziale czwartym przedstawiono dokumentację techniczną systemu. W rozdziale piątym przedstawiono kwestie implementacji bota w grze Rummikub.

Rozdział 1

Opis gry Rummikub

1.1 Gra

Została stworzona przez Ephraima Hertzano i po raz pierwszy wydano ją w 1950 roku. W rozgrywce może uczestniczyć od dwóch do czterech graczy. Gra składa się z kości, które są numerowane od 1 do 13. Występują one w czterech kolorach (pomarańczowy, czerwony, niebieski, czarny). Każda kość o danym kolorze i liczbie występuje dwa razy. Ponadto występują dwie specjalne kości jako jokery. Łącznie gra zawiera 106 kości.

Gra polega na wykładaniu grup, bądź serii. Grupa to trzy lub cztery kości o różnych barwach, ale z tą samą liczbą, natomiast seria to co najmniej trzy kolejne kości o tym samym kolorze.

W przypadku braku odpowiednich kości do gry można posłużyć się dwoma zestawami kart po 52 karty i 2 jokery. Gdzie 1, 11, 12, 13 można zastąpić odpowednio asem, waletem, damą i królem.

1.2 Rozgrywka

Zbiór wszystkich wymieszanych kości nazywany jest bankiem. Z niego na początku gry każdy gracz otrzymuje 14 kości. W przypadku pierwszego ruchu należy wyłożyć własne kości o sumie numerów tych kości co najmniej 30, bez możliwości modyfikowania kości leżących na planszy.

Po wyłożeniu pierwszego ruchu gracz może modyfikować inne wyłożone wcześniej układy. Dozwolone jest przebudowywanie układów kości (rozbijanie lub rozbudowywanie). Jednak w każdym ruchu należy wyłożyć przynajmniej jedną własną kość.

Na każdy ruch przypada ustalony wcześniej limit czasowy. Po jego upływie w przypadku braku wyłożenia kości przez gracza, lub gdy stan planszy nie spełnia zasad gry, gracz pobiera z banku jedną kość i cofa wprowadzone zmiany układów kości.

Joker w grze Rummikub symbolizuje dowolną kość. Można nim zastąpić brakujący element do utworzenia układu kości. Joker ma taką samą wartość jak kość, którą zastępuje. Można zabrać wyłożonego jokera zastępując go odpowiednią kością i wykorzystać go w innym miejscu na planszy.

Gra kończy się w momencie, gdy któryś z graczy wyłoży wszystkie swoje kości lub gdy zabraknie kości w banku. W przypadku drugim wygrywa ten gracz, który ma najmniejszą sumę liczb znajdujących się na kościach.

Rozdział 2

Projekt serwisu

W tym rozdziale przedstawiono szczegółowy projekt systemu w notacji UML uwzględniający wymagania funkcjonalne opisane we wstępie. Do opisu relacji pomiędzy składowymi systemu wykorzystano diagramy UML.

2.1 Struktura

W serwisie wykorzystano architekturę klient-serwer. Istnieje jeden serwer, do którego może podłączać się wiele klientów i odpowiada on za komunikację i zarządzanie danymi. W komponencie klienta wykorzystano architekturę trójwarstwową. Architektura ta dzieli komponent na trzy osobne części:

- warstwa prezentacji,
- warstwa biznesowa,
- warstwa danych.

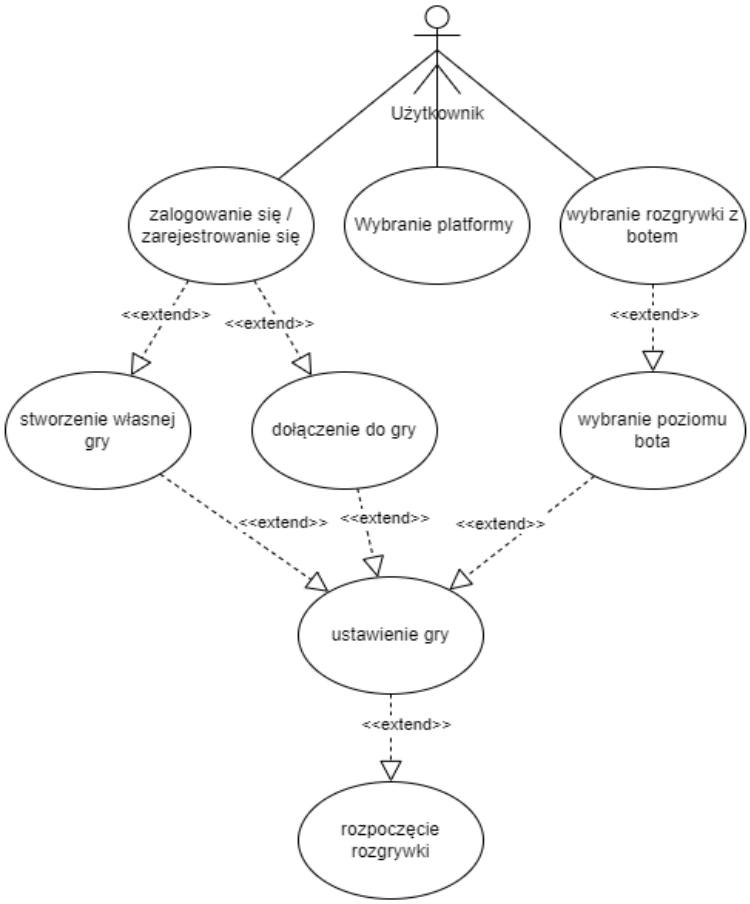
Warstwa prezentacji jest to interfejs graficzny użytkownika. Jest odpowiedzialna za interakcję z użytkownikiem (wyświetlanie i wprowadzanie danych).

Warstwa biznesowa odpowiada za przetwarzanie komunikatów od użytkownika lub ze strony serwera. Tutaj zawarta jest wszelka logika aplikacji. Przetworzone dane są przekazywane do warstwy prezentacji i/lub warstwy danych. Warstwa ta jest łącznikiem pomiędzy warstwą prezentacji, a warstwą danych.

Warstwa danych jest dostępem do danych. Obsługuje połączenie aplikacji z zewnętrznym obiektem dostarczającym dane (baza danych, serwer).

2.2 Przypadki użycia

Użytkownik może wybrać na jakiej platformie będzie chciał korzystać z serwisu. Ma do wyboru stronę internetową albo aplikacje na telefon. Aby móc dołączyć do gry z innymi graczami należy zalogować się lub zarejestrować w aplikacji. Następnie gracz ma możliwość stworzenia własnej gry lub dołączenia do innej. Jeśli zdecyduje się stworzyć własną grę, musi wybrać od 2 do 4 osób spośród wszystkich aktywnych graczy. Dalej przechodzi do panelu konfiguracji gry. Wybiera dozwolony czas na wykonanie ruchu oraz liczbę graczy, jeśli zdecydował się na dołączenie do innej gry. Dołączenie do innej gry działa na zasadzie znalezienia oczekującej gry o takich samych ustawieniach jakie wprowadził gracz. W przypadku nie znalezienia takiej gry jest tworzona nowa gra. Jeśli wszystkie miejsca w grze zostały wypełnione rozgrywka się rozpoczyna. W przypadku wybrania z gry z botem, odbywa się ona lokalnie bez łączenia z serwerem. Użytkownik ma do dyspozycji dwa poziomy bota. Następnie również konfiguruje grę i rozpoczyna rozgrywkę.



Rysunek 2.1: Diagram przypadków użycia.

2.3 Diagramy aktywności

2.4 Diagramy stanów

Rozdział 3

Opis technologii

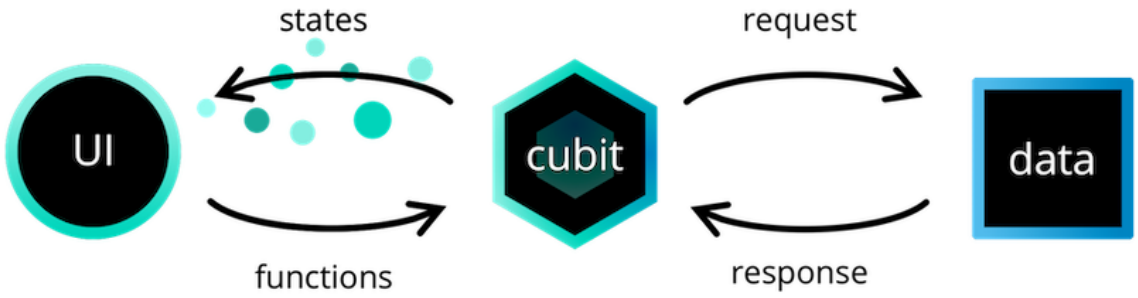
W rozdziale tym wskazano jakie podejście technologiczne zostało użyte wraz z omówieniem i zaargumentowaniem powodu takiego wyboru. Przedstawiono również jakie konkretne aspekty tych technologii zostały wykorzystane w projekcie.

3.1 Flutter

Do implementacji serwisu po stronie klienta użyto technologii Flutter. Jest to zestaw narzędzi pozwalający tworzyć natywne, wieloplatformowe aplikacje mobilne, komputerowe oraz internetowe. Flutter stworzony jest przez firmę Google, a jego pierwsza stabilna wersja ukazała się pod koniec 2019 roku. Mimo stosunkowo młodej technologii Google przyczynia się do jej dynamicznego rozwoju i zdobywania popularności wśród programistów. Aplikacje we Flutterze pisze się w języku Dart. Jest to zorientowany obiektowo, statycznie typowany, wysokopoziomowy język programowania. Składnia języka Dart wzorowana była na takich językach programowania jak Java czy C#, aby ułatwić programistom naukę nowego języka.

3.1.1 Architektura

Flutter domyślnie nie oddziela kodu dotyczącego sposobu działania programu od części wizualnej. Chcąc wyodrębnić aspekty wizualne od aspektów logicznych aplikacji, należy użyć wybranego wzorca architektury. Istnieje wiele pakietów pomagających zaimplementować poszczególne wzorce. W tym projekcie zastosowano rekomendowany przez Google pakiet BLoC, który umożliwia implementacje architektury trójwarstwowej z podziałem na warstwę wizualną, logiki i danych, jak również wprowadza system zarządzania stanami w aplikacji.



Rysunek 3.1: Model struktury wzorca projektowego BLoC.

BLoC (Business Logic Component) - wzorzec projektowy pomagający oddzielić elementy wizualne projektu od części logiki działania programu. Dzieli projekt na trzy główne komponenty UI, cubit, data.

Warstwa UI jest odpowiedzialna za część wizualną aplikacji, cubit jest warstwą zawierającą mechanizmy działania aplikacji oraz pomostem pomiędzy interfejsem użytkownika, a zewnętrznymi danymi, natomiast warstwa data komunikuje się z zewnętrznymi instancjami (serwer, baza danych).

Cubit komunikuje się z warstwą danych poprzez funkcje asynchroniczne. Natomiast chcąc wpłynąć na zmianę wyglądu aplikacji korzysta z programowania reaktywnego. Mianowicie wysyła poszczególne stany strumieniem, którego dany element interfejsu aplikacji nasłuchuje. Stan jest to klasa reprezentująca dany stan poszczególnego elementu aplikacji. Relacja w drugą stronę polega na wywoływaniu bezpośrednio funkcji (synchronicznej lub asynchronicznej) na instancji cubita.

3.2 Firebase

Rolę serwera i bazy danych pełni platforma Firebase. Technologia ta, zarządzana również przez Google, określana jest jako Backend-as-a-Service. Firebase dostarcza wiele funkcjonalności wspierające tworzenie projektów w tym między innymi uwierzytelnianie użytkowników, bazę danych NoSQL oraz API odpowiedzialne za komunikację. Usługi, które świadczy Firebase są bezpłatne, jeśli nie przekroczą pewnego z góry ustalonego limitu zdefiniowanego dla poszczególnej funkcjonalności.

3.2.1 Firestore

Firestore jest to baza danych NoSQL w chmurze. Jej strukturą są kolekcje, które zawierają dokumenty, które te z kolei zawierają dane, ale także mogą zawierać kolejne kolekcje.
Definiując zapytania do bazy danych zwracane są jedynie dokumenty z poszczególnej kolekcji lub grup kolekcji. Firestore umożliwia tworzenie prostych zapytań SQL, które filtruje wyniki na podstawie wartości danych znajdujących się w dokumencie.
Firebase umożliwia po stronie klienta implementacje słuchaczy, którzy będą niezwłocznie informowani, gdy w obserwowanym dokumencie zaszła zmiana.
Firestore umożliwia konfigurację dostępu do bazy danych za pomocą definiowania reguł bezpieczeństwa. Dla każdego dokumentu oraz pola można zdefiniować zakres dostępu (czytanie, modyfikacja, usuwanie) dla poszczególnego użytkownika.

3.2.2 Funkcje Firebase

Umożliwiają zaimplementowanie własnego kodu po stronie serwera. Są wywoływane poprzez zapytania HTTPS lub w odpowiedzi na zdarzenia powstałe w bazie danych Firestore.
Możliwe jest wywoływanie bezpośrednio z poziomu aplikacji. Do zapytania dołączane są tokeny uwierzytelniania Firebase i tokeny sprawdzania aplikacji.

3.2.3 Konsola Firebase

Jest to strona internetowa, która oferuje dostęp do wszystkich funkcjonalności Firebase. Aby korzystać z platformy Firebase należy założyć tam własny projekt i skonfigurować go z docelową aplikacją. Po wstępnej konfiguracji otrzymuje się panel kontroli nad platformą Firebase dla projektu. Widzi się cały ruch generowany przez klientów w poszczególnych funkcjonalnościach np. liczba zapisów do bazy danych. Za pomocą konsoli można konfigurować i modyfikować każdą usługę. Zapewnia bezpośrednio dostęp do bazy danych, zarządzanie zarejestrowanymi użytkownikami, monitorowanie procesu działania serwisu.

3.2.4 Emulator

Korzystając z funkcjonalności funkcji Firebase napotyka się na problem długiego czasu oczekiwania na zaktualizowanie kodu źródłowego programisty w chmurze Firebase (czas trwania około minuty). Oczekiwanie dłuższe niż parę sekund sprawia, że czas tworzenia kodu źródłowego po stronie serwera znacząco się wydłuża. Kolejną sprawą jest fakt, że w trakcie, gdy dany projekt jest już dostępny dla użytkowników, nie jest wskazana jego modyfikacja. Zatem trzeba tworzyć kopie projektu, które przeznaczone są modyfikacji,

co w przypadku, gdy nad projektem pracuje wielu programistów, staje się jeszcze bardziej nieefektywne. Ostatnim aspektem są koszty. W momencie programowania i testowania funkcjonalności serwera może dochodzić do generowania dużej ilości zapytań, co z kolei sumuje się do liczby zapytań i ewentualnych płatności.

Z powodu wyżej wymienionych problemów wraz z rozwojem platformy Firebase w 2019 roku oficjalnie został zaprezentowany Firebase Local Emulators. Emulator ten pozwala uruchomić wszystkie usługi Firebase lokalnie na własnym komputerze. Dzięki czemu szybko można nanieść zmiany w funkcjach Firebase, nie modyfikując właściwego projektu, a wykonywane zapytania nie są zliczane.

Każda usługa udostępniana przez platformę Firebase ma swój własny emulator, który jest uruchamiany na osobnym porcie. Zatem można uruchomić jedynie te emulatory tych funkcjonalności, które są wykorzystywane w projekcie. Ponadto fakt uruchomienia lokalnie funkcji Firebase umożliwia debugowanie kodu.

Rozdział 4

Implementacja serwisu

4.1 Serwer

Przy implementacji serwera wykorzystano głównie trzy moduły Firebase takie jak uwierzytelnianie, firestore (baza danych NoSQL) oraz funkcje. Jako metodę rejestracji użytkownika wybrano opcję weryfikacji za pomocą adresu email. Komunikacja pomiędzy serwerem, a klientem opiera się na dwóch sposobach. Klient może bezpośrednio nasłuchiwać zmian zachodzących w bazie danych Firestore, albo wywoływać funkcje Firebase.

4.1.1 Baza danych

W bazie danych czasu rzeczywistego Firestore zostały umieszczone dwie kolekcje - *Users*, *Games*.

W kolekcji *Users* są przechowywane dokumenty indeksowane unikatowymi kodami ID użytkownika, które są przydzielane w czasie rejestracji. Każdy dokument zawiera pola *name* i *active*. Pole *name* odnosi się do nazwy użytkownika, a pole *active* jest wartością logiczną wskazującą, czy dany użytkownik jest dostępny w grze (jest zalogowany i obecnie nie znajduje się w rozgrywce z innymi graczami). Opcjonalne pole *invitation* jest mapą zawierającą klucze *gameId* oraz *player*, których wartości wskazują kolejno na kod ID gry i nazwę gracza, który zaprosił danego gracza do swojej gry.

W kolekcji *Games* są przechowywane dokumenty indeksowane automatycznie generowanymi unikatowymi kodami ID przy tworzeniu dokumentu i które są przypisane jako kody ID gier. Każdy dokument zawiera trzy pola: *available* (wskazuje ile graczy może jeszcze dołączyć do gry), *currentTurn* (wskazuje jakiego gracza jest teraz kolej), *size* (wskazuje na liczbę graczy w grze). Ponadto dokument danej gry zawiera jeszcze subkolekcje *playersQueue*, *playersRacks*, *pool*, *state*.
Subkolekcja *playersQueue* jest to zbiór dokumentów indeksowanych kodami ID graczy, którzy uczestniczą w tej grze. Każdy dokument zawiera pole *name* z nazwą gracza oraz pole *initialMeld*, które wskazuje czy dany gracz wyłożył już rozdanie początkowe.
Subkolekcja *playersRack* to zbiór dokumentów z automatycznie generowanymi kodami ID. Dokumenty te przedstawiają kości, które są przydzielone graczowi. Każdy dokument zawiera dwa pola *color* oraz *number*.
Subkolekcja *pool* to z kolei zbiór dokumentów z automatycznie generowanymi kodami ID. Zbiór tych dokumentów przedstawia bank w grze, czyli wszystkie kości, które nie znajdują się na planszy ani nie są przydzielone do gracza. Każdy dokument ma pola *color* oraz *number*.
Subkolekcja *state* zawiera dokładnie jeden dokument *sets* o najbardziej złożonej strukturze. W dokumencie *sets* znajdują się wszystkie zbiory, które są wyłożone na planszy. Struktura tego dokumentu składa się z mapy, w której klucze to pozycja pierwszej kości z danego zbioru na planszy, a więc moment, w którym rozpoczyna się dany zbiór kości. Wartości tej mapy to tablica kości, gdzie każda kość jest w postaci mapy z kluczami *color* i *number*. Taki sposób przedstawienia stanu planszy wynika głównie z optymizacji kosztów modyfikowania bazy danych. Nie rozbito każdego zbioru kości na osobne dokumenty, ponieważ zwiększa to nam liczbę zliczanych zapisów do bazy danych. Ponadto w tym przypadku nie ma potrzeby

korzystania z właściwości oferowanych przez dokumenty takie jak śledzenie zmian w danym dokumencie czy możliwość tworzenia prostych zapytań SQL na zbiorze dokumentów.

4.1.2 Uwierzytelnianie

Tak jak wspomniano wyżej występują dwa sposoby komunikacji klienta z serwerem. W przypadku funkcji Firebase do zapytań HTTPS z aplikacji, automatycznie są dołączane tokeny uwierzytelniania. W przypadku nasłuchiwaniu zmian zachodzących w bazie danych lub operacjach zaczytywania i modyfikowania bazy danych Firestore za proces autoryzacji dostępu są odpowiedzialne reguły bezpieczeństwa Firestore.
W tym projekcie reguły bezpieczeństwa Firestore są zdefiniowane następująco:

- w kolekcji *users* użytkownik ma dostęp jedynie do dokumentu z własnym ID, na którym może wykonywać operacje czytania i modyfikacji,
- w kolekcji *games* użytkownik ma dostęp jedynie do dokumentu gry, w którym on sam jest jednym z graczy. W tym dokumencie będzie miał dostęp do pól zdefiniowanych w dokumencie oraz do subkolekcji *playersQueue* oraz *playersRacks*. Co więcej w *playersRacks* będzie miał dostęp jedynie do dokumentu z własnym polem ID. We wszystkich dostępnych miejscach użytkownik ma prawo jedynie wykonywać operacje czytania.

4.1.3 Funkcje

Za pomocą funkcji Firebase został zaimplementowany kod serwera serwisu. Jego głównymi zadaniami jest zarządzanie instancjami gier oraz uniemożliwienie prób oszukiwania w czasie rozgrywki przez graczy za pomocą ingerencji w kod źródłowy aplikacji po stronie klienta.

Zbiór zdefiniowanych funkcji został uporządkowany w trzy podzbiory. W pierwszym podzbiorze znajdują się wszystkie możliwe do wywołania przez użytkownika funkcje serwera. Drugim podzbiorem jest klasa *GameUtils*, zawierająca statyczne funkcje odpowiedzialne za zarządzanie instancjami gier. Trzecim podzbiorem jest klasa *GameLogic*, zawierająca statyczne funkcje, które odpowiadają za walidację ruchów użytkowników w grze i implementację logiki rozgrywki gry.

Charakterystyka funkcji serwera możliwych do wywołania przez użytkownika:

- *createGame* - jako paramtry wejściowe przyjmuje ID gracza i jego nazwę, listę nazw graczy zaproszonych do gry oraz czas przeznaczony na wykonanie ruchu w trakcie gry.
  Wywołuje ona metodę *createGame* z klasy *GameUtils*. Następnie wyszukuje poszczególnych graczy i informuje ich o zaproszeniu do gry.
  Parametrem zwracanym jest ID nowo utworzonej gry.
- *searchGame* - jako parametry wejściowe przyjmuje ustawienia gry: liczbę graczy i czas przeznaczony na ruch gracza oraz ID gracza i jego nazwę.
  Wywołuje funkcję *findGame* z *GameUtils*. Jeśli gra zostanie znaleziona wywoływana jest funkcja *addToGame*. Następnie jeśli gracz zajął ostatnie wolne miejsce w grze, wywoływana jest funkcja *startGame* z *GameUtils*. Jednak jeśli nie znaleziono gry, wywoływana jest funkcja *createGame* z klasy *GameUtils*. Aby uchronić się przed wyszukiwaniem i modyfikowaniem bazy danych jednocześnie przez kilka wywołań funkcji *searchGame* przez różnych graczy, proces wyszukania gry i zapisu do niej odbywa się poprzez transakcje.
  Parametrem wyjściowym jest ID gry.
- *addToExistingGame* - jako parametry wejściowe przyjmuje ID gry oraz ID gracza i jego nazwę.
  Funkcja szuka instancji gry, a następnie wywołuje *addToGame* z *GameUtils*. Operacje na bazie danych również odbywają się z pomocą transakcji.
- *startGame* - jako parametr wejściowy przyjmuje ID gry.
  Wywołuje ona funkcje *startGame* z *GameUtils*.

- *putTiles* - jako parametry wejściowe przyjmuje ID gracza i gry oraz zbiór zbiorów kości znajdujących się na planszy.
  Znajduję instancję gry i subkolekcje graczy uczestniczących w niej, a następnie wywołuje metodę *checkTurn* z *GameLogic*. Potem nadaje prawo ruchu kolejnemu graczowi w kolejce. Dalej wywołuje metodę *addNewTiles* z *GameLogic*. Jeśli metoda ta zwróci, że gracz jest zwycięzcą, wskazuje zwycięzcę. Jeśli zwróci, że gracz nie wykonał żadnego ruchu lub próbował oszukiwać, to doda do jego zbioru kości kolejną kość z banku. W przypadku, gdy była to ostatnia kość z banku, to wywołuje metodę *pointTheWinner* z *GameLogic*.

- *leftGame* - jako parametry wejściowe przyjmuje ID gracza i gry.
  Usuwa gracza z kolejki graczy w grze. Jeśli akurat ten gracz miał prawo ruchu, daje możliwość ruchu następnemu graczowi.

Charakterystyka funkcji statycznych serwera z klasy *GameUtils*:

- *createGame* - tworzy dokument gry w kolekcji *games*. Dodaje założyciela do subkolekcji *playersQueue* oraz tworzy wszystkie możliwe kości w postaci dokumentów i dodaje je do subkolekcji *pool*.
  Zwraca ID gry.

- *addToGame* - dodaje gracza do istniejącej już gry.
  Zwraca ID gry oraz wartość logiczną dotyczącą występowania wolnych miejsc w grze.

- *findGame* - za pomocą parametrów ustawień gry (liczba graczy, czas na ruch) szuka wolnej instancji gry.
  Zwraca instancję gry lub null.

- *startGame* - przydziela graczom po 14 kości losowo wybranych z subkolekcji *pool*. Nadaje prawo do wykonania ruchu pierwszego graczowi w subkolekcji *playersQueue*.

Charakterystyka funkcji statycznych serwera z klasy *GameLogic*:

- *checkTurn* - sprawdza, czy dany gracz ma prawo wyłożyć kości.

- *addNewTiles* - przeprowadza walidację nowego zbioru kości na planszy. Sprawdza, czy zbiory są poprawnie ułożone, czy występują wszystkie poprzednie kości na planszy, czy nowo położone kości należą do gracza. W przypadku wyłożenia początkowego sprawdza czy gracz wyłożył nowe ułożenia o wartości co najmniej 30 punktów oraz czy poprzednie ułożenie nie zostały zmodyfikowane. Po spełnieniu tych warunków usuwa nowo wyłożone kości z puli gracza i modyfikuje stan planszy. W przypadku wyłożenia początkowego odznacza, że zostało wykonane.
  Zwraca informację, czy kości zostały poprawnie dodane lub informację, że gracz zwyciężył w przypadku usunięcia wszystkich kości z jego puli.

- *getTileFromPool* - wybiera jedną kość z subkolekcji *pool* i przydziela je danemu graczowi.
  Zwraca informację, czy w *pool* znajdują się jeszcze dostępne kości.

- *pointTheWinner* - sumuje wszystkie wartości kości w poszczególnych pulach graczy. Wskazuje zwycięzcę lub zwycięzców w przypadku równości sum.

4.2 Klient

Warstwa klienta w serwisie napisana we Flutterze jest dostępna jako aplikacja na telefon albo jako strona internetowa. Z uwagi na wieloplatformowość Fluttera kod źródłowy obu wersji klienta jest wspólny z odpowiednimi konfiguracjami i modyfikacjami na poszczególne platformy.

4.2.1 Diagramy klas

4.2.2 Diagramy sekwencji

Rozdział 5

Bot

W ostatnim rozdziale omówiono aspekty związane z implementacją bota. Jakie podejścia zostały zrealizowane do zrealizowania tego celu oraz przeanalizowanie złożoności gry Rummikub. W związku z zastosowaniem architektury trójwarstwowej po stronie klienta instancja Bota zastępuję warstwę danych bez konieczności modyfikowania logiki aplikacji.

5.0.1 Struktura

5.0.2 Podstawowy

5.0.3 Zaawansowany

Podsumowanie

W podsumowaniu należy określić stan zakończonych prac projektowych i implementacyjnych. Zaznaczyć, które z zakładanych funkcjonalności systemu udało się zrealizować. Omówić aspekty pielęgnacji systemu w środowisku wdrożeniowym. Wskazać dalsze możliwe kierunki rozwoju systemu, np. dodawanie nowych komponentów realizujących nowe funkcje.

W podsumowaniu należy podkreślić nowatorskie rozwiązania zastosowane w projekcie i implementacji (niebanalne algorytmy, nowe technologie, itp.).

Bibliografia

[1] Firebase technology. Web pages: https://firebase.google.com/.

[2] A. Miola. *Flutter Complete Reference*. 2020.

[3] J. N. van Rijn, F. W. Takes, J. K. Vis. The complexity of rummikub problems. 2016.

Załącznik A

Zawartość płyty CD

W tym rozdziale należy krótko omówić zawartość dołączonej płyty CD.