Politechnika Śląska w Gliwicach
Wydział Automatyki, Elektroniki i Informatyki

Laboratorium Programowania Komputerów

temat projektu

4. Statki

autor	Michał Jankowski
prowadzący	dr inż. Jolanta Kawulok
rok akademicki	2018/2019
kierunek	Informatyka
rodzaj studiów	SSI
semestr	3
termin laboratorium / ćwiczeń	czwartek, 8:30 – 10:00
grupa	1
sekcja	1
termin oddania sprawozdania	2019-01-24
data oddania sprawozdania	2019-01-24

1. Temat

Celem projektu jest napisanie programu służącego do grania w grę „Statki” dla 2 użytkowników. Gra toczy się na planszy kwadratowej o rozmiarach podanych przez użytkownika. Również, jako parametry gry podawane są liczby statków o poszczególnych rozmiarach (dane te są wczytywane z osobnego pliku).

Statki ustawiane są w pozycji pionowej lub poziomej, tak aby nie stykały się one ze sobą w żaden sposób (ani bokami, ani rogami).

Program powinien również mieć możliwość zapisu stanu gry w sposób binarny, a następnie po wczytaniu pliku możliwość kontynuacji gry.

Program powinien być uruchamiany z linii poleceń, gdzie są podawane parametry programu. Należy też zastosować parametry domyślne (np. nazwa pliku z zapisem stanu gry).

2. Analiza tematu

2.1 Doprecyzowanie tematu

Zgodnie z wymogiem projektu założono możliwość gry zarówno z użytkownikiem, ale także w celu urozmaicenia gry dodano opcje gry z komputerem, jako rodzaj sztucznej inteligencji (AI).

Projekt zakłada, iż gracz może wybrać dowolny rozmiar planszy jednak nie powinien on przekraczać liczby słów w alfabecie łacińskim.

Interfejs interakcji z użytkownikiem będzie informował o większości dokonanych akcji przez gracza z pominięciem oczywistych błędów, które wpłynęłyby na czytelność gry. Jest to projekt konsolowy, dlatego starano się o zadbanie odpowiedniej przejrzystości dla gracza.

2.3 Wybór klas oraz bibliotek

Zdecydowano o następującym wyborze klas:

- Board;
- Field;
- CheckTile;
- Player;
- ComputerPlayer;
- HumanPlayer;
- Ship;
- Display_Interface;
- Game;
- Parameters;

Klasy **Board**, **Field**, **CheckTile** są odpowiedzialne za przetwarzanie danych do tablicy stanów wyświetlającej aktualny stan gry oraz trafienia lub ich brak w statki przeciwnika. Klasa **CheckTile** pozwala na pobranie informacji o współrzędnych pola, natomiast Field ustawia stan logiczny pola (trafiony lub nietrafiony). **Board** umieszcza informacje o polach do odpowiedniej tablicy, która jest wykorzystywana w grze do reprezentacji planszy. Taki sposób podziału klas pozwala na czytelny podział danych wprowadzonych do tablicy.

Klasy **Player**, **ComputerPlayer**, **HumanPlayer** są odpowiedzialne za ustawianie imienia, typu gracza oraz ruchu gracza. Klasa Player zawiera czyste metody wirtualne, które są wykorzystywane w klasie **ComputerPlayer** oraz **HumanPlayer**. Wykorzystano następujący podział, aby móc zastosować polimorfizm oraz uwidocznić zależność klas **ComputerPlayer** oraz **HumanPlayer** od **Player**.

Klasa **Ship** zawiera wszystkie informacje o statkach, aby móc je wprowadzić do tablicy gry**. Ship** jest niezbędną klasą w celu przechowywania kluczowych danych o położeniu statków.

Klasa **Display_Interface** zawiera informacje o poziomych oraz pionowych indeksach wyświetlanej tablicy gry. Jest również

odpowiedzialna za podział planszy gry dla gracza i przeciwnika. **Display_Interface** umożliwia sprawny podział i czytelny podział plansz gry dla obu użytkowników.

Game jest najważniejszą klasą w całym programie. Jest tutaj zaimplementowana najważniejsza logika gry oraz ze względu na bezpośredni dostęp danych zapis i odczyt danych z pliku.

W klasie **Parameters** przechowywane są i odpowiednio przetwarzane dane z wiersza poleceń, w celu odpowiedniego podziału i umieszczenie ich w odpowiednich zmiennych.

Zastosowano następujące biblioteki:

- *iostream*-> w celu skorzystania z strumienia wejścia/wyjścia;
- *vector* -> niezbędne w celu skorzystania z wektorów oraz dostępnych ich funkcjonalności;
- *string*-> w celu wykorzystania łańcuchu znaków;
- *istream* oraz *fstream* -> w celu operacji na plikach;
- *iomanip* -> w celu wykorzystania metody setw();
- *iterator* -> w celu wykorzystania iteratorów;
- *algorithm* -> w celu określenia elementu największego i najmniejszego (metoda max() oraz min());
- *windows*.h -> w celu określenia aktualnego czasu.

2.2 Algorytmy, struktury danych, ograniczenia specyfikacji

Struktury danych

Wykorzystano struktury dynamiczne:

- Tablice (pamięć alokowana własnoręcznie);
- Wektory.

Wykorzystano struktury statyczne:

- Tablice.

Powodem ich zastosowania była możliwość łatwiej reprezentacji danych oraz ich późniejszego przetwarzania. Dodatkowo znaczącą zaletą stosowania dynamicznych struktur jest nieograniczanie z góry danych wykorzystywanych w projekcie.

Algorytm:

W danym projekcie zastosowane zostały głównie proste algorytmy pomijając algorytm kolizyjny wstawiania statków.

*Wykorzystano następujące algorytm*y*:*

- Przechodzenia po tablicy iterując po danych elementach w szukanej strukturze danych;
- Sprawdzanie zgodności wprowadzonych danych za pomocą operatorów logicznych;
- Inicjowania pustych pól danych struktur;
- Prostego interfejsu opierającego się na wpisywanych przez użytkownika danych;
- Zapisywania do pliku binarnego danych;
- Odczytywania z pliku binarnego danych;
- Kolizji statków podczas ich wstawiania

Zastosowanie ich okazało się kluczowe do wykonania danego projektu. Wypunktowane algorytmy pozwoliły na umiejętny podział wykonywanych przez dane metody poleceń.

Ograniczenia specyfikacji:

Projekt pomimo dużej złożoności nie wyczerpuje w pełni opisywanego zagadnienia. W wyniku wgłębiania się w dany projekt zauważono, iż można było poprawić następujące jego elementy:

- Udoskonalić sposób gry z komputerem, aby dodać więcej poziomów trudności;
- Dodać tablice wyników;
- Dodać możliwość losowego wstawiania danych;
- Zmienić kolor wyświetlania danych, to znaczy nie ograniczać się do koloru białego i czarnego;

- Zmienić metodę sprawdzającą parametry podane z linii poleceń niesprawdzającą tak wielu warunków;
- Przekształcić metodę wyświetlającą indeksy poziome i pionowe w przypadku gdy gracz przekroczy liczbę znaków w alfabecie łacińskim.

3. Specyfikacja zewnętrzna

3.1 Format danych wejściowych

Program ,,Statki" jest programem przyjmującym zewnętrzne parametry jakimi są:

- Nazwa pliku binarnego ***.dat**, w celu wczytania wcześniej zapisanej gry;
- Rozmiar kwadratowej planszy w postaci liczby np. 7;
- Plik tekstowy *.txt, który zawiera informacje o liczbie i rozmiarze statków;
- Nazwa pliku binarnego ***.dat**, który ma służyć do zapisu gry w trakcie rozgrywki;

Wszystkie parametry są opcjonalne. Można nie wpisywać żadnego, jednak wtedy zostaną wykorzystane parametry domyślne.

Użytkownik uruchamiając program z wiersza linii poleceń ścieżkę programu następnie, jeżeli nie chcemy korzystać z domyślnych parametrów wybieramy dla nas odpowiadający przełącznik.
Po wpisaniu następujących przełączników podajemy dla:

- **„-in"** nazwę pliku do kontynuacji gry.
- **„-innum"** nazwę pliku tekstowego zawierającego dane do wprowadzenia o statkach**;**
- **„-n"** rozmiar kwadratowej planszy**;**
- **„-out"** nazwa pliku wyjściowego.

Jeżeli podamy plik do kontynuacji gry, pozostałe parametry zostaną pominięte.

Przykładowe prawidłowe formaty wprowadzonych danych:

- **„Project_Ships.exe -in plik.dat”**
- **„Project_Ships.exe ”**
- **„Project_Ships.exe -innum liczba_statkow.txt –n 7 ”**

Jeżeli, któryś z parametrów zostanie podany błędnie lub podany plik nie będzie istniał program poinformuje użytkownika komunikatem: „**Some parameters may be wrong!”** lub „**Couldn’t open <nazwa pliku> for reading!**”.

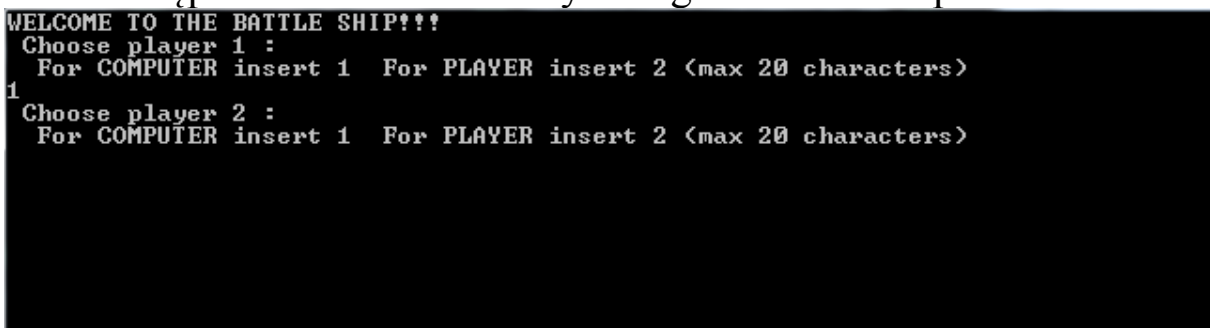
3.2 Obsługa programu

Program korzysta z prostego interfejsu wyświetlającego informację w konsoli w zależności od wpisanych danych. Aby dane zostały poprawnie wprowadzone przed każdą konkretną modyfikacją danych użytkownik zostanie poinformowany jak należy je poprawnie podać.

Dla lepszego zrozumienia wykorzystywanych oznaczeń pól w planszy wykorzystano legendę:

- „X” -> Trafiony (Hit);
- „*”-> Pudło (Miss);
- „.”-> Woda (Water);
- „#”->Statek (Ship).

1) Najpierw użytkownik zostanie powitany komunikatem, a następnie ma możliwość wyboru gracza lub komputera



Powyżej znajduje się wybrany przez użytkownika 1 gracz, jako **komputer** z oczekiwaniem na wybór 2 gracza.

```
WELCOME TO THE BATTLE SHIP!!!
 Choose player 1 :
  For COMPUTER insert 1  For PLAYER insert 2 (max 20 characters)
1
 Choose player 2 :
  For COMPUTER insert 1  For PLAYER insert 2 (max 20 characters)
dane
Incorrect value
5
Incorrect integer
```

Jako przypomnienie niepoprawne dane zostały zanotowane

2) Po wyborze gracza innego niż komputer użytkownik zostanie poproszony o podanie nazwy użytkownika

```
WELCOME TO THE BATTLE SHIP!!!
 Choose player 1 :
  For COMPUTER insert 1  For PLAYER insert 2 (max 20 characters)
1
 Choose player 2 :
  For COMPUTER insert 1  For PLAYER insert 2 (max 20 characters)
dane
Incorrect value
5
Incorrect integer
2
Welcome To Battle Ship Game! What is your name  (max 20 chars)?: Michal
```

3) Następnie gracz zostanie poproszony o podanie położenia swoich statków

```
Ship placed successfully
    ABCDEFGHIJ      ABCDEFGHIJ
  0 ..........    0 ..........
  1 #.........    1 ..........
  2 ..........    2 ..........
  3 ..........    3 ..........
  4 ..........    4 ..........
  5 ..........    5 ..........
  6 ..........    6 ..........
  7 ..........    7 ..........
  8 ..........    8 ..........
  9 ..........    9 ..........
Player Michal places ships
Ship 1
Place 2 ships of length 4
Enter coordinates: a4
Enter coordinates: a7
```

Powyżej od góry znajduje się informacja o poprawnie wstawionym statku na pole A1. Następnie poniżej znajduje się plansza do gry.

```
Player Michal places ships
Ship 1
Place 2 ships of length 4
Enter coordinates: a4
Enter coordinates: a7
```

Powyżej od góry widoczna jest informacja, jaki gracz wstawia statki, następnie informacja, który z kolei jest to statek oraz linijkę niżej mamy informacje o ilości ogólnej do wstawienia statków oraz ich długości.

Następnie poniżej wprowadzone współrzędne przez gracza w dwóch seriach najpierw początek, a następnie koniec. **Należy pamiętać, iż współrzędne zawsze są wprowadzone w dwóch seriach, a nie w jednej linii!** Dlatego, aby wstawić statek 4 elementowy podano współrzędne A4 i A7. A-A= 0 w wartościach poziomych, natomiast |4-7|+1 = 4. Dane się zgadzają.

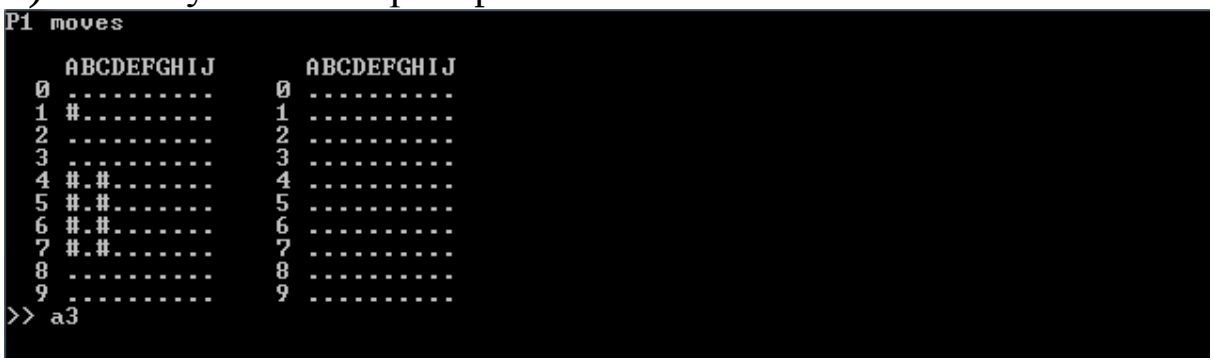
4) Kolejnym krokiem jest podanie następnego statku przy okazji sprawdzona zostanie kolizja

```
Ship placed successfully
    ABCDEFGHIJ      ABCDEFGHIJ
  0 ..........    0 ..........
  1 #.........    1 ..........
  2 ..........    2 ..........
  3 ..........    3 ..........
  4 #.........    4 ..........
  5 #.........    5 ..........
  6 #.........    6 ..........
  7 #.........    7 ..........
  8 ..........    8 ..........
  9 ..........    9 ..........
Player Michal places ships
Ship 2
Place 2 ships of length 4
Enter coordinates: B4
Enter coordinates: B7
Collision detected at: B4, B5, B6, B7
Enter coordinates: C4
Enter coordinates: C7
```

Powyżej widoczna jest podkreślona informacja o wystąpieniu kolizji na wszystkich odpowiednich polach, które miałby zostać zajęte przez inny statek. W takim razie wybrano pola C4 do C7.

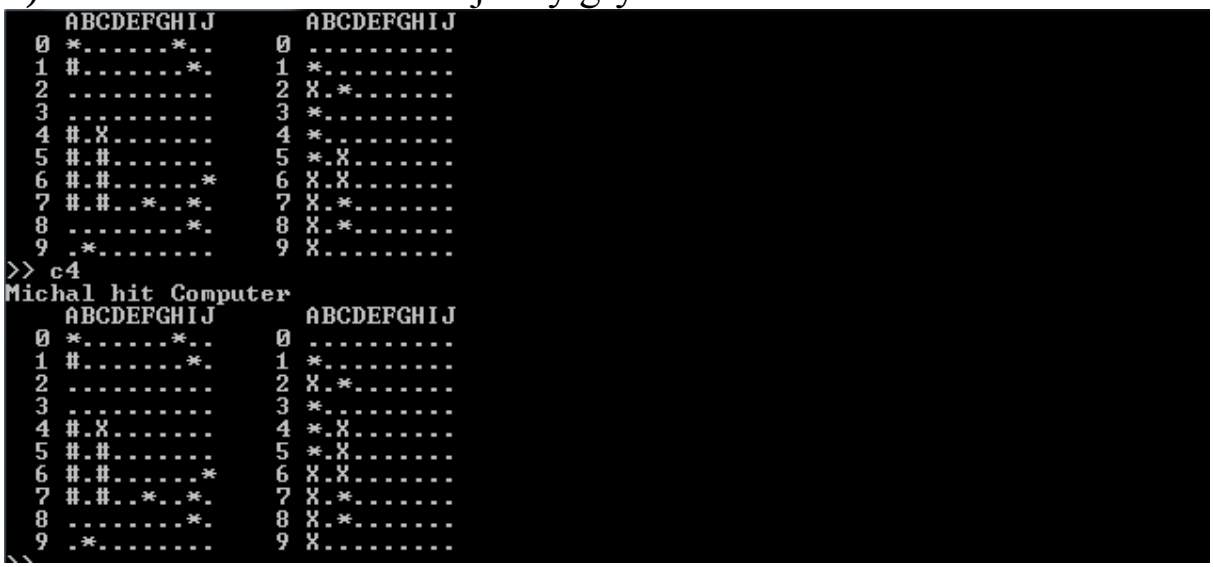
5) Komunikat informujący o zakończeniu wstawiania statków przez gracza 1 oraz początek wstawiania
dla gracza 2

6) Możemy atakować pola przeciwnika



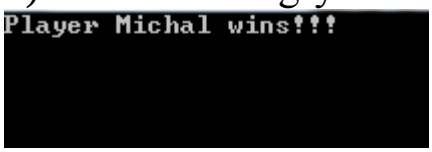
Po lewej znajduje się nasza plansza z statkami po prawej plansza gracza bez widocznych statków.

7) Przedstawienie końcowej fazy gry



Poniżej widoczne są trafione pola gracza przez komputer oraz trafione pola komputera przez gracza.

8) Zakończenie gry



Po trafieniu wszystkich statków użytkownik wygrywa.

Oczywiście w trakcie gry możliwe jest zapisywanie.
Dostępne są 3 opcje zapisu:

- „*save*” -> zapis z nazwą podaną przez użytkownika;
- „*defsave*” -> zapis domyślny wykorzystujący aktualną datę;
- „*exit*” -> zapis domyślny zakończony wyjściem z gry.

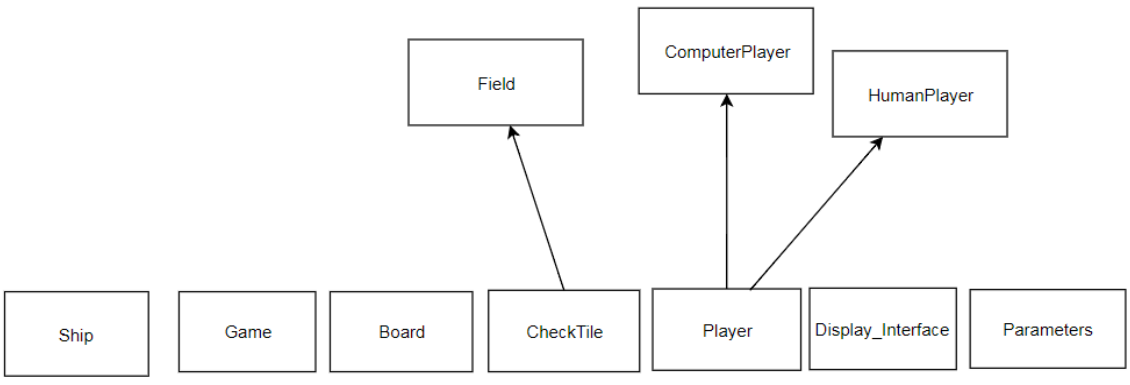
3.3 Dodatkowe informacje

Dane wejściowe:
Należy pamiętać, iż jako ograniczenie interfejsu maksymalny rozmiar tablicy nie powinien przekraczać 26, ponieważ 27 nie jest już literą.

4. Komunikacja wewnętrzna

4.1 Zastosowane klasy

Hierarchia wykorzystanych klas znajduje się poniżej w postaci diagramu.



4.2 Metody

Poniżej znajduje się opis istotnych metod oraz konstruktorów w programie.

A) Dla klasy Check_Tile

Klasa CheckTile umożliwia nam pobranie informacji o wierszu oraz kolumnie danego pola w postaci getterów, dzięki czemu mamy do nich ułatwiony dostęp.

Istotne pola:

- `char _row, _column;`

`CheckTile(char column, char row) : _column(column), _row(row) { }`
Konstruktor argumentowy tworzący pusty wiersz oraz kolumnę (ustawiane, jako 0).

`char GetRow() const;`
Metoda pobierająca informacje o wierszu, tzw.”getter”

`char GetCol() const;`
Metoda pobierająca informacje o kolumnie, tzw.”getter”.

B) Dla klasy Field

Klasa Field dziedziczy po CheckTile w celu pobrania współrzędnych danego pola oraz ustawia odpowiedni stan logiczny w danym polu. Dzięki takiej implementacji klasy można przedstawić reprezentacje danej informacji podczas gry.

Istotne pola:

- `bool _ship;`
- `bool _shot.`

`void SetShot(bool value = true);`
Metoda ustawiająca pole _shot (domyślnie true), tzw.”setter”.

`void SetShip(bool value = true);`
Metoda ustawiająca pole _ship (domyślnie true), tzw „setter”.

```
bool GetShip()const;
```
Metoda pobierająca zawartość pola _ship.

```
bool GetShot()const;
```
Metoda pobierająca zawartość pola _shot.

```
bool IsSunk() const;
```
Metoda zwracająca wartość dwóch stanów równocześnie _ship i _shot.

C) Dla Klasy Board

Klasa Board dziedziczy po Field oraz ustawia rozmiar tablicy gry. Odpowiada również za dodawania statku do tablicy gry w odpowiedniej reprezentacji. Dzięki takiemu podziałowi klasy można w łatwy sposób zauważyć, co dana klasa ma robić.
Istotne pola:

- `std::vector<Ship> _ships;`
- `std::vector<std::vector<Field>> _shipGrid;`

```
Field& getField(int r, int c);
```
Metoda zwracająca z vectora obiektów Field informacje o konkretnej wartości wiersza i kolumny. W reprezentacji tablicy dwuwymiarowej (_shipgrid[r][c])

```
int getSize()const;
```
Metoda zwraca rozmiar (_size) planszy gry.

```
bool validCoord(int row, int col) const;
```
Metoda sprawdzająca czy podane współrzędne są prawidłoe, tzn. wartość kolumny lub wiersz większa od 0 i jednocześnie nie powinna przekraczać rozmiaru planszy (_size).

```
void addShip(Ship& s);
```
Metoda wykorzystująca pętlę foreach do ustawienia stanów pola w tablicy _shipGrid , a następnie za pomocą metody push back() wpisanie informacji o istnieniu danego statku do vectora obiektów klasy Ship _ships tej informacji.

`void operator+=(Ship& s);`
Przeciążony operator +=, który inicjuje metodę `addShip(Ship& s);`

`inline ShipGridRef operator[](int row)`
Przeciążony operator[] zwracający obiekt będący pośrednikiem. Wykorzystywany w celach uniemożliwienia modyfikacji vectora _shipGrid. Wartość zwracana: brak

`void markBoard(Ship&);`
Metoda wykorzystująca pętlę foreach do oznaczenia odpowiednim stanem logicznym wartości statku w vectorze _shipGrid. Wykorzystano do tego obiekt klasy Ship.

W klasie Board występuje również klasa pomocnicza.

```
//klasa pomocnicza zapamietujaca odwolania do vectora fieldow
//daje nam mozliwosc odwolania sie do elementu vectora ale nie daje mozliwosci modyfikacji vetora
class ShipGridRef {
    std::vector<Field>& ref;
public:
    //posrednik
    inline ShipGridRef(std::vector<Field>& ref) :
        ref(ref)
    {}

    //operator pobierajacy element z vectora na podanym indeksie
    inline Field& operator[](int col) {
        return ref[col];
    }
};
```

D) Klasa Player

Klasa Player wykorzystuje polimorfizm w związku z ustawianiem wszystkich danych dla gracza. W tej klasie występuje również deklaracja przyjaźni klasy HumanPlayer : `friend class HumanPlayer`.

Istotne pola:

- `Game* game;`
- `std::string _name;`

Obiekt Game wykorzystywany jest do utworzeni obiektu klasy Player.

`virtual bool Move(int& col, int& row, Board& board) = 0;`
Czysto wirtualna metoda wykorzystywana później w innych klasach po niej dziedziczących do ustawienia ruchu gracza.

`virtual void SetUpName() = 0;`
Czysto wirtualna metoda wykorzystywana później w innych klasach po niej dziedziczących do ustawienia imienia gracza.

`void SetName(std::string const& s);`
Metoda ustawiająca imię. Wykorzystuje pole _name do jego ustawienia.

`inline std::string GetName() const { return _name; }`
Metoda zwracająca zawartość pola _name.

`Player(Game* game);`
Konstruktor jednoargumentowy wykorzystujący obiekt game do jego inicjalizacji.

`virtual int Type() = 0;`
Czysta metoda wirtualna, wykorzystywana do określenia typu gracza. 1 dla ComputerPlayera, a 2 dla HumanPlayera.

E) Dla klasy HumanPlayer

Klasa HumanPlayer dziedziczy po Player oraz korzysta z metod wirtualnych dotyczących m.in. wykonywania ruchu przez użytkownika. Występuje tutaj deklaracja przyjaźni klasy Player: `friend class Player`.

Istotne pola:

- `static std::string StaticValidateSaveFile;`

```
virtual bool Move(int& col, int& row, Board& board) override;
```
Metoda wirtualna ustawiająca na podstawie podanych współrzędnych możliwość ataku na dane pole planszy gry board.

```
virtual void SetUpName() override;
```
Metoda wirtualna ustawiająca imię gracza.

```
virtual int Type() override { return 2; }
```
Metoda zwraca informacje o typie gracza.

```
static int ColIndex(char colLetter);
```
Metoda sprawdzająca numer kolumny. Uwzględnia czy litera została podana z małej czy wielkiej litery.

```
bool validateCommand(std::string const& line, Game *game);
```
Metoda wykorzystywana do komunikacji z użytkownikiem, w celu możliwość zapisania stanu gry.

```
static bool GetCoord(int& row, int& col, Board& board, std::string const& line);
```
Metoda sprawdzająca poprawność wprowadzonych współrzędnych przez gracza w trakcie gry.

```
static bool GetCoord(int& row, int& col, Board& board);
```
Metoda sprawdzająca poprawność wprowadzonych współrzędnych przez gracza w trakcie ustawiania statków na planszy.

F) Dla klasy ComputerPlayer

Klasa ComputerPlayer dziedziczy po Player oraz korzysta z metod wirtualnych dotyczących m.in. wykonywania ruchu przez komputer. Komputer ma ograniczone możliwości w stosunku do klasy HumanPlayer.

```
virtual void SetUpName() override;
```
Metoda wirtualna ustawiająca imię komputera.

```
virtual bool Move(int& col, int& row, Board& board) override;
```
Metoda wirtualna wykorzystywana do wykonania ruchu przez komputer.

```
virtual int Type() override { return 1; }
```
Metoda wirtualna zwracająca typ gracza. Dla komputera to 1.

G) Dla klasy Display_Interface

Zadaniem tej klasy jest wyświetlanie informacji o grze w konsoli. Ustawia m.in. indeksy poziome i pionowe planszy.

Istotne pola:

- `std::vector<std::string> _panelLeft;`
- `std::vector<std::string> _panelRight;`

Wykorzystano również iteratory dla łatwiejszego przemieszczania się po tablicy.

```
using display_iterator = std::vector<std::vector<std::string> >::iterator;
using panel_iterator = std::vector<std::string>::iterator;
```

```
friend std::ostream& operator<<(std::ostream& os, DisplayInterface const& di);
```
Przeciążony operator wykorzystywany do oddzielenia jednej tablicy od drugiej z uwzględnieniem należytych proporcji przy wyświetlaniu gry.

```
void placeBoardLeft(Board& board, bool hidden);
```
Ustawia zawartość tablicy dla lewej strony z ukryciem statków, aby przeciwnik ich nie widział. Metoda podobna do podanej to:
`placeBoardRight(Board& board, bool hidden);`

```
void initPanel(std::vector<std::string>& panel);
```
Metoda wykorzystywana do inicjalizacji tablicy pustymi wartościami o rozmiarze _size.

```
void placeBoard(std::vector<std::string>& panel, Board& board, bool hidden);
```
Metoda ustawia tablice odpowiednimi stanami logicznymi.

Wykorzystano również w tej klasie namespaca

```
//enum stanow wewnatrz namespaca ShipState
//określa wizualna reprezentacje pola zgodnie z okresolnym stanem w grze
namespace ShipState {
    enum _ShipState : char {
        Hit = 'X', Miss = '*', Water = '.', Ship = '#'
    };
}
```

H) Dla klasy Parameters

Podana klasa ustawia wartości odpowiednich pól wykorzystywanych później w programie zgodnie z podanymi przez użytkownika wartościami z linii wiersza poleceń.

Istotne pola:

- `std::string  BoardSizeParam;`
- `std::string ShipSizeFileNameParam;`
- `std::string SaveFileNameParam;`
- `std::vector<std::string> arguments;`
- `std::string _LoadFileName;`
- `std::vector<ShipLength> shipLengths;`

```
void CheckParam(std::vector<std::string> arguments);
```
Metoda sprawdzająca możliwości ustawienia parametrów podanych przez użytkownika. Wpisuje ich zawartość do odpowiednich pól klasy Parameters.

`void SetupDefaultShipLengths();`
Metoda ustawia domyślny dane dotyczące ilości i rozmiaru statku w przypadku gdy nie zostaną podane takowe dane przez użytkownika.

`void ShipSizeFile();`
Metoda otwierająca plik przechowujący informacje o statkach, w innym przypadku wykorzystuje domyślne wartości.

`void GetFileSize();`
Metoda wykorzystywana do otwarcia wcześniej zapisanej gry z pliku. W przypadku nie znalezienia kontynuacji gry wykorzystujemy domyślne parametry.

`void readLine(char* buffer);`
Funkcja odczytująca nazwę linii autobusowej, korzystająca ze scanf_s(). Wartość zwracana: brak

`friend std::istream& operator>>(std::istream& is, ShipLength& coordinates);`
Przeciążony operator strumieniowy niezbędny do pobierania danych z pliku tekstowego dotyczącego rozmiaru i ilości staków.

Wykorzystano również w tej klasie pomocniczą strukturę

```
//struktura pomocnicza do przekazywania informacji o dlugosci i liczbie statkow
//wykorzystywana do ustawiania wspolrzednych
struct ShipLength {
    unsigned length;
    unsigned amount;

    friend std::istream& operator>>(std::istream& is, ShipLength& coordinates);

};
```

I) Dla klasy Ship

Klasa wykorzystywana do przechowywania wszystkich danych o statkach.

Zawiera ona takie informacje jak m.in. współrzędne statku oraz liczbę jego elementów. Przechowywane są te dane w wektorze _sections.

Istotne pola:

- `std::vector<Field*> _sections;`
- `int _size, _col1, _col2, _row1, _row2;`

Wykorzystana również iterator po vectorze Field* :

```
using iterator_ship = std::vector < Field *>::iterator;
```

`Ship( Board* board, int column1, int row1, int column2, int row2, int size)`
Konstruktor wieloargumentowy inicjalizujący podanymi danymi statek oraz przypisuje również te wartości do odpowiednich pól klay Board.

`bool IsSunk();`
Metoda sprawdzająca czy dany statek ma wszystkie pola ustawione na Trafione (Hit). Iteracja następuje w pętli foreach. Zwraca true jeśli statek zatopiony.

`void saveToFile(FILE* file) const;`
Metoda wykonująca zapis binarny do pliku z zachowaniem odpowiedniej kolejności, aby w łatwy sposób można było te dane odczytać.

`void readFromFile(FILE* file);`
Metoda wykorzystywana do odczytania danych o statkach z pliku w taki samej kolejności jak zostały zapisane.

I) Dla klasy Game

Jest to jedna z najważniejszych klas w całym programie. Znajduje się tutaj większość logiki gry. Ze względu na łatwość dostępu do danych tutaj również zachodzi zapis i odczyt z pliku. Istotne pola:

- `Player* p1, *p2;`
- `Player* _currentPlayer;`
- `bool _finished;`
- `Board p1Board, p2Board;`

Obiekt p1 oraz p2 reprezentują jeszcze nieustalonych graczy. Te 2 obiekty są aalokowane dynamicznie.
Obiekt _currentPlayer określa, który aktualnie gracz wykonuje ruch.

`void Run();`
Metoda rozpoczynająca grę. Ustala czy użytkownik wczytał grę, jeżeli nie to rozpoczynamy nową grę.

`void InitializePlayer();`
Metoda wykorzystywana do inicjalizacji gracza. Tutaj użytkownik wybiera, z kim chce grać.

`bool InitializeOpponent();`
Metoda wykorzystywana w `void InitializePlayer()` w celu sprawdzenia poprawności danych.

`static bool** allocBoolBoard(size_t size);`
Metoda wykorzystywana do alokacji tablicy kolizyjnej, która sprawdza kolizyjność statków.

`void SaveGame();`
Metoda wykorzystywana do zapisu gry do pliku binarnego. Zwiera w sobie inne metody zapisujące poszczególne kluczowe dane.

```
void LoadGame();
```
Metoda wykorzystywana do odczytu gry z pliku binarnego. Zwiera w sobie inne metody odczytujące poszczególne kluczowe dane.

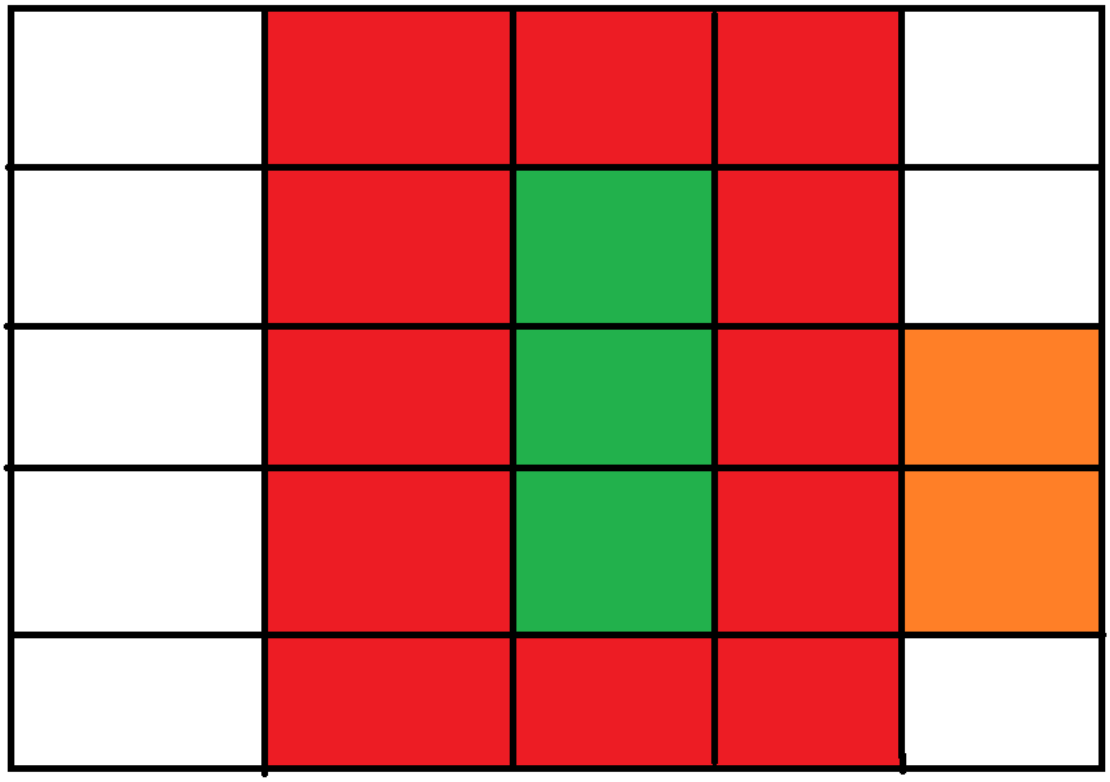
```
void UpdateP1();
```
Metoda aktualizująca ruchy wykonywane przez gracza. Informuje o ruchach gracza poprzez komunikaty. Metoda ta również ustawia, że gracz drugi będzie po nim wykonywał ruch.

```
void GameUpdate();
```
Aktualizuje stan gry z sprawdzeniem, kto będzie następny wykonywał ruch.

```
void SetupBoard(Player* player, Board& board);
```
Metoda kolizyjna. Wykorzystuje kilka algorytmów do sprawdzenia poprawności wyboru miejsca statku przez gracza niebędącym w obszarze kolizyjnym. Korzysta z innych metod m.in. `static bool** allocBoolBoard(size_t size)` w celu utworzenia nasze drugiej tablicy wykorzystywanej do kolizji statków.

Bardzo dokładny opis tej metody znajduje się w samym pliku Game.cpp, ze względów przejrzystości i czytelności nie umieściłem go tutaj. Zalecam jednak przejrzenie go.

Poniżej znajdue się tablica kolizyjna, gdzie:
a) Czerwony -> możliwa kolizja ze statkiem zielonym
b) Zielony-> wstawiony statek
c)Pomarańczowy->kolejny statek
Przedstawia ona możliwy scenariusz wtawienia statku i jak program interpretuje takie wstawienie.



```
void GameEnd();
```

Metoda ustawiająca wartość pola _finished na true. Powoduje to, że w następnej iteracji w logice gry następuje wyjście z programu.

```
bool GameOver() const;
```

Metoda zwracająca zawartość pola _finished. Sugeruje koniec gry, jeżeli pole te przyjmuje wartość true.

5.Testowanie

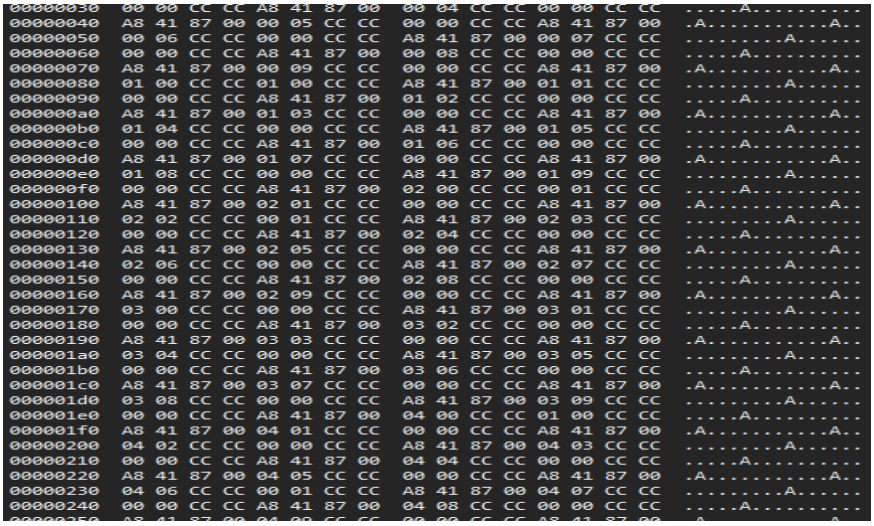
5.2 Dane testowe – uzasadnienie

Do testowania programu został wykorzystany plik liczba_statków.txt, który zawiera przygotowaną i działającą bazę danych statków. Testowane były różne położenia statków oraz ich rozmiar. Dla wszystkich przetestowanych danych nie napotkano problemów. Sprawdzano również różne metody zapisu, tzn. wykorzystywano możliwość własnoręcznego zapisu, ale także zapisu domyślnego. Dla każdego przypadku dane działały. Granie gracz na gracz działało poprawnie. Sprawdzano czy przerwy pomiędzy wstawianiem statku są odpowiednie, aby żaden gracz nie widział statków przeciwnika.

5.3 Napotkane problemy

Istotnym problemem okazał się zapis binarny, ponieważ zapisywano często błędnie dwa razy te same dane, które nadpisywały inne niezbędne dane.

Poniżej znajduje się przedstawienie danej sytuacji", która powodowała nam zły zapis danych.

to  Rozwiązano poprawieniem błędnie zapisanej pętli for.

Napotkano się również na problem nie zapisywania się statków po wyjściu z gry. Problemem okazał się brak takiej metody, która mogłaby to zrobić w klasie Game, dlatego zapis i odczyt statków odbywa się w klasie Ship.

Istotnym znaczeniowo dla wizualnej przejrzystości gry było stosowanie odpowiednich przerw pomiędzy wstawianiem statków przez graczy. W trakcie dodawania statku oraz wykonywania ruchu przez gracza dodaje się pusta nowa linia, która przesuwała tablicę o 1 wiersz w dół. Powodowało to „rozcinanie planszy”. Dodanie w tablicy kolizyjnej `std::cin.ignore(INT_MAX, '\n')` spowodowało naprawienie tego błędu.

6.Wnioski

Program „Statki” pomimo okazał się projektem ciekawym oraz uczącym aspektów tworzenia logiki gier. Jako, iż pierwszy raz tworzyłem grę zauważyłem, że dzięki wszelkim wizualnym implementacją można uzyskać ciekawe efekty przedstawienia różnych danych. W porównaniu do tworzenia baz danych łatwiej jest również zauważyć jakikolwiek błąd lub usterkę w programie. Najwięcej czasu zajęło mi zaimplementowanie logiki gry m.in. kolizji statków. Projekt podsumowuje, jako przyjemny i uczący algorytmicznego myślenia.