

Marian Stopyra 164014 P3

2EF-DI 2021/2022

C++ projekt – dokumentacja

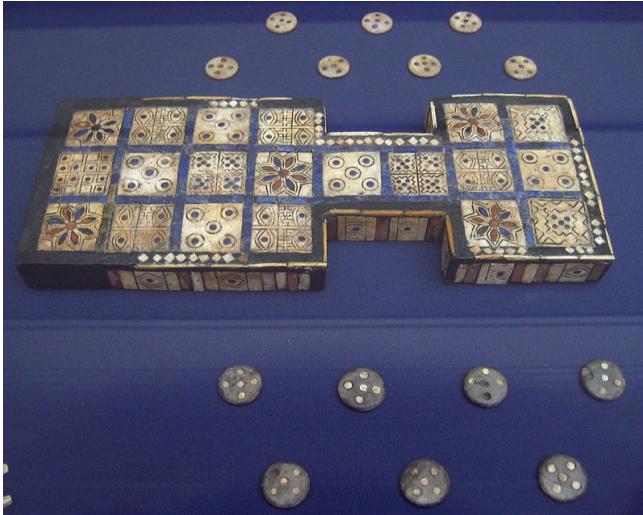
Temat projektu: Gra planszowa - Królewska Gra z Ur

Spis treści:

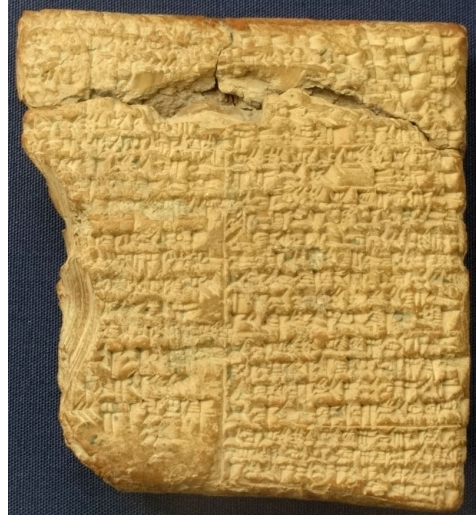
Opis i zasady gry	str. 2
Instrukcja obsługi programu	str. 4
Dokumentacja techniczna:	
Klasa Pawn	str. 9
Klasa Player	str. 10
Klasa Board	str. 11
main i zmienne globalne	str. 13

1. Królewska Gra z Ur – opis i zasady gry

Nazwana po mezopotamskim mieście Ur, w którego cmentarzu królewskim pierwsze egzemplarze tej gry zostały wydobyte, jest to najstarsza gra jakiej zasady są dziś znane. Najstarsze egzemplarze datowane są na rok 2600 p.n.e.



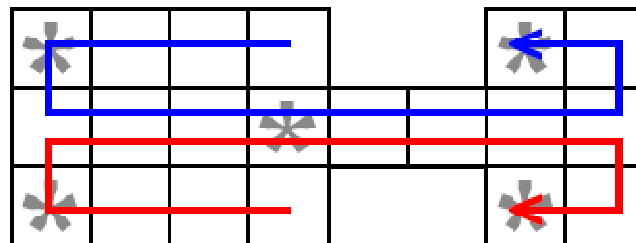
Plansza i pionki do gry z Ur na wystawie w Muzeum Brytyjskim w Londynie



Tabliczka z zasadami datowana na 177r. p.n.e.

Gra przeznaczona jest dla dwóch graczy.

Rozgrywka opiera się o wyścig przez planszę na zaznaczonej na rysunku poniżej trasie.



Trasa jaką przechodzą poszczególni gracze

Każdy gracz ma własną część trasy po swojej stronie, ale środek planszy jest współdzielony. W tej części możliwy jest konflikt pomiędzy graczami.

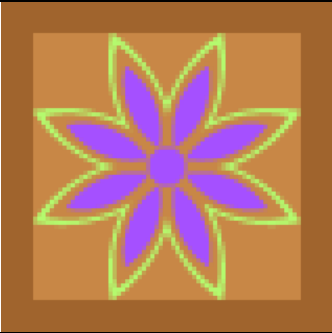

Gracze rzucają naprzemiennie 4 czworościennymi kostkami, których rogi zaznaczone są białymi lub czarnymi kropkami, dając możliwość wyrzucenia od 0 do 4 białych kropek.



Nowożytna wersja kości do gry z Ur

Ruch do przodu o wyrzuconą liczbę białych kropek można wykonać dowolnym pionkiem, także wstępując nowym pionkiem na planszę, ale nie można ich dzielić pomiędzy kilka pionków.

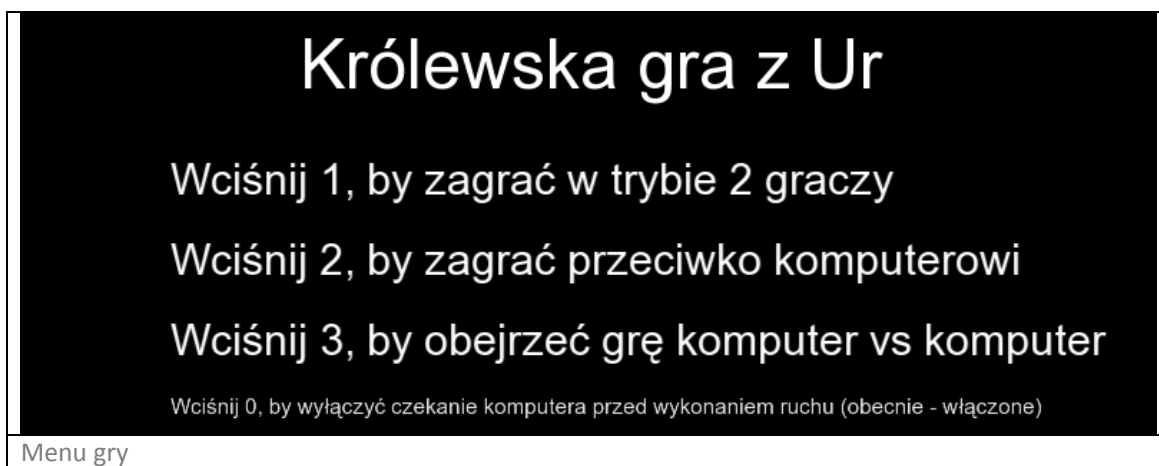
Wylądowanie na rozetce, nagradza gracza dodatkowym ruchem, oraz chroni stojący na niej pionek przed zbiciem.

	
<p>Rozetka (pole specjalne) w programie</p>	<p>Rozetka na oryginalnej, historycznej planszy</p>

Wygrywa gracz, który jako pierwszy dotrze wszystkimi 7 swoimi pionkami do mety.

2. Instrukcja obsługi programu

Pierwsze co widzi gracz po uruchomieniu gry, to menu z wyborem trybu.

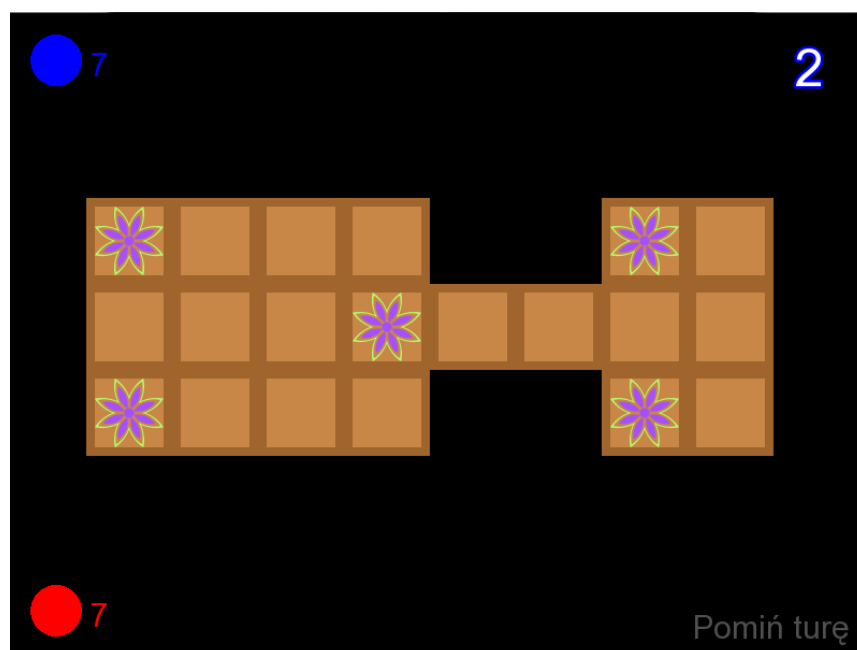


Menu pozwala wybrać tryb gracz vs gracz (po wciśnięciu przycisku 1 na klawiaturze), gracz vs komputer (przycisk 2), i komputer vs komputer (przycisk 3).

Pozwala też wybrać czy komputer ma czekać chwilę przed wykonaniem ruchu (by dać graczowi szansę prześledzić swoje akcje), czy może reagować natychmiast.

Pozostała część instrukcji napisana jest w większości dla przypadku gracza 1 (niebieskiego), kontrolowanego w trybach gracz vs gracz, i gracz vs komputer.

Po wybraniu trybu ukazuje nam się okno gry.



Widok okna gry

Na planszy nie ma jeszcze postawionych żadnych pionków.

Jednak są widoczne inne ważne elementy:



Licznik pionków poza planszą gracza niebieskiego



Licznik pionków poza planszą gracza czerwonego



Wskaźnik wyrzuconej liczby oczek



Przycisk zrezygnowania z ruchu

Licznik pionków poza planszą gracza niebieskiego: Pokazuje ile pionków jeszcze pozostało graczowi niebieskiemu do postawienia na planszy. Kliknięcie na ikonkę pionka po lewej stronie licznika pozwala wejść na planszę nowym pionkiem.

Licznik pionków poza planszą gracza czerwonego: Działa analogicznie, ale dla gracza czerwonego.

Wskaźnik wyrzuconej liczby oczek: Pokazuje odległość na jaką w danej rundzie można się poruszyć.

Jako że gra symuluje rzut 4 kostkami, z których każda może wyrzucić 1 lub 0 oczek, możliwości są takie same jak przy rzucie 4 monetami:

0 – prawdopodobieństwo wyrzucenia: $1/16$ (6.25%)

1 – prawdopodobieństwo wyrzucenia: $1/4$ (25%)

2 – prawdopodobieństwo wyrzucenia: $3/8$ (37.5%)

3 – prawdopodobieństwo wyrzucenia: $1/4$ (25%)

4 – prawdopodobieństwo wyrzucenia: $1/16$ (6.25%)

Kolorowa obłamówka pokazuje czyj ruch jest w danej rundzie.

Dla przykładu:

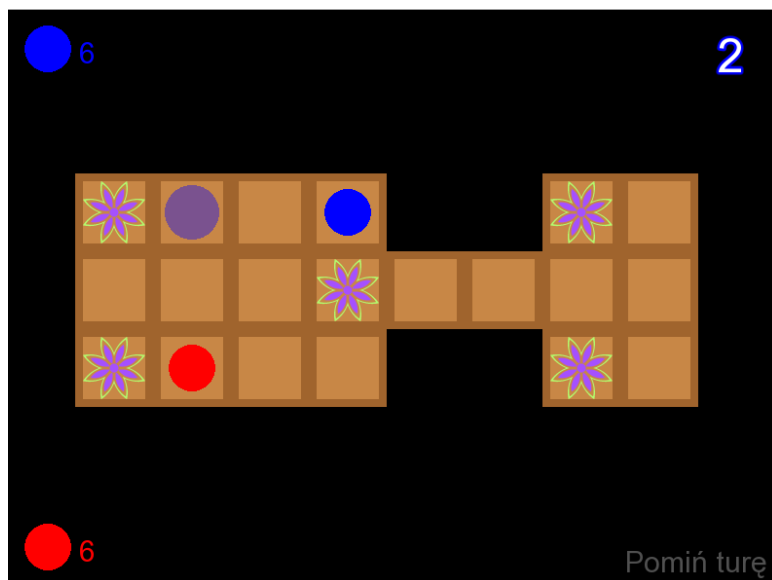


Gracz niebieski
może wykonać ruch
o 1 pole



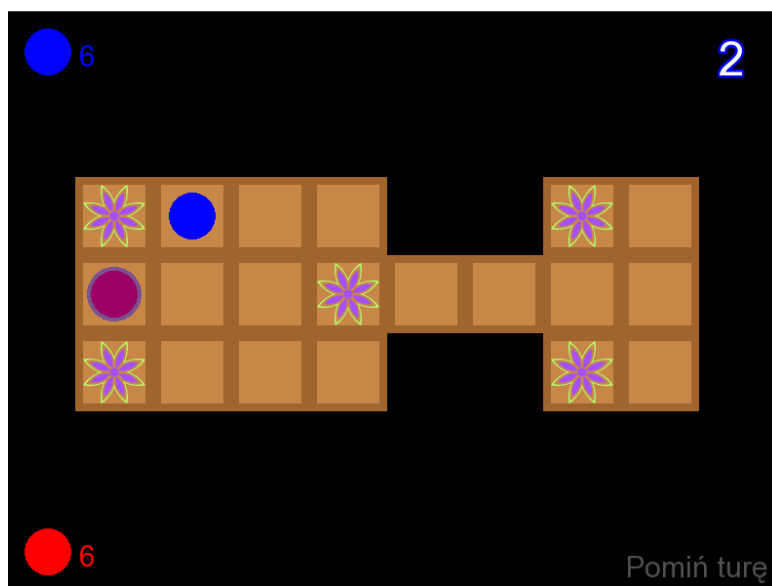
Gracz czerwony
może wykonać ruch
o 2 pola

Po ustawieniu pionka na planszy, można wykonać nim ruch poprzez kliknięcie na niego. Po najechaniu ukazuje się podgląd ruchu.



Podgląd przesunięcia pionka o 2 pola

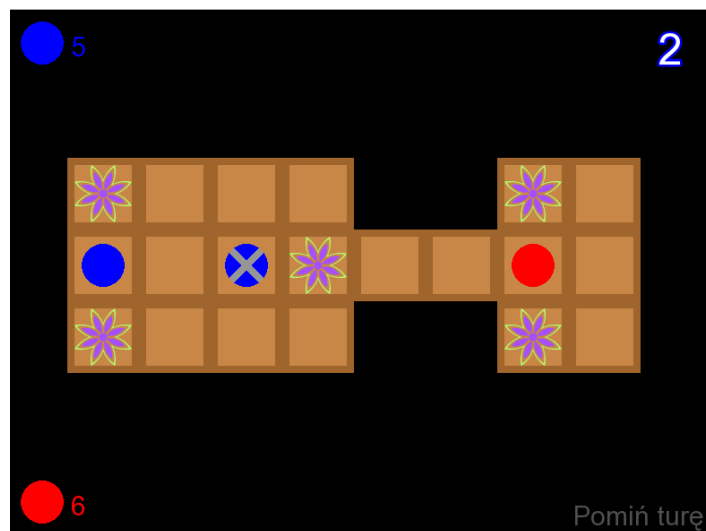
Postawienie pionka na miejscu zajęтым przez przeciwnika pozwala go skuć.



Skuwanie pionka gracza czerwonego przez gracza niebieskiego

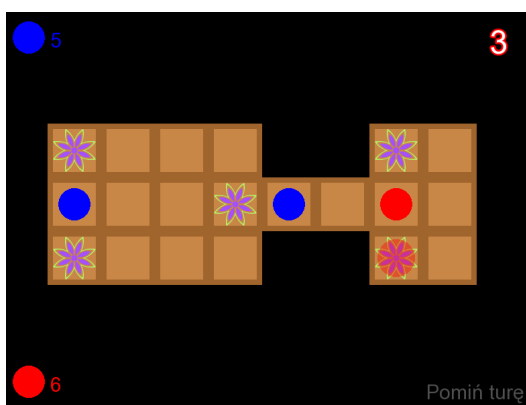
Zbity pionek wraca do Puli pionków czekających na wejście na planszę.

Nie można postawić pionka na polu zajęтым przez własny pionek.

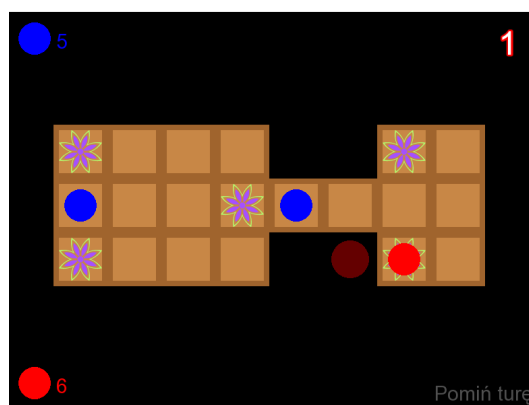


Próba ruchu gracza niebieskiego na pole już przez niego zajęte – ruch niedozwolony.
Jeśli nie można wykonać żadnego ruchu, można pominąć turę.

Ruch na rozetkę nagradza gracza dodatkowym ruchem.



Podgląd ruchu gracza czerwonego na rozetkę

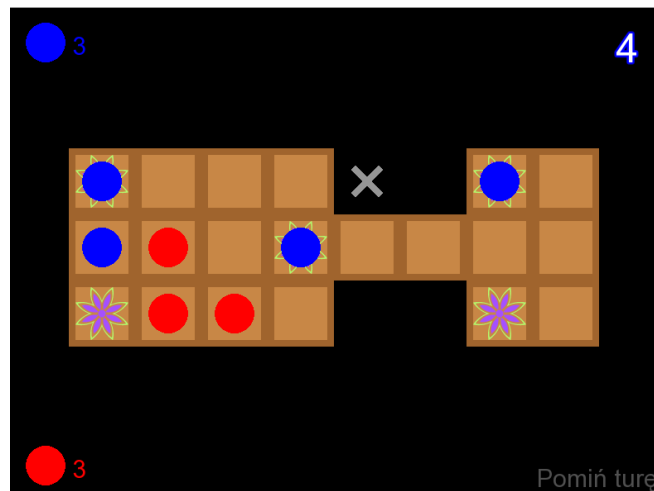


Następna tura – dodatkowy ruch gracza czerwonego

Powyższy obrazek pokazuje też możliwość zejścia z planszy.

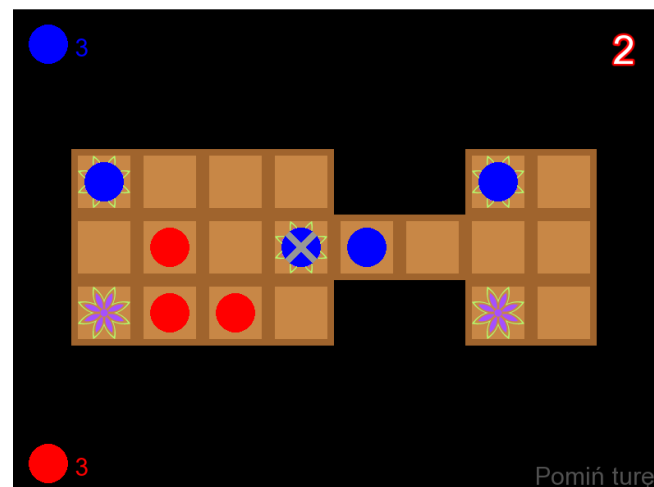
Po zejściu z planszy, pionek nie jest doliczany z powrotem do pionków czekających na wejście na planszę.

Żeby zejść z planszy trzeba wyrzucić dokładną wymaganą liczbę oczek.



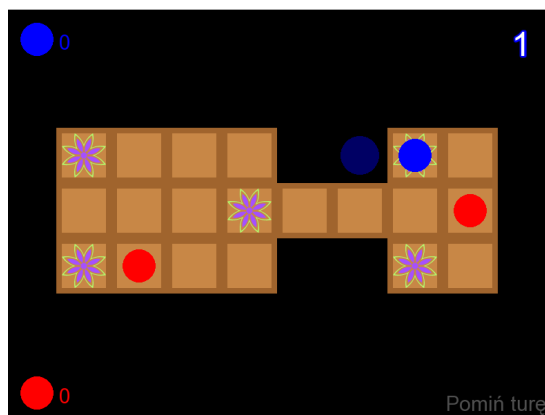
Wyrzucenie zbyt dużej liczby oczek by zejść z planszy – niedozwolony ruch.

Stojący na rozetce pionek jest chroniony przed byciem skutym.

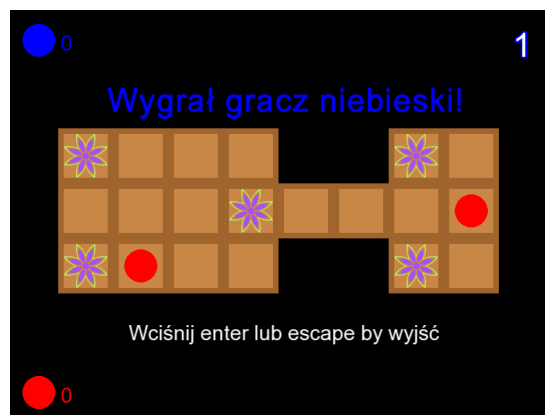


Próba skucia pionka stojącego na rozetce – niedozwolony ruch

Gra kończy się, gdy wszystkie pionki jednego z graczy zejdą z planszy.



Podgląd ruchu gracza niebieskiego tuż przed wygraną gry



Gracz niebieski wygrywa grę

3. Dokumentacja techniczna

Klasa Pawn:

```
friend class Player;  
friend class Board;
```

Upoważnia klasy Player i Board do korzystania z wszystkich pól składowych i metod, gdyż są one prywatne, dla zablokowania dostępu do nich z poziomu zakresu main.

Pola:

```
private:    sf::RenderWindow* ptrWindow;  
            Przekazuje wskaźnik do okna (obiektu sf::RenderWindow)  
            Zapewnia funkcjom rysującym dostęp do metody sf::RenderWindow::draw  
  
private:    int ownerID;  
            Określa gracza będącego właścicielem pionka.  
  
private:    sf::Color color;  
            Określa kolor pionka.  
  
private:    int tileOnPath = 0;  
            Określa pole na trasie (Player::path) na którym znajduje się pionek.
```

Funkcje i metody:

```
private:    Pawn(sf::RenderWindow* ptrWindow, int playerID, sf::Color pawnColor);  
            Konstruktor klasy Pawn.  
            Przyjmuje jako argument wszystkie swoje składowe, poza położeniem na ścieżce.  
  
private:    void drawPawn(float x, float y, bool larger = false,  
bool transparent = false)const;  
            Metoda rysująca pionek na dowolnych współrzędnych w obrębie okna.  
            bool larger – określa czy pionek ma być narysowany powiększony.  
            bool transparent – określa czy pionek ma być narysowany przezroczysty.  
            Używany do wyświetlania przewidywanych ruchów po najechaniu na pionek.  
  
private:    void drawPawn(sf::Vector2i tile, bool larger = false,  
bool transparent = false, bool forbidden = false)const;  
            Rysuje pionek na planszy na polu o wskazanych współrzędnych.  
            Otrzymane współrzędne planszy tłumaczy na współrzędne w obrębie okna, po czym  
            podaje otrzymane parametry do metody drawPawn(float x, float y, ...).  
            bool forbidden – sygnalizuje metodzie, że ma przekazać współrzędne do metody  
            drawForbidden(float x, float y)  
  
private:    void drawForbidden(float x, float y)const;  
            Zamiast pionka rysuje na podanych współrzędnych szary X.  
            Sygnalizuje ruchy niedozwolone.  
  
friend bool operator==(Pawn Lvalue, Pawn Rvalue);  
            Redefinicja operatora porównania.  
            Przyjmuje dwa pionki jako argumenty.  
            Sprawdza, czy dwa pionki to tak naprawdę ten sam pionek.
```

Klasa **Player**:

Pola:

```
private:    sf::RenderWindow* ptrWindow;
            Przechowuje wskaźnik do okna podawany pionkom, oraz używany przy rysowaniu HUDu

private:    const sf::Font* ptrFont;
            Przechowuje wskaźnik na czcionkę, żeby nie musieć wczytywać jej a każdym
            wywołaniem funkcji rysujących HUD.

private:    sf::Vector2i path[20];
            Wektor koordynatów na planszy, odpowiadający ścieżce po której poruszają się
            pionki.
            Osobny dla każdego gracza, gdyż ścieżki różnią się od siebie.

private:    int inactivePawns = 7;
            Przechowuje liczbę pionków oczekujących na wejście na planszę.

private:    std::vector<Pawn> activePawns;
            Przechowuje listę pionków znajdujących się obecnie na planszy.

public:     int ID;
            Przechowuje czy jest to gracz 1, czy gracz 2.

public:     bool isBot;
            Przechowuje informację o tym, czy gracz jest kontrolowany przez computer,
            czy przez użytkownika.

public:     sf::Color pawnColor;
            Określa kolor pionków należących do danego gracza.
```

Funkcje i metody:

```
friend class Board;
            Daje klasie Board dostęp do pól i metod, do których dostęp jest ustawiony na
            prywatny.

private:    Pawn* newActivePawn(int startingPosition);
            Wstawia na planszę nowy pionek, i ustawia go na podaną pozycję na ścieżce.
            Zwraca wskaźnik do nowo utworzonego pionka.

public:     void drawInactivePawnHUD(bool isLarger = false) const;
            Rysuje licznik pionków czekających na wejście na planszę, oraz ikonkę pionka,
            której kliknięcie stawia na planszy nowy pionek.
            bool isLarger - powiększa ikonkę po najejchaniu na nią.

public:     void drawDiceRollHUD(int rolledToMove) const;
            Rysuje HUD pokazujący wyrzuconą liczbę oczek, oraz obłamówkę pokazującą czyja
            jest tura.
            int rolledToMove - liczba wyrzuconych oczek.

public:     bool winCondition() const;
            Zwraca sprawdzenie, czy dany gracz spełnia warunki wygrania gry.
```

```

public:      Player(sf::RenderWindow* ptrWindow, const sf::Font* ptrFont, int
playerID, sf::Color pawnColor, bool isBot = false);
    Konstruktor obiektu klasy Player.
    Przyjmowanymi argumentami wypełnia pola składowe ptrWindow, ptrFont,
    playerID, pawnColor, i isBot.

public:      friend bool operator==(Player Lvalue, Player Rvalue);
public:      friend bool operator!=(Player Lvalue, Player Rvalue);
    Redefinicje operatorów == i !=.
    Sprawdzają, czy obiekty klasy Player Lvalue I Rvalue to tak naprawdę te same
    obiekty.

```

Klasa Board:

Pola:

```

private:      sf::RenderWindow* ptrWindow;
    Przechowuje wskaźnik do okna.

private:      sf::Font* ptrFont;
    Przechowuje wskaźnik do czcionki.

private:      sf::Texture rosetteImg;
    Przechowuje teksturę rozetki, żeby nie musieć jej wczytywać z pliku przy każdym
    wyrysowaniu jej.

private:      Pawn* tileContents[8][3];
    Przechowuje wskaźniki na pionki stojące na polu o koordynatach określonych
    indeksami tablicy.
    Wartość nullptr odpowiada pustemu polu.

private:      bool isRosette[8][3];
    Przechowuje informację o tym, czy pole o koordynatach określonych indeksami
    tablicy jest rozetką.

public:      Player player1;
public:      Player player2;
    Obiekty klasy Player odpowiadające poszczególnym graczom.

public:      bool applyBotDelay;
    Określa czy computer ma odczekać chwilę przed wykonaniem ruchu, aby użytkownik
    mógł nadążyć za tym, co robi.

public:      sf::Time botDelay;
    Określa długość czasu, przez jaki computer powinien czekać przed wykonaniem
    ruchu.
    Inny dla trybów gracz vs komputer i komputer vs komputer.

```

Metody:

```

private:      void refreshPointers();
    Odświeża tablicę wskaźników na pionki Pawn* tileContents po wykonaniu ruchu.
    Naprawia to błędy związane ze zgubieniem pionków po realokacji wektora
    activePlayers w obiektach klasy Player.

private:      void drawBoard()const;
    Rysuje planszę na środku ekranu.

```

```

private:    void drawPawns(const Player& player)const;
           Rysuje pionki znajdujące się na planszy.

private:    bool checkIfValidMove(Pawn& curPawn, int moveBy,
sf::Vector2i targetTile)const;
           Sprawdza, czy wykonanie ruchu danym pionkiem na daną pozycję na ścieżce i
           koordynatach na planszy jest ruchem dozwolonym.
           Zwraca wartość logiczną, mówiącą o tym, czy ruch jest dozwolony.
           Pawn& curPawn – odnosi się do pionka, którego ruch ma zostać oceniony.
           int moveBy – określa o ile pól do przodu pionek próbuje się poruszyć.
           sf::Vector2<int> targetTile – określa koordynaty pola na planszy, na które
           pionek próbuje wejść.

private:    bool checkIfValidMove(const int& curPlayer, int moveBy,
sf::Vector2i targetTile)const;
           Sprawdza, czy postawienie nowego pionka na wyznaczonej pozycji na ścieżce i
           koordynatach na planszy, jest dozwolonym ruchem.
           Zwraca wartość logiczną, mówiącą o tym, czy ruch jest dozwolony.
           int& curPlayer – określa który gracz próbuje postawić na planszy nowy pionek.
           int moveBy – określa pozycję na ścieżce na której nowy pionek ma próbować się
           pojawić.
           sf::Vector2<int> targetTile – określa koordynaty pola na planszy, na którym nowy
           pionek ma próbować się pojawić.

private:    void movePawn(sf::Vector2i fromTile, sf::Vector2i toTile,
int newPositionOnPath);
           Przemieszcza istniejący pionek z pola określonego koordynatami przechowanymi w
           argumencie fromTile, do pola określonymi koordynatami toTile.
           int newPositionOnPath – określa nową pozycję na ścieżce – nową wartość pola
           tileOnPath którą pionek ma przyjąć.

public:     void drawGame(Player* whoseTurn, int moveBy)const;
           Wywołuje odpowiednie metody rysujące.

public:     void processPlayerActions(Player* whoseTurn, float mX, float mY,
bool released, int moveBy, bool* landedOnRosette, bool* hasMoveBeenMade);
           Odnajduje obiekt na którym znajduje się kursor myszki użytkownika, i wykonuje
           odpowiednie akcje, zależnie od tego czy użytkownik kliknął na obiekt, czy tylko
           na niego najechał.
           Player* whoseTurn – określa którego gracza jest w danej chwili ruch.
           Float Mx, float My – koordynaty kursora względem okna.
           bool released – czy użytkownik puścił myszkę. W ten sposób wykrywane są
           pojedyncze kliknięcia.
           int moveBy – określa wyrzuconą liczbę oczek.
           Funkcja zwraca przez wskaźniki landedOnRosette i hasMoveBeen made informację o
           dodatkowej turze za wylądowanie na rozetce, oraz o końcu tury gracza.

public:     void processBotActions(Player* whoseTurn, int moveBy,
bool* landedOnRosette, bool* hasMoveBeenMade);
           Decyduje jaki ruch w danej turze wykona komputer, oraz go wykonuje.
           Argumenty i wartości zwracane przez wskaźniki są takie same jak w metodzie
           processPlayerActions, ale pomniejszone o funkcje opisujące położenie i stan
           myszki.

public:     Board(sf::RenderWindow* window, sf::Font* ptrFont);
           Konstruktor obiektu klasy Board.
           Przyjmuje wskaźnik na okno, by móc rysować w nim obiekty, oraz podać je do
           obiektów innych klas znajdujących się w polach składowych.
           Z podobnych powodów, przyjmuje też wskaźnik na czcionkę (obiekt sf::Font).

```

W zakresie **main**:

```
int rollDice();  
    Symuluje rzut 4 kostkami (analogiczny do rzutu 4 monetami).  
    Zwraca liczbę wyrzuconych oczek.
```

Zmienne globalne:

```
#define WINDOW_WIDTH 800.f  
    Określa szerokość okna.  
  
#define WINDOW_HEIGHT 600.f  
    Określa wysokość okna.  
  
#define BOARD_POSITION_X (WINDOW_WIDTH/2 - 4*BOARD_TILE_SIZE)  
    Określa koordynat X lewego górnego rogu planszy.  
  
#define BOARD_POSITION_Y (WINDOW_HEIGHT/2 - 1.5*BOARD_TILE_SIZE)  
    Określa koordynat Y lewego górnego rogu planszy.  
  
#define BOARD_TILE_SIZE 80.f  
#define BOARD_TILE_OUTLINE_THICKNESS (BOARD_TILE_SIZE / 10)  
#define PAWN_SIZE (BOARD_TILE_SIZE - 4*BOARD_TILE_OUTLINE_THICKNESS)  
#define HUD_MARGIN 20.f  
    Określają wymiary pola na planszy, grubość oblamówki pól, średnicę pionka,  
    margines od granic okna, rozmiary tekstu...  
    Określają wymiary i położenie wszystkiego, co wyrysowywane jest na ekran.  
  
#define LARGER true  
#define TRANSPARENT true  
#define FORBIDDEN true  
    Używane do czytelniejszego określania parametrów dla metod rysujących pionki.  
  
#define FINISH_LINE 14  
    Określa pole na ścieżce, którego osiągnięcie wiąże się z zejściem pionka z  
    planszy.  
  
#define BOT_DELAY_PLAYER_VS_BOT 400  
#define BOT_DELAY_BOT_VS_BOT 200  
    Czas oczekiwania komputera przed wykonaniem ruchu, wyrażony w milisekundach.  
    Zależy od trybu.
```