**ZPR projekt:**

**klon gry Agar.io**



**Kajetan Śpionek**

**Wojciech Przybysz**

**1. Struktura klas serwera**

W projekcie wykorzystano przykład *CHAT\_SERVER* ilustrujący wykorzystanie biblioteki boost::asio do stworzenia serwera http, udostępniony pod licencją *Boost Software License,* która umożliwia bezpłatne użycie, modyfikacje i rozprzestrzenianie kodu. Całość kodu z przykładu umieszczono w przestrzeni nazw *http.* Na tę przestrzeń nazw składają się struktury:

Header - definiująca nagłówek http

Reply - definiująca strukturę odpowiedzi i jej przetwarzanie

Request - definująca zapytanie http

oraz klasy:

RequestHandler

RequestParser

zajmujące się poprawnym przetwarzaniem zapytania.

Klasy: Dataframe i DataframeParser - zajmują sie przetwarzaniem ramek zgodnie z protokołem WebSocket (RFC 6455).

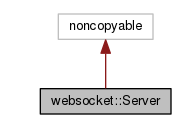
Klasy: GameBoard, Element, Ball, FoodItem - implementują logikę gry.

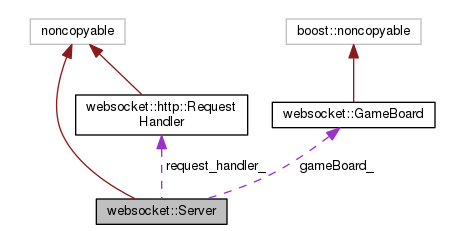
Klasy: Player, Session, Server - implementują nawiązywanie połączenia i przetwarzanie komunikatów.

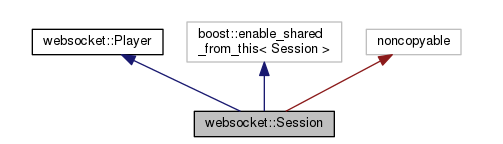
Serwer wykorzystuje asynchroniczną komunikację boost::asio. W pliku server.cpp są zaimplementowane metody tworzące nową sesję tcp, której działanie jest zawarte w pliku session.cpp, jak również tworzona jest plansza gry GameBoard Klasa Session posiada odwołania do odpowiednich parserów, stanów połączenia oraz do metod klasy GameBoard, gdzie zaimplementowana jest logika gry.

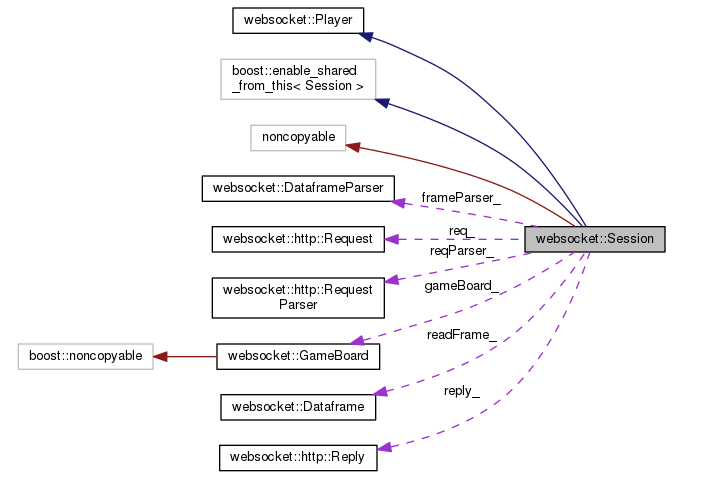
**DIAGRAM KLAS:**

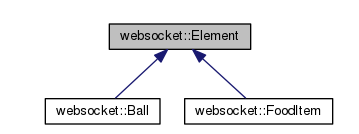
**Klasa Server**

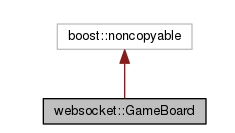




****

****

****

****

**2. Struktura połączenia**

Połączenie nawiązywane jest przez klienta, który wysyła ramkę *newPlayerName:* ze swoim nickiem. Serwer sprawdza dostępność i odsyła ramkę zwrotną *newPlayerValidNick:* z informacją *OK* bądź *TAKEN,* co pozwala klientowi na ewentualną zmianę nicku. Następnie klient wysyła ramkę gotowości do gry *newPlayerStatus: .* Serwer na informację o gotowości do gry klienta przesyła:

-*mapSize:* aktualne wymiary mapy,

-*gameState:* aktualny stan gry, czyli położenie kulek,

-*newBall:* przypisuje kulkę graczowi i informuje innych graczy,

-*newFoodItem:* dodaje n nowych statycznych kulek do gry.

Następnie rozpoczyna się właściwa gra. Klient wysyła do serwera unormowane do jedynki położenia myszki. Serwer przetwarza zmianę położenia, ustala prędkość kulki według jej aktualnego promienia. Jeśli w obrębie nowego położenia kulki znajduje się inna kulka o mniejszym promieniu, zostaje ona ‘zjedzona’ - usunięta z gry. Wtedy promień kulki, która zjadła inną kulke się powiększa oraz zostają zapisane jej statystyki.

**3. Zrealizowane funkcjonalności**

**4. Planowany czas, a rzeczywista czasochłonność projektu.**

|  |  |  |
| --- | --- | --- |
| **Zadania** | **Czas planowany** | **Czas**  **rzeczywisty** |
| Implementacja Serwera | 35 | 57 |
| * Stworzenie interfejsu dla WebSocket | 10 | 30 |
| * Podstawowe połączenie z pojedynczym klientem | 3 | 2 |
| * Nawiązanie komunikacji z wieloma klientami | 7 | 2 |
| * Implementacja logiki gry | 5 | 10 |
| * Przesyłanie docelowych danych do klientów | 5 | 5 |
| * Testowanie i korekcja błędów | 5 | 8 |
|  |  |  |
| Implementacja Klienta | 15 |  |
| * Stworzenie podstawowego interfejsu przesyłania danych | 4 |  |
| * Stworzenie interfejsu graficznego | 3 |  |
| * Stworzenie docelowego schematu komunikacyjnego | 5 |  |
| * Testowanie | 3 |  |
|  |  |  |
| Sumaryczny czas: | 50 |  |