Pac-Man Arena

Zaawansowane C++

Michał Dunajski

michal.dunajski.stud@pw.edu.pl

01201897

I ETAP – wstępna analiza funkcjonalna

1. Na podstawie listy proponowanych tematów chciałbym zrealizować projekt o nazwie **Doom dla opornych**, gdzie chciałbym zaimplementować grę Pac-Man, ale z możliwością rozgrywki online kilku graczy.
2. **Krótki opis:**
   W grze PacMan Arena mogłoby grać czterech graczy. Rozgrywka byłaby w trybie survival – tj. trwałaby tak długo, aż pozostałyby dostępne życia tylko jednemu graczowi. W takiej sytuacji byłby zwycięzcą, a pozostali grający znajdowaliby się na miejscach w zależności od tego kiedy odpali z rozgrywki. Na planszach gry zostają przypisane miejsca każdemu z graczy i można opuścić grę. Każdy z graczy widziałby pole gry innych. Mógłby pozostawać w trybie widza po utracie swojego ostatniego życia.
3. **Dłuższy opis:**
   Gracz po uruchomieniu aplikacji widzi ekran, na którym napisane jest **„Graj”** i komentarz pod napisem „naciśnij klawisz Enter by zacząć wyszukiwanie rozgrywki”. Po naciśnięciu widzi stopień zapełnienia lobby, tj. ilu z czterech graczy też czeka na rozgrywkę. Na tym ekranie widzi także napis „naciśnij klawisz Esc by opuścić wyszukiwanie rozgrywki”. Po naciśnięciu Esc wraca do początkowego ekranu z napisem Graj oraz przyciskiem do wyjścia z gry. Po znalezieniu 4 graczy pod statusem lobby pokazuje się komunikat, w którym każdy z graczy widzi: znaleziono graczy, naciśnij enter by rozpocząć. W tym momencie każdy z graczy w ciągu 20 sekund musi potwierdzić Enterem rozpoczęcie. Jeżeli, któryś z graczy nie potwierdzi rozgrywki zostaje wyrzucony z lobby i ponownie wraca do etapu wyszukiwania graczy. W momencie, w którym wszyscy gracze potwierdzają rozgrywkę, ładuje się ekran z grą tj. 4 plansze gry PacMan (2 plansze w 2 rzędach). Każdy z graczy widzi wszystkie plansze gier, ale te na którym gracz wykonuje ruchy zostaje wyróżnione/zaznaczone. Rozgrywka wygląda jak standardowej grze Pac-Man, każdy z graczy jest tytułowym Pac-Manem na swojej planszy i próbuje wyczyścić planszę z małych kulek omijając przeszkadzające mu duchy. Może użyć większych kulek znajdujących się na planszy i wtedy może sam pochłaniać duchy, za które również dostaje punkty (jak za kulki). Każdy z graczy ma 2 dodatkowe życia (tj. pierwsze z którym startuje i dwa dodatkowe odrodzenia po kolizji w duchem). Każdy z duchów ma inną prędkość, ruch duchów jest

losowy. Gra kończy się gdy tylko 1 gracz ma dostępne życia. W takiej sytuacji gra się zatrzymuje i pokazana jest końcowa klasyfikacja graczy (wraz z ilością punktów). W trakcie gry każdy z graczy może opuścić grę przez naciśnięcie przycisku „opuść grę". Przycisk ten dostępny jest także przy końcowej klasyfikacji wyników. W menu głównym jest także możliwość pokazania ekranu pomocy, gdzie wytłumaczone są zasady. Należy nacisnąć klawisz „P" by wejść do tego menu. W menu pomoc napisane jest że klawisze strzałek używane są do ruchu Pac-Manem i że należy nacisnąć Esc by wrócić do menu głównego.

4. **Technologie, które zostaną użyte w aplikacji.**
   W skrócie:
   Grafika: SFML
   Sieć: SFML Network (może asio boost).
   Obsługa wejścia: SFML.
   Kompilator: MinGW.
   Narzędzia developerskie: Visual Studio Code (może CLion), Github, CMake.
   Wytłumaczenie decyzji:
   Ze względu na brak doświadczenia z takimi aplikacji/grami dobór technologii został dokonany na podstawie informacji znalezionych w sieci. Chciałbym skorzystać w bibliotek SFML (w wersji 3.0, ze względu na wsparcie C++17), które umożliwią narysowanie ekranów gry, obsługę wejścia (klawiszy graczy) oraz sieci (do przesyłania stanu gry innych graczy). Chciałbym żeby ekrany menu oraz gra była zbudowana z podstawowych kształtów dostarczonych przez bibliotekę SFML (okna tekstowe oraz podstawowe figury geometryczne jak prostokąty do ścian oraz okręgi do duchów i Pac-Mana).
   Alternatywnie brana pod uwagę jest biblioteka Asio do obsługi części sieciowej, ze względu na ogólną chęć poznania jej.
   Repozytorium z kodem zostanie umieszczone na platformie Github.
   Zamierzam skorzystać z kompilatora MinGW, ze względu na używany system operacyjny – Windows.
   Środowisko do rozwijania aplikacji, którego będę używać to Visual Studio Code, choć skłaniam się ku temu, żeby użyć CLion. Chciałbym użyć CLion, bo słyszałem o tym narzędziu pochlebne opinie i z tego co mi wiadomo domyślnie wspiera budowanie aplikacji z wykorzystaniem CMake, którego również zamierzam użyć. Możliwym jest, że środowisko Visual Studio Code zostanie użyte jako te do pisania kodu (ze względu na duże doświadczenie w tym narzędziu – praca na co dzień), a CLion byłby używany do debugowania.
5. **Pierwsze przemyślenia/założenia implementacyjne/techniczne**
   W tej części chciałbym opisać pierwsze przemyślenia na temat szczegółów bardziej technicznych czy implementacyjnych nad którymi się zastanawiam.
   Kto jest serwerem [kontekst sieciowy]?

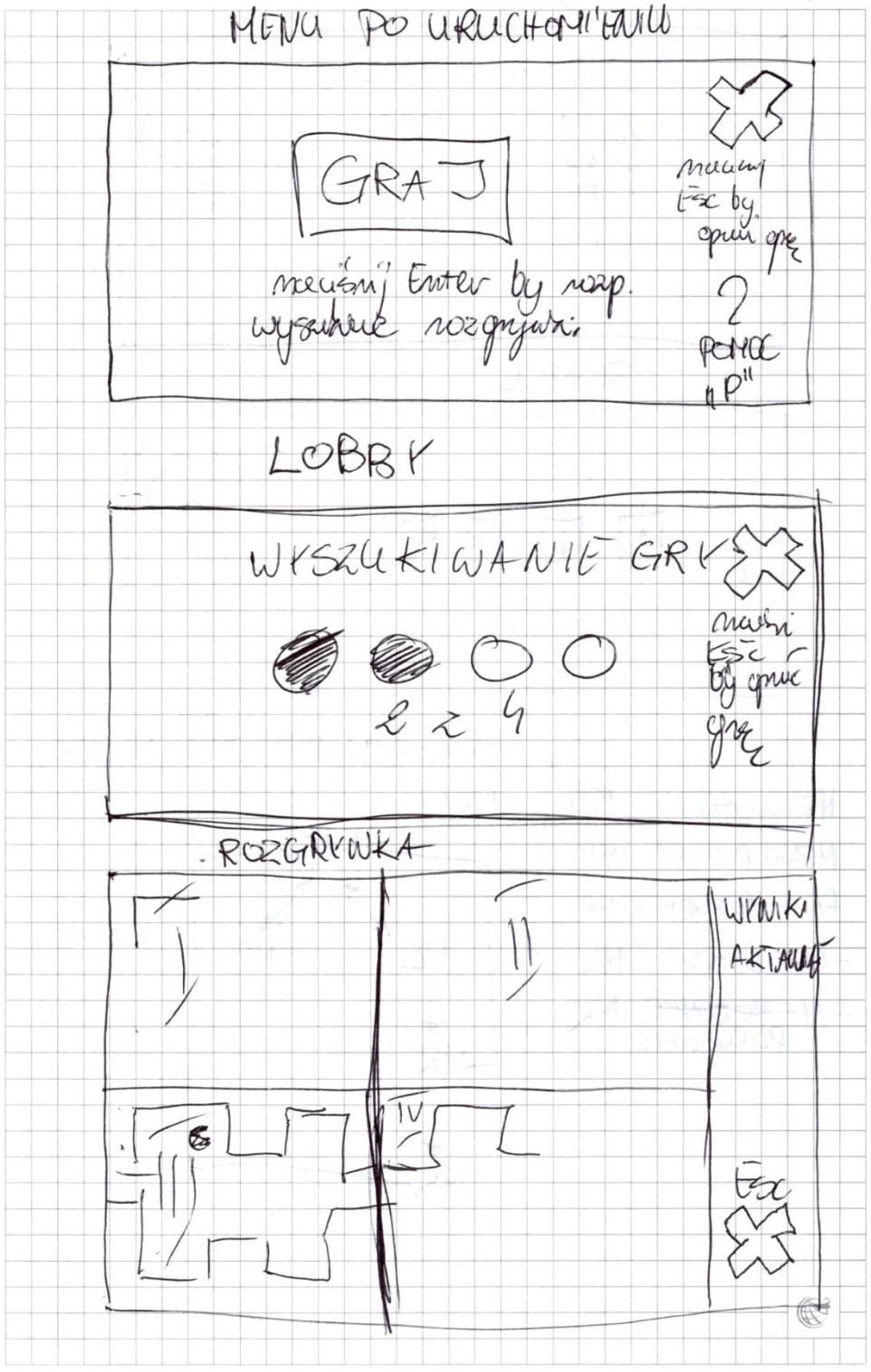
Początkowo chciałem założyć, że jeden z czterech graczy oprócz tego że byłby klientem to dodatkowo świadczył by usługę serwera dla danej rozgrywki, ale wtedy powstałby problem, w którym jeden z graczy opuszcza rozgrywkę w trakcie jej trwania. Należałoby implementować jakiś algorytm, który „ratowałby" rozgrywkę pozostałych graczy (np. poprzez uruchomienie nowego serwera na urządzeniu jednego z pozostałych graczy – dziwne rozwiązanie) lub rozgrywka byłaby nierozstrzygnięta – trochę „słabe" rozwiązanie. Z tego powodu podjęto decyzję, że serwer będzie uruchomiony na stałe, a klient gry może się do niego podłączyć (po uprzednim spełnieniu warunku lobby, które też działa na serwerze). Takie rozwiązanie wydaje mi się prostsze. Serwer nasłuchiwałby na porcie i oczekiwał na klienta gry, który po podłączeniu przesyłałby zmiany w swojej rozgrywce. Serwer odsyłałby także do klienta wszelkie zmiany związane ze zmianą w rozgrywkach pozostałych graczy. Ze względu na brak konieczności wysokiej „płynności" danych (gracze nie grają między sobą bezpośrednio tylko każdy na swojej planszy) zamierzam wybrać protokół TCP, który umożliwi mi dokładne odwzorowanie ruchów innych graczy. Całość zamierzam uruchomić w lokalnie.

Ruch duchów.

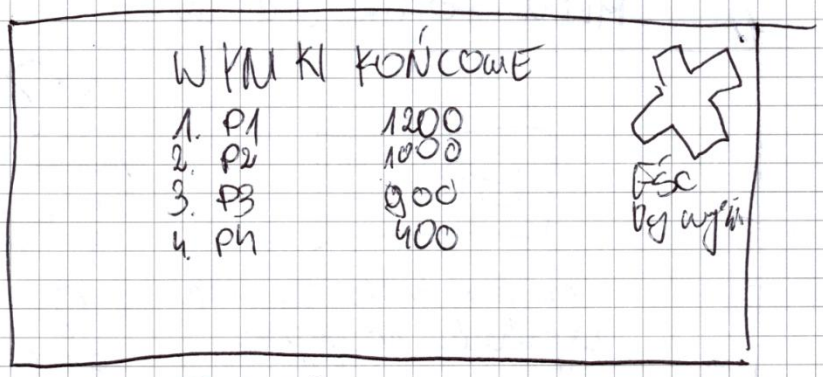
Duchy w standardowej grze Pac-Man poruszają się w różny sposób (wolniej, szybciej, w kierunku Pac-Mana lub w losowym kierunku). W każdym razie chciałbym zacząć od ruchu losowego duchów, a jeżeli pozwoliłby na to czas i zadanie by mnie nie przerosło to spróbować zaimplementować algorytm A*, który jest bardziej zaawansowany.

6. **Odręczne notatki opisujące funkcjonalność.**
   Zamieszczam trzy skany kartek, które zawierają odręcznie narysowane propozycje menu (poglądowe) wraz z diagramem przejść między menu. Sposób łączenia się i przesyłania danych między serwerem i klientami.

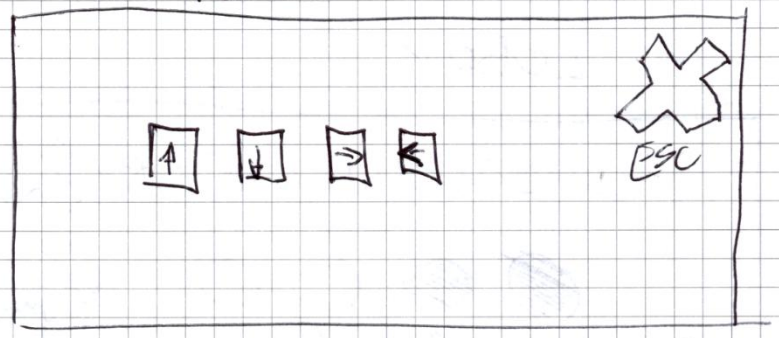
Zaawansowane C++ Projekt Michał Dajska



PO ROZGRYWCE

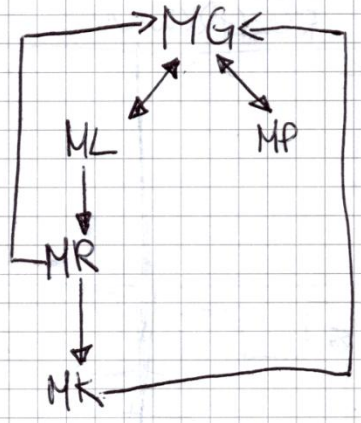


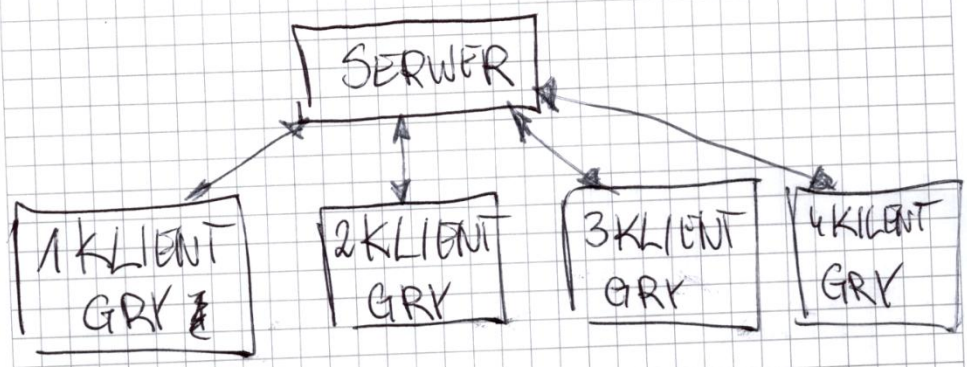
POMOC



MENU GŁÓWNE MG
MENU POMOCY MP
f/1- ROZGRYWKA MR
-11- LOBBY ML
-11- ~~POKOJE~~ PO ROZGRYWCE MK

DIAGRAM PRZEJŚĆ MIĘDZY MENU





Serwer wysyła do wszystkich graczy stan pozostałych gier w danych kliencie.

stan gry gracza 1

K1 $\xrightarrow{SG1}$ S

K2 $\xrightarrow{SG2}$ S

K3 $\xrightarrow{SG3}$ S

K4 $\xrightarrow{SG4}$ S

S $\xrightarrow{SG2, SG3, SG4}$ K1

S $\xrightarrow{SG1, SG3, SG4}$ K2

S $\xrightarrow{SG1, SG2, SG4}$ K3

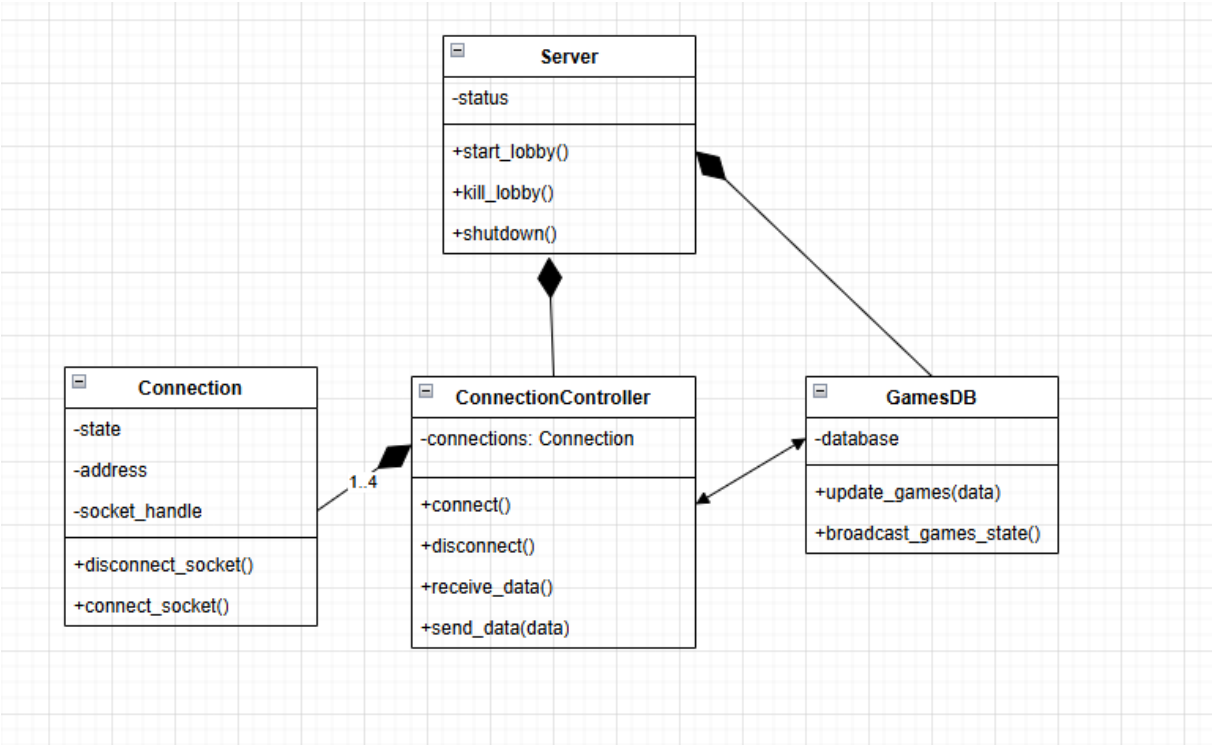
S $\xrightarrow{SG1, SG2, SG3}$ K4

Zastanowić się czy serwer nie lepiej jakby wysyłał do wszystkich to samo, a klient "ignoruje" SG, który go dotyczy. 1 RAMKA DO WSZYSTKICH, mniej zasoby z serializowaniem/deserializowaniem.

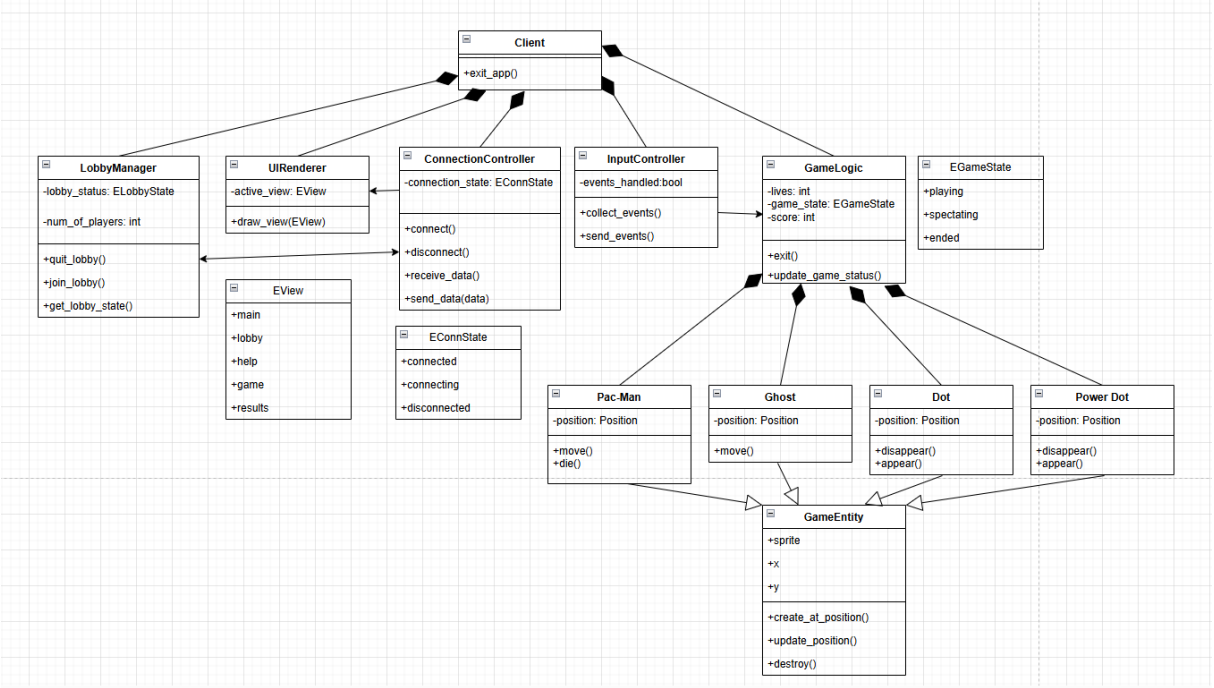
7. **Propozycje możliwych dodatkowych rozwiązań.**
    - Tryb Time-Attack, w którym gracze konkurują między sobą, kto otrzyma największą ilość punktów w określonym czasie, utracenie życia oznacza uzyskanie 0 punktów.
    - Tryb 1v3, w którym trzech graczy przejmuje sterowanie nad duchami, a jeden z nich (losowo wybrany) jest Pac-Manem, reszta rozgrywki pozostaje bez zmian (warunkiem zwycięstwa Pac-Mana jest wyczyszczenie całej planszy, a duchów schwytanie Pac-Mana co najmniej 2 razy).
    - Przechowywanie na serwerze najwyższych wyników. W kliencie należałoby umożliwić ustawienie nicku gracza, a następnie serwer przechowywałby wyniki najlepszych graczy – top 10 najwyższych wyników punktów, ze wszystkich rozgrywek.

II ETAP – diagram klas z ich opisem

1. Aplikacja została podzielona na dwie mniejsze aplikacje:
    a. Aplikacja klienta (Client w diagramach), która służy do:
        i. Obsługi wejścia – zbieranie sygnałów sterujących od użytkownika
        ii. Obsługi połączenia sieciowego – cykliczne wysyłanie stanu gry, odbieranie stanu gry innych grających
        iii. Obsługi rysownia stanu gry wszystkich graczy
    b. Aplikacja serwera gry (Server w diagramach), która służy do zapewnienia połączenia graczy z serwerem w celu pobierania danych o rozgrywkach innych graczy i wysyłania stanu swoich rozgrywek.
2. Diagram Serwera. Na diagramie znajdują się klasy:
    a. **GamesDB**, która odpowiedzialna jest za przechowywanie stanu rozgrywek wszystkich 4 graczy. Klasa ta cyklicznie rozsyła klientom gry stany rozgrywek wszystkich graczy.
    b. **ConnectionController**, która odpowiada za sterowanie połączeniami z klientami. W klasie tej znajduje się tablica połączeń z klientami. Klasa ta umożliwia przesyłanie danych między klientem a serwerem gry.
    c. **Connection,** klasa pomocnicza dla klasy **ConnectionController**, która zawiera w sobie detale dotyczące połączenia (status, uchwyt na socket sieciowy, adress socketa).
    d. **Server**, główna klasa aplikacji, która po uruchomieniu tworzy instancje klas koniecznych do spełnienia funkcjonalności serwera.



3. Diagram Klienta. Na diagramie znajdują się:
    a. **UIRender**, klasa która odpowiedzialna jest za rysowanie poszczególnych widoków menu oraz okna gry. Dla okna gry są to dane otrzymane przez **ConnectionController** z rozgrywek innych graczy oraz dane otrzymane przez **GameLogic** ze stanem gry (lokalnej, rozgrywanej bezpośrednio w kliencie gry).
    b. **LobbyManager**, klasa pełniąca funkcję zarządzanie lobby poszukiwania rozgrywki – na podstawie danych z serwera (klasa korzysta z **ConnectionController** do wymiany informacji) przechowuje ilość graczy znajdujących się w lobby.
    c. **InputController**, klasa zajmuje się obsługą wejścia od użytkownika, przesyła do klasy **Game**, naciśnięcia klawiszy gracza.
    d. **GameLogic**, klasa obsługująca logikę gry. Klasa ta reaguje na zdarzenia na wejściu wysłane przez **InputController**.
    e. **GameEntity**, klasa bazowa dla obiektów w grze, takich jak Pac-Man, duchy i kropki. Przechowuje pozycję oraz umożliwia ich tworzenie, aktualizację i usuwanie. Klasy **Pac-Man**, **Ghost**, **Dot**, **Power Dot** są odpowiednikami klas pochodzących z gry.
    f. **ConnectionController**, która odpowiada za sterowanie połączeniem z serwerem.
    g. **Client**, główna klasa aplikacji, która po uruchomieniu tworzy instancje klas koniecznych do spełnienia funkcjonalności klienta gry.



4. **Komentarz ogólny**

   Starałem się opisać diagramy i klasy jak najlepiej, jednak moje doświadczenie z programowaniem obiektowym i modelowaniem UML jest jeszcze ograniczone.

   Dlatego będę wdzięczny za wszelkie uwagi lub sugestie dotyczące poprawności oraz czytelności diagramów, czy też samej struktury aplikacji.

   Diagramy nie są ostateczne – będą modyfikowane i rozwijane w trakcie implementacji oraz rozbudowy aplikacji. W miarę postępu prac mogą pojawić się nowe zależności, klasy lub zmiany w istniejących strukturach.

   Chciałbym zaimplementować MVC w projekcie, choć może to jasno nie wynika z diagramu. Gdzie Modelem byłaby klasa obsługująca logikę gry, View klasa rysująca/renderująca obszar gry i menu, a Controllerem klasy od obsługi wejścia od użytkownika oraz klasa odpowiedzialna za połączenie sieciowe klienta.