APRO2 Projekt UML - AZUL

Kamil Stachowicz, Paweł Sadowski, Sebastian Pietrzak

Spis treści

1 Założenia projektowe

1.1 Architektura

Projekt składa się z dwóch części: sieciowej (klasy Server i Client), oraz reprezentującej stan gry.
Klasy reprezentujące stan gry i graczy nie zawierają żadnych referencji do części sieciowej, zawierają jedynie dane, co umożliwia łatwe zserializowanie i wymienę danych.
Część sieciowa odowiada za wymianę danych pomiędzy serwerem, klientem i użytkownikiem, oraz zarządzanie grą.
Serwer reprezentuje tylko i wyłącznie jednostkę hostującą i prowadzącą grę, nie jest ściśle związany z żadnym graczem, klient natomiast jest ściśle związany z jednym graczem.
W celu nie tworzenia dodatkowych szczególnych przypadków w kodzie backendowym zdecydowaliśmy, że zarówno rozgrywka zdalna jak i lokalna, będzie realizowana przez stworzenie serwera i połączenie z nim po jednym kliencie dla

każdego gracza. Aplikacja frontendowa powinna w przypadku rozgrywki lokalnej zająć się stworzeniem lokalnego serwera i połączeniem z nim odpowiedniej ilości klientów.

1.2 Komunikacja

Na etapie projektowania wyróżniamy rodzaje wiadomośći (wymienione w *Typ enumeracyjny: Protocol*), które będą wymieniane pomiędzy klientem a serwerem. Nie narzucamy konkretnej (binarnej/tekstowej) reprezentacji tych wiadomości.
Elementem identyfikującym gracza jest jego nazwa.

- Komunikację rozpoczyna klient łącząc się z serwerem i wysyłając mu wiadomość *HELLO* wraz ze swoją nazwą gracza.
- Serwer po otrzymaniu wiadomości *HELLO* od nowego klienta dodaje obiekt gracza do gry, lub jeśli gracz o takiej nazwie jest już w grze, ale rozłączył się, przydziela klientowi istniejący obiekt gracza o danej nazwie. Od tego momentu serwer uznaje połączenie z klientem za nawiązane.
- O każdej zmianie stanu gry, serwer informuje wszystkich połączonych kientów wiadomością *UPDATEGAME*, każdorazowo wysyłając każdemu z nich pełny zserializowany stan gry (obiektu game).
- Klient informuje serwer o wykonaniu ruchu przez gracza wiadomością *TAKEFROMCENTER* lub *TAKEFROMFACTORY*. Jeżeli jest to ruch prawidłowy serwer odpowiednio aktualizuje stan gry.
- Rozłączenie się klienta nie skutkuje usunięciem go ze stanu gry - możliwe jest ponowne dołączenie z zachowaniem stanu (warunkiem jest użycie tej samej nazwy gracza).
- Operator serwera może zatrzymać serwer - rozłączyć wszystkich graczy bez usuwania ich ze stanu gry.
- Operator zatrzymanego (lub nie wystartowanego) serwera może zapisać i wczytać stan gry do/z pliku.
- Operator serwera może wyrzucić danego gracza z gry. Skutkuje to rozłączeniem danego klienta i usunięciem odpowiedniego gracza z stanu gry.

2 Klasy oraz metody

2.1 Klasa: Game

Reprezentuje grę Azul jako całość, kontroluje przebieg rozgrywki, zarządza rundami, a także przechowuje listę graczy, fabryki i wyświetlacz środkowy.

Metody

`playTurn(): void`

Wykonuje turę dla aktualnego gracza.

`nextPlayer(): void`

Przechodzi do kolejnego gracza.

`isRoundEnded(): bool`

Sprawdza, czy runda się zakończyła.

`isGameEnded(): bool`

Sprawdza, czy gra się zakończyła.

`calculateScores(): void`

Oblicza wynik dla każdego gracza.

`resetRound(): void`

Resetuje stan rundy (czyszczenie fabryk, planszy itp.).

`displayGameState(): void`

Wyświetla stan gry, takie jak punkty, plansze graczy, fabryki itp.

2.2 Klasa: Player

Reprezentuje gracza w grze Azul. Przechowuje informacje, takie jak nazwa gracza, plansza gracza (PlayerBoard) i aktualny wynik.

Metody

`takeTiles(factoryIndex: int, color: String): void`

Gracz pobiera kafelki tego samego koloru z wybranej fabryki lub ze środka stołu.

`addToPatternLines(tile: Tile, lineIndex: int): void`

Gracz umieszcza kafelki na swojej planszy w jednym z rzędów wzorów.

`addToFloorLine(tile: Tile): void`

Gracz umieszcza nadmiar kafelków na swojej linii podłogi.

`moveTilesToWall(): void`

Gracz przenosi kafelki z ukończonych rzędów wzorów na ścianę.

`updateScore(points: int): void`

Aktualizuje wynik gracza po każdej rundzie lub podczas końcowego podsumowania.

2.3 Klasa: FactoryDisplay

Reprezentuje fabrykę w grze Azul. Fabryka przechowuje kafelki, które gracze mogą wybrać podczas swojej tury.

Metody

`refillTiles(tiles: List<Tile>): void`

Napełnia fabrykę kafelkami z worka.

`removeTiles(color: String): void`

Usuwa wszystkie kafelki tego samego koloru z fabryki.

`isEmpty(): bool`

Sprawdza, czy fabryka jest pusta.

2.4 Klasa: CenterDisplay

Reprezentuje centralne miejsce na stole, gdzie gracze przenoszą kafelki z fabryk.

Metody

`addTiles(tiles: List<Tile>): void`

Dodaje kafelki do środka stołu.

`removeTiles(color: String): void`

Usuwa wszystkie kafelki tego samego koloru ze środka stołu.

`takeFirstPlayerToken(): void`

Gracz pobiera żeton pierwszego gracza, jeśli jest dostępny.

2.5 Klasa: Tile (kontynuacja)

Reprezentuje kafelek w grze Azul. Posiada atrybut koloru, który określa rodzaj kafelka. Brak dodatkowych metod, ponieważ ta klasa przechowuje tylko informacje o kolorze kafelka.

2.6 Klasa: PlayerBoard

Reprezentuje planszę gracza w grze Azul. Zawiera rzędy wzorów (pattern lines), ścianę (wall) i linię podłogi (floor line).

Metody

`addToPatternLines(tile: Tile, lineIndex: int): void`

Dodaje kafelek do jednego z rzędów wzorów.

`moveTilesToWall(): void`

Przenosi kafelki z ukończonych rzędów wzorów na ścianę, przyznając punkty za odpowiednie połączenia.

`addToFloorLine(tile: Tile): void`

Dodaje kafelek do linii podłogi.

`clearPatternLines(): void`

Usuwa kafelki z rzędów wzorów.

`clearFloorLine(): void`

Usuwa kafelki z linii podłogi.

2.7 Klasa: Bag

Reprezentuje worek z kafelkami w grze Azul. Przechowuje wszystkie kafelki i zawiera metody do dodawania, losowania i sprawdzania, czy worek jest pusty.

Metody

`addTiles(tiles: List<Tile>): void`

Dodaje kafelki do worka.

`drawTiles(n: int): List<Tile>`

Losuje n kafelków z worka.

`isEmpty(): bool`

Sprawdza, czy worek jest pusty.

2.8 Klasa: TileDispenser

Reprezentuje dystrybutor kafelków w grze Azul. Zarządza workiem z kafelkami i stosie odrzuconych kafelków.

Metody

`drawTiles(n: int): List<Tile>`

Losuje n kafelków z worka.

`discardTiles(tiles: List<Tile>): void`

Odrzuca użyte kafelki, przenosząc je na stos odrzuconych kafelków.

`refillBag(): void`

Uzupełnia worek kafelkami ze stosu odrzuconych kafelków.

2.9 Klasa: Server

Reprezentuje serwer prowadzący grę wraz z obecnym stanem gry.

Pola

`port: int`

Port, na którym serwer ma nasłuchiwać.

`game: Game`

Obecny stan gry.

`serverSocket: ServerSocket`

Socket, na którym serwer nasłuchuje.

`clients: List<Socket>`

Lista podłączonych klientów.

`clientPlayers: Map<Client, Player>`

Mapa przyporządkowująca socketom, które przesłały prawidłową wiadomość HELLO, obiekt reprezentujący gracza.

Metody

`start(): void`

Rozpoczyna pracę serwera (nasłuchiwanie).

`stop(): void`

Kończy pracę serwera (nasłuchiwanie).

`startGame(): void`

Tworzy i uruchamia nową grę.

`stopGame(): void`

Siłowo kończy grę (zmienia na null i informuje o tym klientów).

`kickPlayer(Player p): void`

Wyrzuca gracza z gry.

`saveGame(File f): void`

Zapisuje stan gry w pliku.

`loadGame(File f): void`

Wczytuje stan gry z pliku.

Wyrzuca danego gracza z gry (rozłącza klienta).

2.10 Klasa: Client

Reprezentuje klienta (gracza) gry wraz z znanym przez niego stanem gry.

Pola

`socket: Socket`

Socket, którym klient połączony jest z serwerem.

`player: Player`

Najbardziej aktualny obiekt reprezentujący stan gracza.

`game: Game`

Najbardziej aktualny obiekt reprezentujący stan gry.

`playerName: String`

Nazwa (identyfikator) gracza.

Pola

`connect(String server): void`

Próbuje rozpocząć połączenie z serwerem.

`disconnect(): void`

Rozłącza się z serwerem.

2.11

Typ enumeracyjny: Protocol Zawiera wiadomości protokołu komunikacyjnego pomiędzy klientem a serwerem.

- HELLO - wiadomość wysyłana przez klienta po nawiązaniu połączenia. Powinna zawierać nazwę gracza, który chce dołącza do serwera.
- UPDATEGAME - wiadomość wysyłana przez serwer, informująca klienta o aktualizacji stanu gry. Powinna zawierać zserializowany stan gry.
- TAKEFROMCENTER - wiadomość wysyłana przez klienta informująca o chęci wykonania ruchu - wzięcia płytek ze środka. Powinna zawierać wybrany przez gracza kolor płytek.

- TAKEFROMFACTORY - wiadomość wysyłana przez klienta informująca o chęci wykonania ruchu - wzięcia płytek z warsztatu. Powinna zawierać numer wybranego warsztatu i kolor płytek.