Universal Game System Documentation

Maciej Bartoszuk Łukasz Dudziński
Artur Goździkowski Kamil Korolkiewicz Piotr Siewko

6 lutego 2011

Streszczenie

Dokument opisuje projekt systemu Universal Game System, z wyszczególnieniem architektury poszczególnych komponentów. Prezentuje oprogramowanie, które będzie używane podczas pracy, wyszczególnia język programowania, paradygmaty programistyczne, konwencje nazewnicze jak i metodologię tworzenia oprogramowania, które zostaną użyte przy implementacji systemu. Dokument ma na celu dokładne zaprezentowanie kształtu całego systemu w celu późniejszego zaprogramowania go, z uwzględnieniem poszczególnych komponentów oraz zgodnie z wymaganiami przedstawionymi we wstępnej analizie.

Spis treści

Spis diagramów

1 Opis ogólny

1.1 Wprowadzenie

1.1.1 Cel

Celem projektu jest stworzenie systemu uniwersalnego serwera gier, w skład którego wchodzą: aplikacja gracza, aplikacja game mastera oraz serwer gier. Dokument zawiera informację dla programistów, umożliwiające stworzenie systemu, który jednocześnie spełnia następujące kryteria:

- Dowolna gra posiadająca zasady gry może być obsługiwana przez UGS.
- Serwer ma możliwość rozgrywania na nim pojedynczych partii jak i turniejów.
- Game Master kontroluje legalność ruchów graczy i odpowiedni na nie reaguje.
- Gracz jest aplikacją sterowaną przez sztuczną inteligencję.
- Wszystkie programy mają wymagać jak najmniejszej ingerencji z zewnątrz w trakcie swojego funkcjonowania.

1.1.2 Zakres

Swoim zakresem projekt obejmuje stworzenie UGS, a dokładnie trzech podsystemów z których się składa: serwera gier, game mastera oraz graczy. Stworzeniu sztucznej inteligencji sterującej graczami oraz mechanizmów umożliwiających prowadzenie rozgrywki między nimi zarówno w trybie pojedynczej gry, jak i w trybie turnieju. Zdefiniowaniu mechanizmów umożliwiających poprawne działanie serwera, odpowiedzialnego za komunikację gracze - GM oraz samej struktury aplikacji game masterów, którzy będą kontrolować poprawność rozgrywek.

1.1.3 Słownik

Universal Game System (Uniwersalny System Gier) system w skład którego wchodzą: *serwer*, *game masterzy* oraz *gracze*. Umożliwia rozgrywki między *graczami* na zasadach ustalonych przez *game masterów*, organizację *turniejów* etc. Jest to system, którego implementację opisuje ten dokument.

UGS akronim od nazwy *Universal Game System*, wykorzystywany w dalszej części dokumentu.

Gra/Rozgrywka czynność wykonywana wspólnie przez wielu *graczy*. Każda gra posiada te same *zasady* dla wszystkich uczestników. Gra składa się z ruchów wykonywanych kolejno przez wszystkich graczy w określonej kolejności. *Zasady* są niezależne od serwera. Gra rozpoczyna się gdy zbierze się wystarczająca liczba *graczy*. Sposób zbierania uczestników może być różny dla różnych gier i jego kształt leży w gestii *game mastera*. *Game master* decyduje również o tym kiedy gra się zakończy oraz kto jest zwycięzcą.

Zakończenie gry stan *gry*, powodujący zakończenie pojedynczej rozgrywki. Wszyscy gracze są informowani o *rezultacie* gry.

Rozpoczęcie gry stan *gry*, w którym zbierani są *gracze*. Ten etap jest rozpoczynany przez *serwer* i kontrolowany przez *game mastera*.

Game Master (Mistrz Gry) aplikacja, która jest łączy się z *serwerem*. Kontroluje rejestrację *graczy* do *gry* (między-innymi minimalną liczbę *graczy* potrzebnych do rozpoczęcia rozgrywki), kolejność ruchów oraz ich poprawność (legalność). *Game Master* definiuje również *grę*, która będzie rozgrywana przez zebranych *graczy*. Każdy ruch dowolnego z *graczy* jest wysyłany do *serwera*, który następnie przekazuje go do *Game Mastera*. Jeśli ruch jest poprawny, *Game Master* wysyła potwierdzenie (zawierające nowy *stan gry*) do *serwera*, który propaguje tę informację do wszystkich *graczy*. Każdy *Game Master* może kontrolować wiele *gier* (jednego typu) w tym samym czasie.

GM akronim od nazwy *Game Master*, wykorzystywany w dalszej części dokumentu.

Stan gry informacja o *stanie gry*, która jest rozsyłana między *graczami*, *Game Masterem* oraz *serwerem*. *Serwer* przesyła tę informację do *Game Mastera*, który jest odpowiedzialny za wszystkie decyzje dotyczące *zasad*. *Stan gry* jest wysyłany do wszystkich *graczy* po każdym ruchu. *Serwer* nie interpretuje tych informacji, zważywszy, że są one różne dla różnych typów *gier*.

Typ gry Informacja o *typie gry*, którą obsługuje dany *Game Master* lub w którą potrafi grać pewien *gracz*, każdy *gracz* zna zasady tylko jednej *gry*, do której został stworzony. Przykładowymi *typami gier* są: warcaby, szachy, kółko i krzyżyk, statki itd.

Przegrany *gracz*, który przegrywa. Nie ma ograniczeń co do liczby *przegranych* w pojedynczej *grze*. Po zakończeniu *gry*, każdy z *graczy* musi być *zwycięzcą* lub *przegranym*.

Ruch pojedynczy *ruch* wykonywany przez pojedynczego *gracza* w *stanie gry*. *Ruch* może zostać zaakceptowany przez *Game Mastera* jeśli jest zgodny z *zasadami* lub odrzucony, jeśli łamie *zasady* gry.

Maksymalny czas oczekiwania okres czasu, po którym uznaje się, że *gracz* utracił łączność z *serwerem*, bądź sam się rozłączył. Zastosowanie tego mechanizmu zabezpiecza system przed zablokowaniem w sytuacji utraty łączności z którymkolwiek z *graczy*.

Gracz / Uczestnik gry pojedyncza aplikacja zdolna ustanowić połączenie z *serwerem*, ustalić aktualny *typ gry*, odebrać informację o *stanie gry* oraz wykonać *ruch* używając sztucznej inteligencji, która została w niej zaimplementowana.

Lista graczy zbiór *graczy* czekających na zaproszenie do *gry*. Każdy *gracz*, który chce dołączyć do *gry* musi najpierw się zarejestrować do danego *typu gry*. *Serwer* wykorzystuje *listę* przy wyborze *graczy* do *gier*.

Wynik dwustanowa informacja o wszystkich *graczach*, wysyłana do wszystkich *uczestników* po zakończeniu *gry*: *wygrany*, *przegrany*. Na przykład: gracz1: wygrany, gracz2: przegrany, gracz3: wygrany, gracz4: przegrany.

Zasady zbiór *zasad* danej *gry*. *Zasady* są zdefiniowane wewnętrznie w aplikacji *gracz* oraz w aplikacji *Game Mastera*. *Serwer* nie podejmuje żadnych decyzji związanych z *zasadami*. Poprawność ruchów jest sprawdzana przez *Game Mastera*.

Serwer organizuje *gry* między *graczami*. Ogólnie przesyła informacje pomiędzy *Game Masterem* i *graczami*. Nie podejmuje żadnych decyzji związanych z *zasadami*. Wszystkie leżą w gestii *Game Mastera*. *Serwer* może organizować nieograniczoną liczbę *gier* w jednym czasie. *Serwer* wybiera *graczy* z *listy graczy*. *Gracze* nigdy nie komunikują się bezpośrednio, tylko poprzez *serwer*.

Zwycięzca *Gracz*, który wygrywa. Nie ma ograniczeń co do liczby *zwycięzców* w pojedynczej *grze*. Po zakończeniu *gry*, każdy z *graczy* musi być *zwycięzcą* lub *przegranym*.

Turniej specjalny tryb *gry*, w którym *gracze* rozgrywają partie każdy z każdym, *gracz* z największą liczbą wygranych *gier* wygrywa *turniej*. *Turniej* może być organizowany tylko w *typach gier*, które są przeznaczone dla dwóch *graczy*. *Turniej* jest organizowany przez *serwer* i on też przechowuje informacje o wszystkich *wynikach* oraz wybiera *graczy* do *gry*. Jeśli w czasie rozgrywki jeden z zawodników przestanie odpowiadać drugi zostaje *zwycięzcą*. Po zakończeniu *turnieju serwer* prezentuje *wyniki* wszystkich *gier* i wyłania *zwycięzcę* zawodów. Każdy z *graczy* otrzymuje informację o *wynikach* po zakończeniu *turnieju*.

1.1.4 Referencje

- Analiza wymagań na podstawie której jest tworzony ten dokument *Universal Game System Initial Analysis, 19.11.2010 [Krzysztof Kaczmarski]*

1.1.5 Krótkie omówienie

Dokument w dalszej części zawiera informację o poszczególnych podsystemach UGS, które mają zostać zaimplementowane, konwencjach nazewniczych oraz oprogramowaniu wspomagających tworzenie zarówno dokumentacji, jak i ostatecznego produkt (systemu). Dokładnie określa zakres prac programistów, wymogi jakie są przed nimi stawiane oraz metodologie, które będą stosowane w celu organizacji zarówno kodów źródłowych programów, czy też pracy zespołu. Diagramy UML oraz wyjaśnienie każdej z części system ma na celu w znaczący sposób uprościć pracę w kolejnym etapie (implementacji).

1.2 Aspekty techniczne

Niniejszy rozdział jest poświęcony na uściśleniu pewnych aspektów projektu, które potrzebne będą przy jego późniejszej implementacji. Można tu znaleźć informacje na temat przyjętego stylu dokumentacji projektu, przyjęte konwencje programistyczne oraz projekt protokołu komunikacji między elementami systemu. Zawarte informacje należy jednak traktować czysto poglądowo, zastrzegamy sobie prawo do zmiany niniejszych ustaleń (np. w wypadku pojawienia się dodatkowych wymagań podczas pracy)

1.2.1 Standardy dokumentacyjne

W celu sporządzenia tej pracy wykorzystano oprogramowanie do składania tekstu LaTeX, które pozwoliło utrzymać w sposób jednolity rozkład elementów na stronie oraz automatyczne generowanie spisu treści. Podstawę prezentacji opisywanych obiektów i zachodzących między nimi procesów stanowią diagramy zgodne ze standardem UML (Unified Modeling Language). W szczególności są to wykresy przypadków użycia, stanów, klas oraz sekwencji. Na końcu dokumentu zawarto spis wszystkich diagramów, co ma na celu ułatwienie ich wyszukiwania. Ponadto, poza opisem normalnego działania systemu, szczególna uwaga została zwrócona na rozważenie wyjątkowych sytuacji mogących zakłócić poprawne zachowanie się któregoś ze składników systemu.

1.2.2 Standardy programistyczne

Projekt zostanie zaimplementowany w wersji 1.6 języka Java. Zdecydowaliśmy się na ten właśnie język ze względu na niezależność od platformy systemowej oraz fakt, że w standardowych bibliotekach zawarto podstawowe klasy potrzebne do obsługi wielowątkowości oraz komunikacji internetowej. Kod źródłowy aplikacji będzie zgodny z szeroko pojętą konwencją pisania programów w języku Java. Mimo tego, by maksymalnie ujednolicić efekt pracy w zespole, przyjęliśmy następujący schemat pisania oraz formatowania kodu:

1. Nazewnictwo
    - Nazwy wszystkich stałych, zmiennych, pól, metod oraz klas mają być w języku angielskim.
    - Deklaracje stałych pisane wielkimi literami, poszczególne słowa opisujące stałą oddzielone znakiem podkreślenia "_" (np: *static final typ* NAZWA_STAŁEJ)

- Deklaracje zmiennych i pól klas pisane bez spacji, małymi literami oprócz liter rozpoczynających kolejne wyrazy (np: *typ zmienna, typ nazwaZmiennej*)
- Deklaracje metod – tak jak w powyższym podpunkcie, zarówno jeśli chodzi o ich nazwy, jak i nazwy ich parametrów (np *typZwracany nazwaMetody(typ1 nazwaParametru1, typ2 nazwaParametru2, ...)*)
- Deklaracje klas i interfejsów pisane małymi literami, oprócz liter rozpoczynających każdy wyraz (np: *class KlasaBazowa*)
- Oczywiście docelowo wszystkie nazwy mają być jak najkrótsze oraz w miarę możliwości jednoznacznie mówić o ich przeznaczeniu.

2. Kolejność rozmieszczenia elementów w klasie:

   - stałe (publiczne, chronione, prywatne)
   - prywatne pola
   - konstruktory (w kolejności od publicznych do prywatnych, ewentualnie według liczby parametrów)
   - metody ustawione według funkcjonalności
   - metody pozwalające na odczytanie i przypisanie wartości polom w klasie (zarówno publiczne jak i prywatne i chronione)

3. Format kodu:

   - Cały program podzielony zostanie ze względu na zastosowanie poszczególnych klas, czyli w sposób naturalny, na pakiety.
   - Wszystkie klasy oraz interfejsy będą umieszczone w oddzielnych plikach
   - Nawiasy klamrowe rozpoczynające definicję klas i metod mają znajdować się w tej samej linii co nazwa klasy, czy nagłówek metody

Poniżej zamieszczona została definicja przykładowej klasy Human, w celu demonstracji powyższych reguł.

```
class Human{
        public static final int MALE = 0;
        public static final int FEMALE = 1;
        private static final String DEFAULT_NAME="Mały Jasio";

        private int gender;
        private String firstName;

        public Human(int gender){
                        this(gender,  DEFAULT_NAME);
        }
        public Human(int gender, string firstName){
                        setGender(gender);
                        setFirstName(firstName);
        }

        public String toString(){
                        string result =
                                getGender() == MALE ? "M" : "K";
                        result += getFirstNamer() ;
                        return  result;
        }

        public int getGender(){
                        return gender;
        }
        private void setGender(int gender){
                        this.gender = gender;
        }
        public String getFirstName(){
                        return firstName;
        }
        private void setFirstName(String firstName){
                        this.firstName  = firstName;
        }
}
```

1.2.3 Narzędzia

Podstawowym narzędziem wykorzystanym do stworzenia aplikacji będzie darmowe, wieloplatformowe środowisko Eclipse. Być może najważniejszym

argumentem stojącym za tym wyborem był, że wszyscy członkowie naszego zespołu mają doświadczenie w używaniu tego środowiska. Eclipse jest wyposażony w bardzo wiele pomocnych opcji, takich jak inteligentne podpowiedzi, automatyczne generowanie metod do odczytu i zapisu pól w klasach czy współpraca z programem Javadoc służącym do tworzenia eleganckiej dokumentacji kodu. Ostatnia cecha tego środowiska będzie bardzo przydatna w momencie, gdy jednym z zadań narzuconych przez prowadzących będzie stworzenie dokumentacji kodu źródłowego. Ponadto Eclipse ma możliwość rozszerzania swojej funkcjonalności za pomocą wtyczek. Mamy zamiar użyć darmowego dodatku Subeclipse współpracującego z systemem kontroli wersji Subversion, który jest ogólnie polecanym i przyjaznym narzędziem. Repozytorium projektu będzie przechowywane na uczelnianym serwerze studenckim.

1.2.4 Protokół komunikacyjny

Aplikacja wymagała zaprojektowania własnego protokołu w standardzie XML. W załączniku do tego dokumentu zawarty jest projekt w formacie xsd, zgodnym z zaleceniami organizacji w3.org (*http://www.w3.org/2001/XMLSchema*).

Protokół umożliwia przesłanie:

- informacji o błędzie, np:

```
<error>
        <id>
                123
        </id>
        <message>
                Wrong  move
        </message>
</error>
```

- żądania zarejestrowania Game Mastera np:

```
<gameMasterRegister>
        <gameName>
                Poker
        </gameName>
        <playerCount>
                6
        </playerCount>
</gameMasterRegister>
```

- żądania przydzielenia gracza do rozgrywki w odpowiednim trybie np:

```
<requestGame>
        <nick>
                player1
        </nick>
        <gameName>
                Poker
        </gameName>
        <championshipGameMode>
                false
        </championshipGameMode>
</requestGame>
```

- rozpoczęcia rozgrywki wraz z nadaniem grze unikalnego numeru np:

```
<beginGame>
        <gameId>
                2
        </gameId>
        <playerNicks>
                <nick>
                        player1
                </nick>
                <nick>
                        player2
                </nick>
        </playerNicks>
</beginGame>
```

- stanu gry wraz z ewentualną listą zwycięzców lub z zrzutem aktualnego wyglądu planszy; przykład dla stanu końcowego

```
<gameSate>
        <gameId>
                1234
        </gameId>
        <stateName>
                Finished
        </stateName>
        <stateContext>
                <gameResult>
```

```
                                <nick>player1</nick>
                                <result>1</result>
                                <nick>player2</nick>
                                <result>3</result>
                        </gameResult>
                </stateContext>
        </gameSate>
```

- proponowanego ruchu gracza np:

```
<requestMove>
        <gameId>
                1234
        </gameId>
        <move>
                <!-- Move coded in base64-->
        </move>
</requestMove>
```

- zatwierdzonego ruchu gracza przez serwer np:

```
<propagateMove>
        <gameId>
                1234
        </gameId>
        <move>
                <!-- Move coded in bas64-->
        </move>
</propagateMove>
```

Wszystkie powyższe przykłady muszą być otoczone tagami

```
<UGSMessage xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation=
"UGSProtocolXMLSchema.xsd">
```

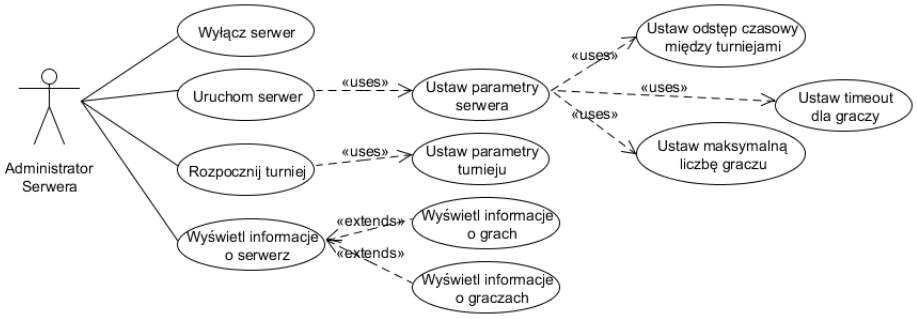
oraz

```
</UGSMessage>
```

Należy dodatkowo zwrócić uwagę na fakt, że powyższy protokół jest nie zależny od typu gry. Zawdzięcza się to użyciu base64 kodowania zrzutu planszy oraz ruchów.

2 Specyfikacja komponentów

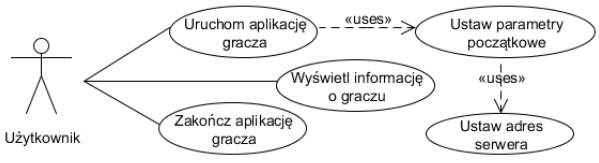
W tej części dokumentu zostaną opisane wszystkie komponenty jakie wystąpią w opisywanym systemie informatycznym. Jednak zanim przejdziemy do opisu konkretnych komponentów, należy powiedzieć jak system będzie wyglądał jako całość. Najpierw użytkownik będzie uruchamiał aplikację Serwera. Jego możliwości konfiguracji i obsługi tej częsci systemu obrazuje poniższy diagram użycia *Diagram przypadków użycia Serwera przez osobę uruchamiającą Serwer.*



Rysunek 1: Diagram przypadków użycia Serwera przez osobę uruchamiającą Serwer

Następnie użytkownik (niekoniecznie ten sam co poprzednio) uruchamia aplikację Game Mastera, podając adres serwera do którego ten musi się podłączyć.

Ostatecznie użytkownik lub użytkownicy (nie muszą być to ci sami co użytkownicy uruchamiający Serwer i Game Mastera) uruchamiają aplikacje Graczy, podając także adres Serwera. Ci Gracze rozgrywają tam pojedyncze gry lub całe turnieje. Dokładniej opisuje to diagram przpadków użycia *Diagram przypadków użycia Gracza przez osobę uruchamiającą Gracza.*

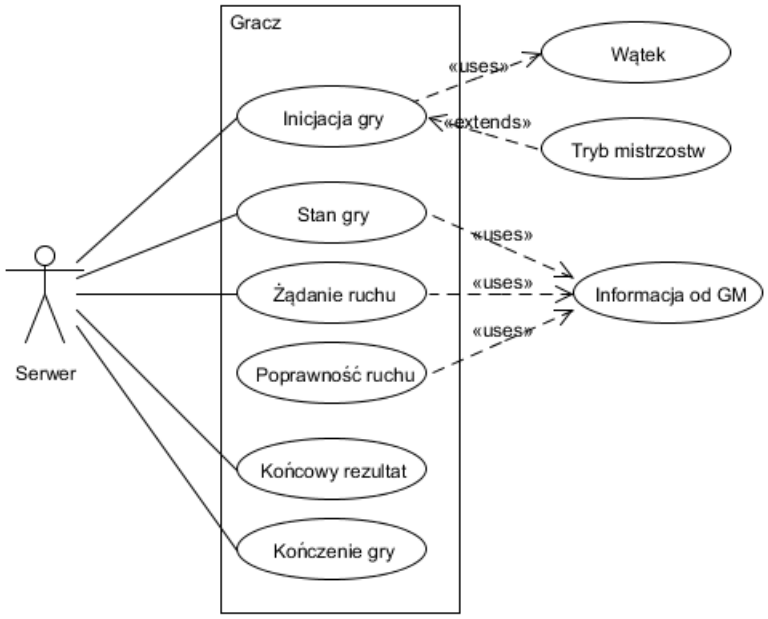


Rysunek 2: Diagram przypadków użycia Gracza przez osobę uruchamiającą Gracza

2.1 Serwer

2.1.1 Wstęp

Serwer jest aplikacją służącą do komunikacji między GM a Graczami uczestniczącymi w Grze. Odpowiada za organizację Gier i Turniejów. Uruchomienie Serwera jest konieczne do poprawnego działania aplikacji GM i Graczy. Zastosowanie Serwera obrazuje poniższy diagram *Diagram przypadków użycia komunikacji Serwer-Gracz.*



Rysunek 3: Diagram przypadków użycia komunikacji Serwer-Gracz

2.1.2 Opis

Zasadniczą funkcją Serwera jest pośredniczenie w przekazywaniu danych od graczy do GM i w drugą stronę. Komunikacja odbywa się poprzez protokół XML. Dowolność liczby gier rozgrywanych wymusza wielowątkowość aplikacji serwerowej.

Dopóki do serwera nie podłączy się GM i wystarczająca liczba Graczy, nie może zostać rozegrana Gra. W przeciwnym wypadku serwer może podjąć decyzję o rozpoczęciu nowej Gry lub Turnieju (na które składa się wiele Gier należących do jednej puli). Warunkiem koniecznym jest jednak, aby liczba Graczy nie uczestniczących w danym momencie w rozgrywce (i będących w

Pili graczy) była większa równa minimalnej liczbie graczy potrzebnych do Gry o minimalnej liczbie uczestników. Turniej może być rozgrywany wyłącznie, jeśli istnieje Gra dla dokładnie dwóch Graczy (ten tryb zostanie opisany później).

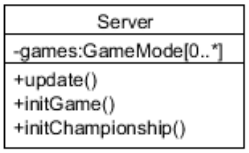
Decyzja o Grze jest podejmowana automatycznie (co jakiś czas) bądź przez użytkownika (w celu możliwości testowania aplikacji).

2.1.3 Funkcje

Serwer oferuje następującą funkcjonalność:

- Inicjowanie rozgrywek
- Przesyłanie Stanu gry między GM a Graczami
- Otrzymywanie od GM informacji o liczbie Graczy proszonych o wykonanie Ruchu
- Wysyłanie żądań Ruchu do graczy
- Otrzymywanie Ruchów od Graczy i ich propagacja do GM
- Otrzymywanie informacji o poprawności Ruchów od GM i ich propagacja do Graczy
- Wysyłanie informacji do Graczy o końcowym rezultacie Gry
- Kończenie rozgrywek

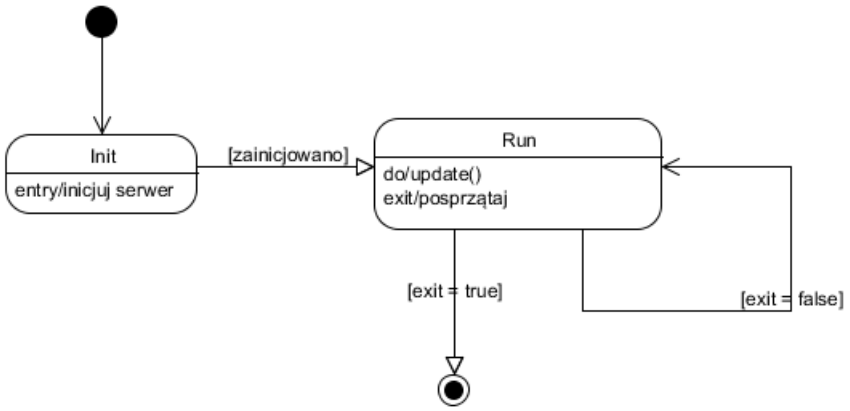
2.1.4 Główny moduł serwera



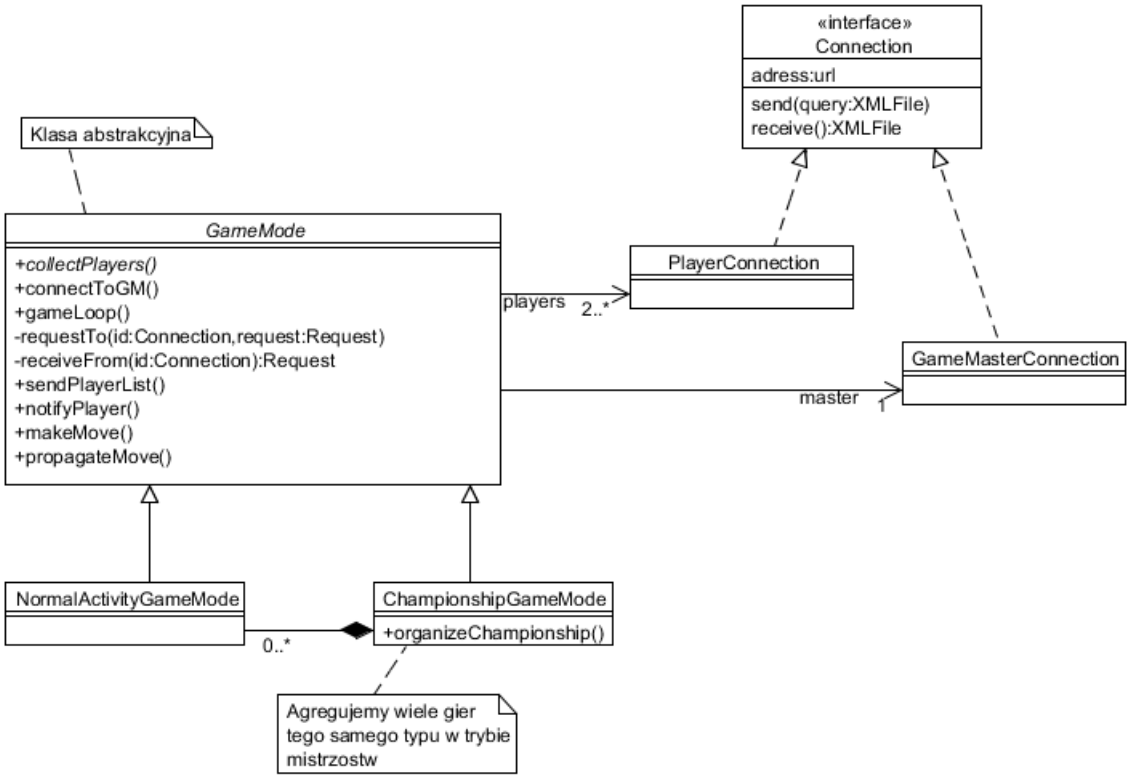
Rysunek 4: Diagram klasy Server

Klasa Serwera (`Server`) posiada informacje o rozgrywanych w danym momencie Grach (`games`). Metoda `update()` jest wykonywana cyklicznie w pętli i odpowiada za inicjowanie Gier (`initGame()` oraz `initChampionship()`) –

tworzenie nowych wątków oraz ich późniejszą obsługę (niszczenie, kwestie wyjątków itp.), a także za rejestrację Graczy. Innymi słowy jest to główna pętla programu. Gdy aplikacja kończy działanie, wykonywane są wszystkie czynności mające na celu zwolnienie zasobów. Wszyscy Gracze i GM otrzymują informacje o zakończeniu połączenia z Serwerem.



Rysunek 5: Diagram stanów obiektu serwera

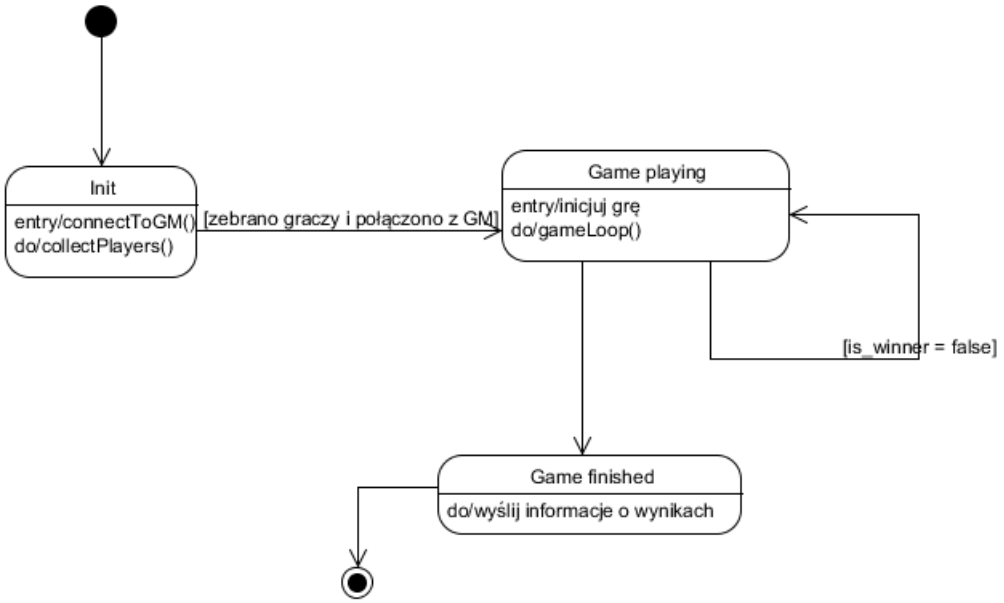


Rysunek 6: Diagram klas modułu gry

2.1.5 Moduł gry

Klasa `GameMode` to abstrakcyjna klasa Gry. Każda Gra jest obsługiwana na osobnym wątku, który posiada przypisaną mu jej klasę. W Grze mamę pulę połączeń z Graczami (`players`) oraz połączenie z GM (`master`). Połączenie (`Connection`) jest interfacem które umożliwia wysyłanie i obieranie wiadomości do i od ustalonego przez adres odbiorcy. Metoda `collectPlayers()` (abstrakcyjna, zależna czy jest to normalna gra, czy tryb mistrzostw) zbiera Graczy chętnych do Gry umiejących grać w grę określoną przez GM. Działa tu zasada kto pierwszy ten lepszy; po uzyskaniu maksymalnej liczby Graczy, Serwer kończy ich kolekcjonowanie. Wcześniej musi zostać nawiązane jednak połączenie z GM (`connectToGM()`) . Następnie po nawiązaniu połączeń Serwer przechodzi do pętli gry (`gameLoop()`) w której odbierane i wysyłane są żądania od Graczy i GM (metody `requestTo()` i `receiveFrom()`) – np. stan gry, żądania ruchu itd. (aktywności opisane dla Game Mastera i Gracza w dalszej części dokumentu). Wrapperami na to są metody takie jak

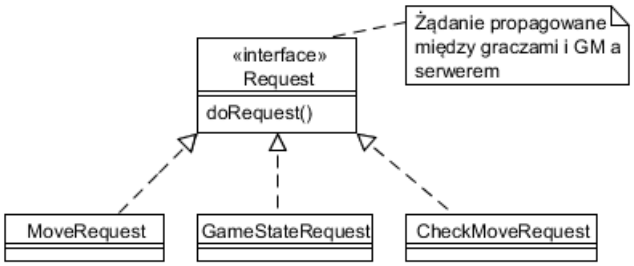
`sendPlayerList()`, `notifyPlayer()` itp. Gdy Gra się zakończy, wysyłane są informacje o końcowym stanie Gry do wszystkich Uczestników i następuje zwolnienie zasobów. Klasa `NormalActivityGameMode` jest już konkretną implementacją normalnej Gry (nie będącej Turniejem).



Rysunek 7: Diagram stanów dla obiektu gry

Żądania (interface `Request`) są przekazywane między Graczami, Serwerem a GM. Mogą one być różnego typu, zależnie od tego jak zaimplementowana jest metoda `doRequest()`. Określają one zadanie do wykonania przez odbiorcę, bądź też pewną informację. Np. Gracz chce wykonać Ruch, wtedy wysyła żądanie `MoveRequest` do Serwera. Ten tylko je odbiera i przekazuje je dalej do GM. Jest to prosty przypadek. Innym, ciekawszym jest wysłanie informacji o końcu Gry. GM wysyła żądanie końca rozgrywki, a te jest wysyłane do wszystkich Graczy.

Przedstawiony niżej diagram jest poglądowy i będzie zawierał więcej klas implementujących interface `Request`.



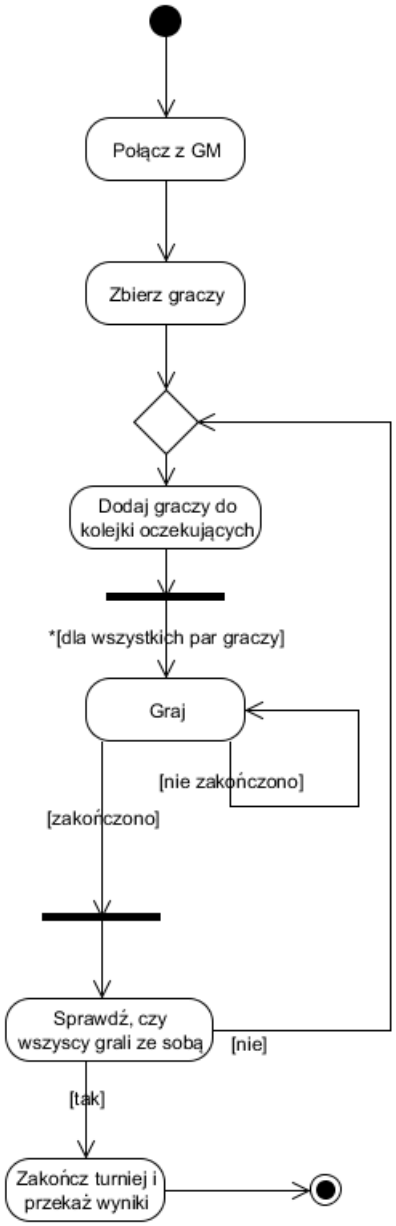
Rysunek 8: Diagram klas interface'u Request i jego pochodnych

2.1.6 Turniej

Turniej jest bardziej rozbudowanym trybem Gry. Klasa `ChampionshipGameMode` agreguje w sobie tyle Gier ile jest potrzebnych do rozegrania Turnieju. Każda z nich jest tego samego rodzaju i, zgodnie z wymogami, musi być przeznaczona dla dokładnie dwóch Uczestników.

Turniej odbywa się następująco: każdy Gracz Gra z każdym (dokładnie raz). Zatem liczba gier będzie wynosiła tyle ile liczba krawędzi w grafie pełnym prostym o N wierzhołkach (N – liczba skolekcjonowanych graczy) czyli $\frac{N(N-1)}{2}$. Każda Gra jest instancją `NormalActivityGameMode` i działa na normalnych zasadach. Jednak Serwer przechowuje informacje o wynikach poszczególnych Gier aż do zakończenia rozgrywek – Zwycięzcą jest Gracz o największej liczbie zwycięstw. Może być wielu Zwycięzców. Informacja o Wyniku jest przekazywana jak w zwykłej Grze.

Gry są rozgrywane równolegle. Algorytm zachłanny paruje Graczy którzy ze sobą jeszcze nie grali (o ile w danym momencie nie uczestniczą w Grze). Gdy gracz kończy Grę, oczekuje on wtedy w kolejce i gdy tylko znajdzie się drugi z którym nie grał, rozpoczynają rozgrywkę. Gdy już wszyscy gracze oczekują w kolejce, Turniej kończy się.



Rysunek 9: Diagram aktywności dla Turnieju

Przedstawiony diagram może być też wykorzystany dla normalnej Gry – zamiast tworzyć wiele Gier, tworzona jest jedna; nie ma też parowania Graczy i sprawdzania czy wszyscy grali.

2.1.7 Problemy

W przypadku problemów z nawiązaniem połączenia, przy tworzeniu nowej gry, serwer zaniechuje te działanie. Jeśli Gra ma się odbyć w trybie turnieju, to próba jej inicjacji będzie odbywać się do skutku (ustalone n razy). Po n+1 próbie, Turniej zostaje anulowany. Jeśli w czasie Gry utracone zostaje połączenie z Graczem – dany Gracz przegrywa. Jeśli brakuje połączenia z GM, gra musi zakończyć się i pozostaje nierozstrzygnięta.

Jeśli w czasie mistrzostw gracz nie odpowiada, przegrywa daną (jedną z wielu rozgrywanych) Grę (i tak do skutku, chyba że połączenie zostanie odnowione).

2.1.8 Uwagi implementacyjne

Program powinien posiadać parametry wywołania takie, jak: sposób rozgrywania (co jaki czas uruchomić rozgrywkę), czasy oczekiwania i ilość prób (przy braku połączenia) etc. Do testowania może się przydać określenie ilości zasobów (np. ograniczenie na liczbę Gier i Graczy). Serwer powinien wypisywać (na konsoli i pliku) maksymalnie dużo informacji (w trybie debug), albo te niezbędne (w normalnym trybie działania).

2.2 Game Master

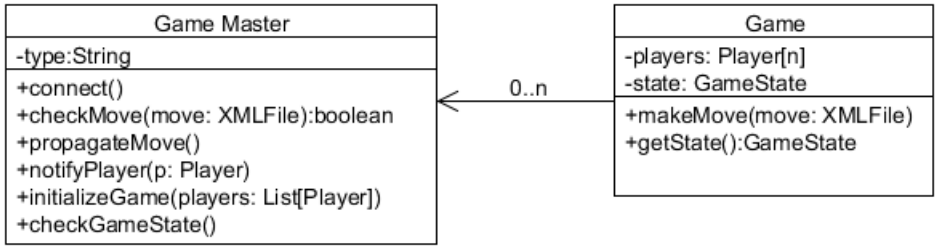
2.2.1 Funkcjonalność

Game Master jest częścią systemu odpowiedzialną za przeprowadzenie gry pomiędzy Graczami.

Aplikacja po uruchomieniu rejestruje się na Serwerze, informując go o typie obsługiwanej gry i minimalnej liczbie Graczy wymaganych do rozpoczęcia gry.

Po przydzieleniu przez Serwer wymaganej liczby Graczy, GM inicjuje rozgrywkę(ustawia grę w stan początkowy zgodny z zaimplementowanymi w nim zasadami, wybiera Gracza rozpoczynającego i rozsyła tę informację do wszystkich Graczy biorących udział w grze).

Zatem klasa aplikacji powinna zawierać następującą funkcjonalność.



Rysunek 10: Diagram klasy GameMaster

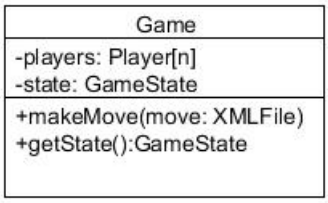
Metoda `connect()` ma na celu rejestrację na Serwerze i ustanowienie połączenia z nim. Pozostałe metody służą do monitorowania stanu gry oraz komunikacji z Graczami. Metoda `initializeGame()` tworzy nową grę(obiekt klasy Game, którym zarządza) dla Graczy zgłoszonych przez Serwer i informuje tychże graczy o początkowym stanie gry. Podejmuje również decyzję o tym, który z Graczy rozpoczyna grę i informuje go o tym przez metodę `notifyPlayer()`.

Metoda `notifyPlayer()` ma na celu poinformować Gracza, że nadeszła jego kolej i poprosić go o wykonanie ruchu.

Po otrzymaniu informacji od gracza o wykonaniu ruchy, GM sprawdza jego poprawność za pomocą metody `checkMove()`. Informacje o wykonanym ruchu oraz o stanie gry są przekazywane za pomocą protokołu XML.

Metoda `propagateMove()` służy do wykonania ruchu i poinformowania graczy biorącyh udział w rozgrywce o zmianie statu gry.

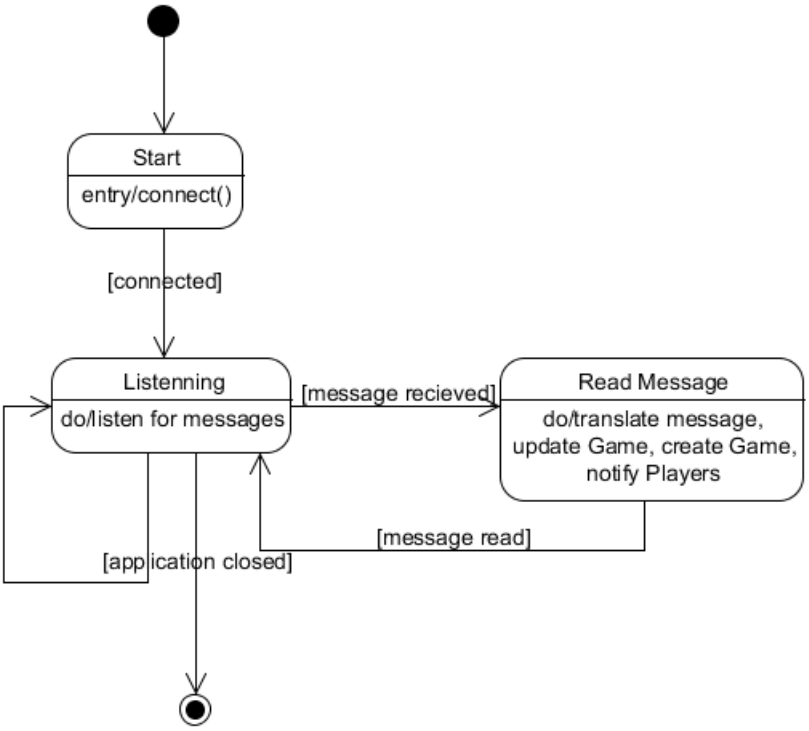
Do przeprowadzenia rozgrywki GM wykorzystuje klasę wewnętrzną klasę Game.



Rysunek 11: Diagram klasy Game

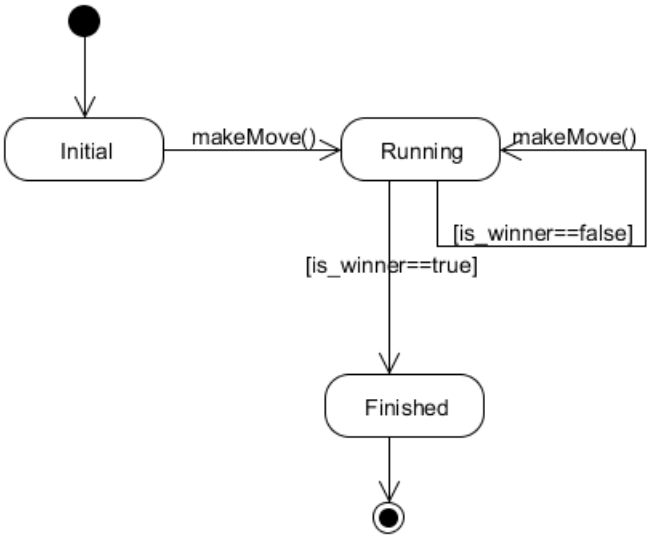
Wykorzystanie klasy pomocniczej pozwala Game Masterowi rozgrywać kilka partii danego typu gry jednocześnie poprzez utworzenie kilku instancji tej klasy dla różnych grup graczy.

Obiekt klasy Game Master nasłuchuje podczas swojego działania wiadomości od Serwera, po czym tłumaczy je i wykonuje odpowiednią czynność(tworzy nową grę na podstawie listy graczy otrzymanej od serwera, lub aktualizuje już trwającą grę) informując po tym wszystkich „zainteresowanych" Graczy(tych dla których nowa Gra została utworzona, lub tych których Gra została zaktualizowana). Zostało to przedstawione na poniższym diagramie *Diagram aktywności Game Mastera*.



Rysunek 12: Diagram aktywności Game Mastera

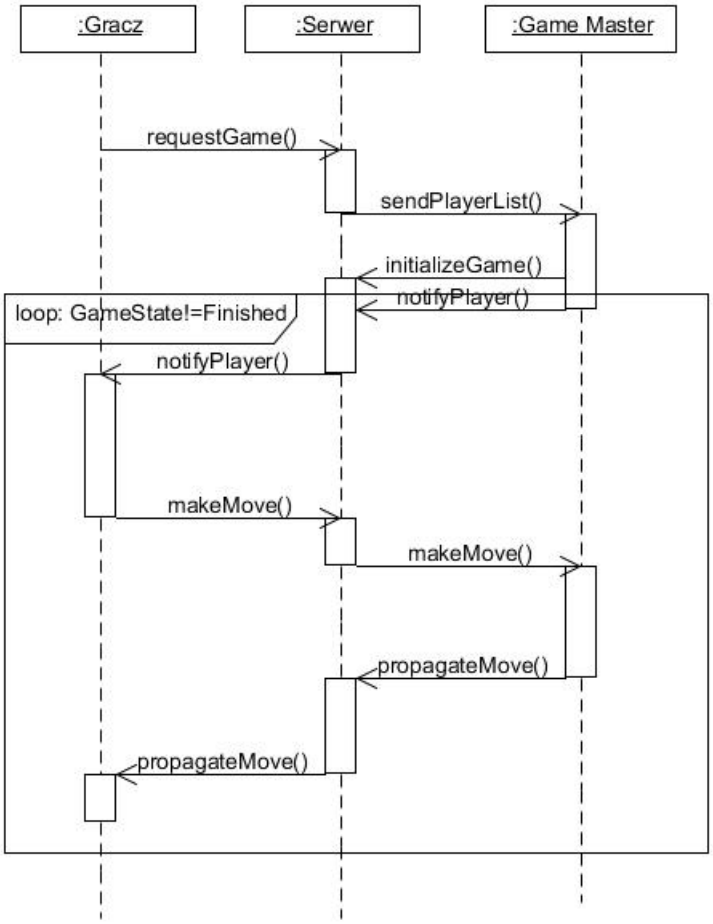
Gra może znajdować się w trzech stanach: Initial(Początkowym), Running(Trwa), oraz Finished(Ukończona).



Rysunek 13: Diagram stanów klasy Game

2.2.2 Komunikacja

Komunikacja pomiędzy Game Masterem a Graczami odbywa się tylko i wyłącznie za pośrednictwem Serwera. Nie istnieje żadna bezpośrednia komunikacja między Graczami, a Game Masterem. Komunikację tę przedstawiono na poniższym diagramie sekwencji(*Rysunek 4: Komunikacja pomiędzy GM--em, a Graczem*).



Rysunek 14: Komunikacja pomiędzy GM-em, a Graczem

Diagram przedstawia komunikację z aktywnym graczem(tym, który ma teraz wykonać ruch). GM wysyła komunikat `notifyPlayer()` tylko do gracza, do którego należy wykonanie aktualnego ruchu. Na diagramie uwzględniono tylko jednego gracza w celu zachowania czytelności diagramu.

2.2.3 Rozwiązywanie problemów

GM w celu zapewnienie rozstrzygnięcia rozgrywki ma ustalony Maksymalny czas oczekiwania na wykonanie ruchu przez Gracza.(może być on różny dla Trybu Normalnego i Trybu Championship). W przypadku, w którym GM nie otrzymał informacji o wykonaniu ruchu przez danego Gracza, gracz ten jest automatycznie uznawany za przegranego i jest wykluczony z dalszej

rozgrywki. To rozwiązanie zapewnia zakończenie rozgrywki w przypadku, gdy jeden z Graczy utraci połączenie z Serwerem(zapobiega to „wiecznemu" czekaniu na tego Gracza).

W przypadku, gdy gracz wykonał nielegalny ruch(GM uznał, że wykonany ruch jest niezgodny z zasadami), Gracz jest informowany o popełnionym błędzie i proszony o powtórzenie ruchu.(opt.: uznawany jest za przegranego i odłączany jest od rozgrywki.)

W momencie, gdy GM utracił połączenie z Serwerem, próbuje do skutku wznowić połączenie.

2.3 Gracz

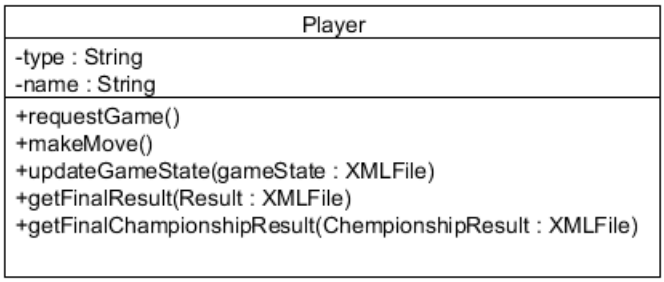
2.3.1 Funkcjonalność

Gracz jest częścią systemu odpowiedzialną za sztuczną inteligencję jednego z uczestników gry, ocenę sytuacji w grze i przesył najlepszego jego zdaniem ruchu do Serwera.

Aplikacja po uruchomieniu rejestruje się na Serwerze, informując go o typie obsługiwanej gry.

Po tym gdy Gracz zostanie przydzielony przez Serwer do gry, Gracz otrzymuje od niego stan gry, a następnie, gdy przyjdzie jego kolej, otrzymuje zapytanie o wykonanie ruchu. W tym momencie Gracz używa zaimplementowanego algorytmu aby wyliczyć najbardziej optymalny ruch, który następnie przesyła do Serwera. Ten przesyła go do Game Mastera, który ten ruch akceptuje lub nie. Jeśli nie, Gracz jest ponownie proszony o wykonanie ruchu.

Zatem klasa aplikacji powinna zawierać następującą funkcjonalność.



Rysunek 15: Diagram klasy Player

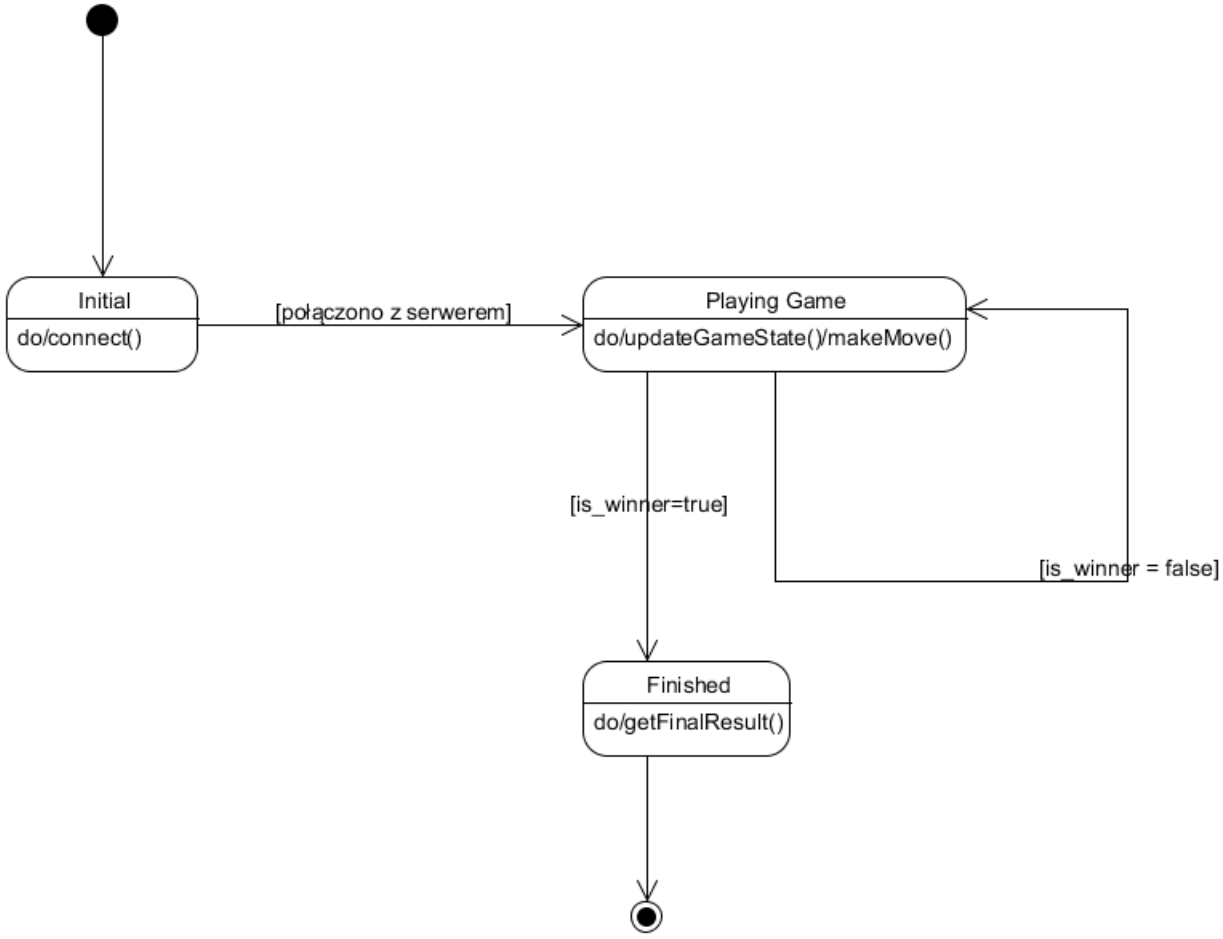
Pole `name` przechowuje nazwę Gracza, pod jaką widzą go inni Gracze. Pole `type` przechowuje nazwę gry, w jaką umie grać dany Gracz.

Metoda `requestGame()` ma na celu rejestrację na Serwerze i ustanowienie połączenia z nim.

Pozostałe metody służą do monitorowania stanu gry i wykonywania ruchu.

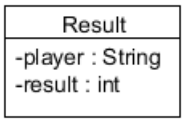
Metoda `updateGameState()` służy do zaktualizowania stanu gry, po tym gdy Gracz dostanie od Serwera informację o tym że inny gracz się ruszył. Informacja o nowym stanie gry przekazywana jest za pomocą protokołu XML. Znowu metoda `makeMove()` wywoływana jest, gdy Gracz otrzyma informację, że teraz jest jego ruch. Metoda `getFinalResult()` służy do poinformowania Gracza o ostatecznym wyniku gry.

Gracz może znajdować się w jednym z trzech stanów: Initial(Początkowym), Playing Game(w grze), Finished(po zakończeniu gry).



Rysunek 16: Diagram stanów klasy Player

Poza tym wygodne będzie wprowadzenie klasy Result, która będzie przechowywać wynik danego gracza:



Rysunek 17: Diagram klasy Result

Pole player będzie identyfikatorem danego gracza, zaś result jego wynikiem. Wynik dla pojedynczej gry będzie przyjmował wartość 1 (zwycięzca) lub 0(przegrany). Dla turnieju będzie to liczba zwycięstw w turnieju danego gracza.

2.3.2 Komunikacja

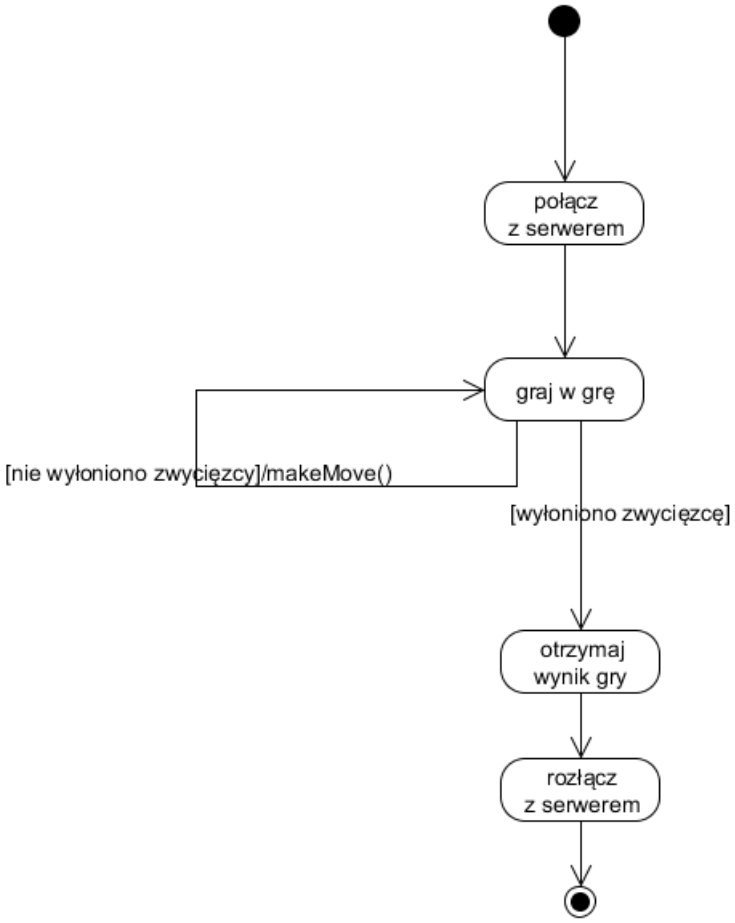
Komunikacja pomiędzy Graczem a innymi Graczami i GM-em odbywa się tylko i wyłącznie za pośrednictwem Serwera. Nie istnieje żadna bezpośrednia komunikacja między Graczami między sobą, czy też na linii Gracz-Game Master. Aby lepiej zrozumieć co Gracz może zrobić po podłączeniu się do Serwera, spójrzmy na ten diagram przypadków użycia:

Rysunek 18: Diagram przypadków użycia Serwera przez Gracza

2.3.3 Turniej

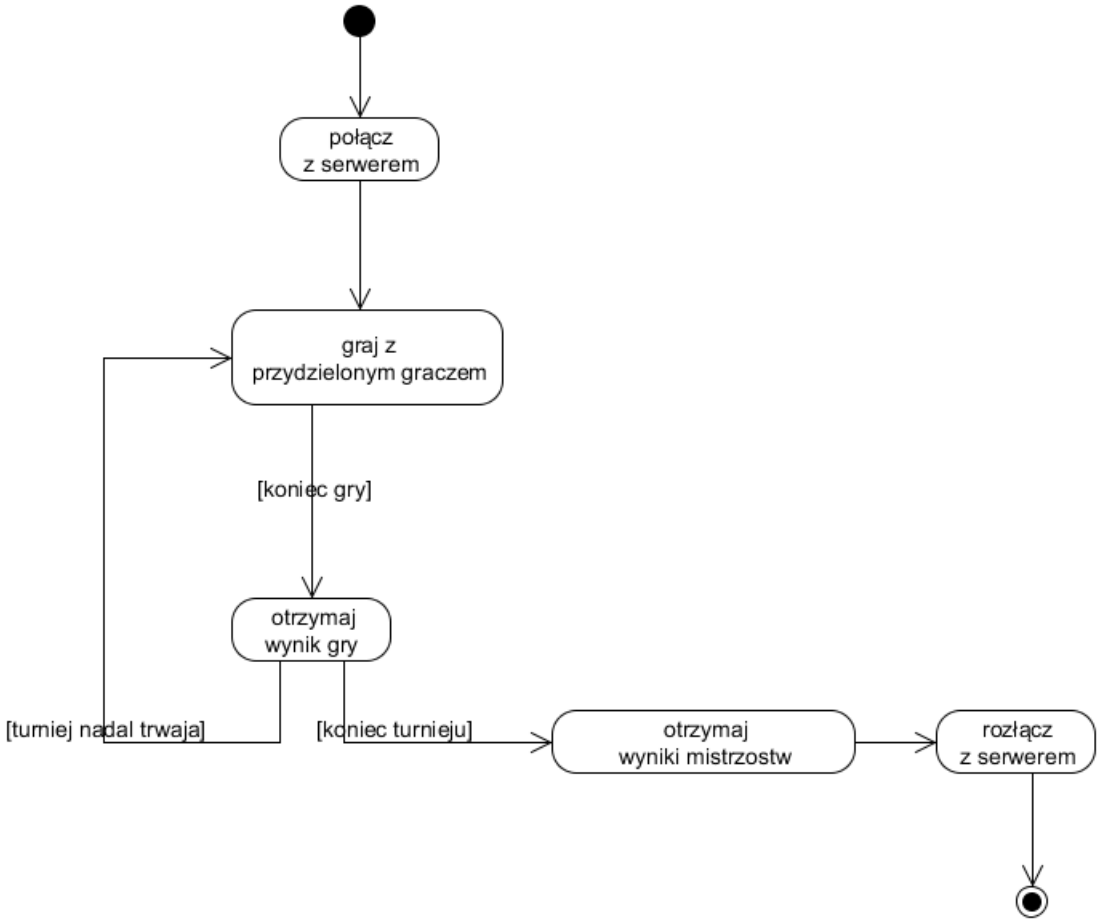
Gracz może zostać uruchomiony z parametrem "-championship". Wtedy dołącza się on do Serwera z chęcią grania w konwencji turniejowej. Gra on wtedy z każdym z innych Graczy, którzy także uczestniczą w turnieju, a ostatecznie Game Master rozsyła wyniki całego turnieju, podając liczbę zwycięstw każdego z Graczy. Ten Gracz, który uzyskał najwięcej zwycięstw, jest zwycięzcą turnieju. Aby dokładnie zrozumieć różnicę między pojedynczą grą, a mistrzostwami, najlepiej przyjrzeć się diagramom aktywności dla Gracza w tych dwóch formułach:

Gracz gra w pojedynczą grę:



Rysunek 19: Diagram aktywności dla pojedynczej gry

A tutaj Gracz gra w konwencji turniejowej:



Rysunek 20: Diagram aktywności dla turnieju

Jak widać Gracz nie rozłącza się po rozegraniu jednej gry, a gra następną, aż do końca turnieju. Potem otrzymuje wyniki całego turnieju i dopiero wtedy rozłącza się z serwerem.

2.3.4 Rozwiązywanie problemów

Jeśli ruch wykonany przez Gracza zostanie uznany przez Game Mastera za niepoprawny, Gracz może mieć zaimplementowany inny algorytm, który znajdzie tym razem poprawny ruch.

W przypadku gdy Gracz utracił połączenie z Serwerem, próbuje do skutku wznowić połączenie. Jeśli uda mu się wznowić połączenie, jednak zrobił to za późno i został już wykluczony z dalszej rozgrywki, wykonuje ponownie

metodę `requestGame()` i próbuje zagrać w nową grę.

A Załączniki

1. UGSProtocolXMLSchema.xsd - Przykładowy plik XSD