**Gry planszowe**

Dokumentacja techniczna

Projekt PROI

**Ewa Miazga**

Numer indeksu: **318694**,

**Bartłomiej Niewiarowski**

Numer indeksu: **318701**,

**Jakub Kieruczenko**

Numer indeksu: **318669**,

1. Cel projektu

Celem naszego projektu było stworzenie programu, który umożliwia użytkownikowi nie tylko rozwiązywanie Sudoku, Krzyżówek czy Kółko krzyżyk, ale również możliwość wglądu w swoje statystyki i porównanie swoich wyników z innymi graczami. Program wizualnie miał być intuicyjny dla użytkownika, dlatego zdecydowaliśmy się na interfejs graficzny zamiast konsolowego. Projekt ma być odpowiedzią na temat zadany w trakcie realizacji przedmiotu programowanie obiektowe, natomiast postanowiliśmy zaimplementować go tak, aby z łatwością można było go rozbudować poprzez dodanie kolejnych gier.

1. Opis projektu

Na szkielet projektu składają się trzy główne części, które współpracując ze sobą tworzą komplementarny program. Pierwsza z nich to logika gier, która została zaimplementowana w 3 osobnych klasach TicTacToe, Sudoku oraz Crossword. Są to klasy pochodne, dziedziczące po klasie Games, który tworzy interfejs publiczny tej części programu. Kolejną częścią jest obsługa użytkownika, pozwalająca utworzyć i wczytać profil oraz powiązane z nim statystyki. Projekt posiada również graficzny interfejs, wyświetlający gry oraz użytkowników.

1. Podział obowiązków

Staraliśmy się dokonać podziału obowiązków sprawiedliwie, ale biorąc pod uwagę mocne strony każdego uczestnika. Dzięki temu, mogliśmy wzajemnie się uzupełniać i również uczyć się od siebie. W ten sposób postanowiliśmy, że każdy z nas otrzyma następujące zadania, które zmieniły się podczas rozwoju projektu:

Ewa Miazga:

* Udział w kreowaniu założeń oraz w desk research’u
* Stworzenie logiki gier TicTacToe, Sudoku, Crossword
* Testowanie napisanego kodu
* Znalezienie niezależnych testerów manualnych gry
* Stworzenie dokumentacji projektu

Bartłomiej Niewiarowski:

* Udział w kreowaniu założeń oraz w desk research’u
* Organizacja pracy zespołu
* Znalezienie niezależnych testerów manualnych gry
* Modyfikacja gier w celu umożliwienia połączenia z resztą projektu
* Stworzenie kont użytkownika
* Stworzenie klas aplikacji łączących elementy projektu
* Scalenie elementów projektu

Jakub Kieruczenko:

* Udział w kreowaniu założeń oraz w desk research’u
* Organizacja pracy zespołu
* Napisanie testów jednostkowych
* Porządkowanie repozytorium oraz kodu
* Ujednolicenie interfejsu gier i klas dziedziczących
* Całkowita redakcja i naprawa dokumentacji
* Stworzenie pomocniczych klas do wyświetlania – Board, Tile
* Współpraca w utworzeniu klas aplikacji

1. Opis techniczny
   1. Logika

Klasa Games:

Jest to klasa abstrakcyjna, która definiuje interfejs publiczny części logicznej programu. Po niej dziedziczy klasa TicTacToe, Sudoku oraz Crossword.

Metody publiczne:

* Games() = default;

Domyślny konstruktor klasy.

* ~Games() = default;

Domyślny destruktor klasy.

* virtual void play() = 0;

Metoda czysto wirtualna. Służyła do uruchomienia gry.

* virtual void display() = 0;

Metoda czysto wirtualna. Służyła do wyświetlania obiektów w terminalu.

Aby umożliwić przekazywanie informacji o stanie gry, klasa została poszerzona o metody publiczne:

* virtual void gameOver() = 0;

Metoda czysto wirtualna. Sygnalizuje zakończenie gry.

* virtual void getValue(int column, int row) = 0;

Metoda czysto wirtualna. Umożliwia dostęp do wartości planszy przechowywanych przez poszczególne gry.

Klasa TicTacToe:

Klasa definiuje wszystkie metody umożliwiające granie w kółko krzyżyk. Rozgrywka może toczyć się zarówno między dwoma graczami, jak i między graczem, a botem. Bot może grać strategią agresywną, przeciętną oraz prostą. Klasa dziedziczy po klasie abstrakcyjnej Games.

Stałe oraz makra:

Za pomocą dyrektyw preprocesora w czasie kompilacji są tworzone stałe:

N 3 – wymiar tablicy

rows 3 – ilość wierszy

columns 3 – ilość kolumn

emptySquare '\_' – definicja pola pustego

oraz makra:

findWin(row1, col1, row2, col2, row3, col3) – określa czy w danym wierszu, kolumnie lub linii doszło do zwycięstwa

willWin(row1, col1, row2, col2, row3, col3, value) – określa czy w danym wierszu, kolumnie lub linii może dojść do zwycięstwa w kolejnym ruchu.

findChanceToWin(row1, col1, row2, col2, row3, col3, value) - określa czy w danym wierszu, kolumnie lub linii warto wstawić pionek, by w trzecim ruchu mieć szansę na wygraną.

findDoubleChanceToWin(row1, col1, row2, col2, row3, col3)

findEmptyLine(row1, col1, row2, col2, row3, col3) - określa czy dany wiersz, kolumna lub linia jest pusta.

Atrybuty prywatne:

* char board[rows][columns];
* std::string lvl;

Metody publiczne:

* TicTacToe() = default;

Domyślny konstruktor klasy.

* TicTacToe(std::string lvl);

Konstruktor.

* ~TicTacToe() = default;

Domyślny destruktor klasy.

* bool isEmpty(int row, int column);

Metoda sprawdzająca czy zdefiniowane za pomocą współrzędnych pole w planszy jest puste.

* void insert(int row, int column, char value);

Metoda wstawiająca pionek gracza w zdefiniowane za pomocą współrzędnych pole.

* char getWinner();

Metoda zwracająca zwycięzcę rozgrywki.

* bool isFullBoard();

Metoda zwracająca wartość bool, w zależności czy wszystkie pola planszy są zdefiniowane.

* bool gameOver() override;

Metoda zwracająca wartość bool i wypisująca na ekran informację kto jest zwycięzcą w zależności czy wszystkie pola planszy są zdefniowane i pobierająca informację z metody getWinner(), aby móc podać jaki pionek został zwycięzcą.

* std::pair<int, int> checkWillWin(char value);

Metoda zwracająca koordynaty pola, w które trzeba wstawić pionek, aby zwyciężyć w kolejnym ruchu.

* std::pair<int, int> chooseSecondMove(char value);

Metoda zwracająca koordynaty pola, w które trzeba wstawić pionek, aby w drugim ruchu stworzyć szansę na wygraną w kolejnym.

* std::pair<int, int> emptyLineFullOpponentSq(char value);

Metoda zwracająca koordynaty pola, w które trzeba wstawić pionek, aby zapełnić pole w linii, w której nie ma żadnego zdefiniowanego pola. Metoda, próbuje również wybrać pole, które byłoby najkorzystniejszym wyborem dla przeciwnika.

* std::pair<int, int> defaultMove();

Metoda zwracająca koordynaty pierwszego znalezionego pola, które jest puste.

* void moveAIHard(char value, char opponentValue);

Metoda definiująca ruch bota, grającego strategią agresywną – bezwzględnej wygranej lub remisu. Bot podejmuje decyzję wykorzystując funkcje checkWillWin(char value), chooseSecondMove(char value), emptyLineFullOpponentSq(char value), defaultMove(). Dodatkowo, od samego początku gry próbuje zająć środek planszy.

* void moveAIMedium(char value, char opponentValue);

Metoda definiująca ruch bota, grającego strategią przeciętną – nastawioną na wygraną, ale jest możliwe jego pokonanie. Bot podejmuje decyzję wykorzystując funkcje checkWillWin(char value), chooseSecondMove(char value), defaultMove().

* void moveAIEasy(char value, char opponentValue);

Metoda definiująca ruch bota, grającego najprostszą strategią – ruchy są wybierane losowo. Bot podejmuje decyzję wykorzystując funkcję defaultMove().

* int chooseStartingPlayer();

Metoda wykorzystuje moduł random, wybierając losowo, który gracz powinien rozpocząć rozgrywkę.

* void display() override;

Metoda wypisująca na ekran terminala, planszę do gry. Jest to redefinicja metody klasy abstrakcyjnej Games().

* void play() override;

Metoda umożliwiająca grę w kółko krzyżyk. Komunikuje się z graczem, umożliwiając mu wybór, czy chciałby grać z drugim graczem czy z botem. Wtedy również jest wybierany poziom umiejętności bota. Następnie prowadzi gracza przez całą rozgrywkę, na końcu zwracając informację o zwycięzcy. Jest to redefinicja metody klasy abstrakcyjnej Games().

* char getValue(int column, int row) override;

Implementacja czysto wirtualnej metody klasy bazowej pozwalająca na dostęp do komórki planszy.

Klasa Sudoku:

Klasa definiuje wszystkie metody umożliwiające rozwiązywanie sudoku. Możliwe jest wygenerowanie planszy o trzech poziomach zaawansowania – trudnej, średniej oraz prostej. Klasa dziedziczy po klasie abstrakcyjnej Games.

Stałe oraz makra:

Za pomocą dyrektyw preprocesora w czasie kompilacji są tworzone stałe:

emptySquare 0 - definicja pola pustego

N 9 – wymiar tablicy

rows 9 – ilość wierszy

columns 9 - ilość kolumn

solvable 18 – minimalna ilość pól, które muszą być początkowo zdefiniowane, aby rozwiązanie planszy było możliwe.

hard 40 – górna granica ilości zdefiniowanych pól, kiedy generowana jest plansza trudna.

medium 65 – górna granica ilości zdefiniowanych pól, kiedy generowana jest plansza średnia.

Atrybuty prywatne:

* int\*\* board;

Plansza do gry

* std::pair<int, int> difficultyLevel;

Poziom trudności planszy -> zakres pól, które muszą pozostać zdefiniowane.

* int guessNum[9] = {1, 2, 3, 4, 5, 6, 7, 8, 9};

Tablica numerów, które są uważane za pola zdefiniowane.

* int gridPos[81];

Tablica przechowująca pola, służące do losowania pól podczas generowania planszy.

Metody publiczne:

* explicit Sudoku(std::string difficultyLevelValue);

Konstruktor klasy. Aby utworzyć obiekt klasy, konieczne jest podanie poziomu planszy sudoku, która ma być wygenerowana.

* ~Sudoku();

Destruktor klasy.

* int\*\* getBoard();

Metoda umożliwiająca dostęp do atrybutu prywatnego board.

* std::string toString(int\*\* tab);

Metoda konwertująca tablicę intów na ciąg znaków.

* std::pair<int, int> getDifficultyLevel();

Metoda umożliwiająca dostęp do atrybutu prywatnego DifficultyLevel.

* void fillBoard(int\*\* tab);

Metoda uzupełniająca planszę podaną przez paramter tab tablicą intów.

* void assignDiffcultyLevel(std::string difficultyLevelValue);

Metoda obliczająca w jakim zakresie powinna zmieść się ilość pól zdefiniowanych na planszy Sudoku, na podstawie podanego poziomu trudności.

* int calcDifficultyLevel(int\*\* tab);

Metoda obliczająca ile pól na planszy nie zostało zdefiniowanych.

* void createEmptyBoard(int\*\* tab);

Metoda tworząca pustą planszę i przypisująca ją do tablicy przekazanej jako parametr.

* bool usedInRow(int row, int value, int\*\* tab);

Metoda sprawdzająca czy dana liczba została wcześniej użyta w danym rzędzie.

* bool usedInColumn(int column, int value, int\*\* tab);

Metoda sprawdzająca czy dana liczba została wcześniej użyta w danej kolumnie.

* bool usedInBox(int row, int column, int value, int\*\* tab);

Metoda sprawdzająca czy dana liczba została wcześniej użyta w danym kwadracie 3x3.

* bool checkRepetition(int row, int column, int value, int\*\* tab);

Metoda wywołuję kolejno: usedInRow(int row, int value, int\*\* tab),

usedInColumn(int column, int value, int\*\* tab),

usedInBox(int row, int column, int value, int\*\* tab).

Zwracając na tej podstawie wartość boolowską, czy dana liczba może być wstawiona w dane pole zgodnie z zasadami gry w sudoku.

* bool isEmpty(int& row, int& column, int\*\* tab);

Metoda sprawdzająca czy zdefiniowane za pomocą współrzędnych pole w planszy jest puste.

* void insert(int row, int column, int num);

Metoda wstawiająca wartość w zdefiniowane za pomocą współrzędnych pole.

* bool isSolved(int\*\* tab, int& i, int& j);

Metoda sprawdzająca czy wszystkie pola planszy są zdefiniowane.

* void generateSolvedBoard();

Metoda, która generuje rozwiązane sudoku, którego wszystkie pola są poprawnie wypełnione. Metoda, wywołuje w swoim ciele createEmptyBoard(int\*\* tab) oraz fillBoard(int\*\* tab). Funkcja jest wykonywana rekurencyjnie, aż do momentu utworzenia poprawnie uzupełnionej planszy, która jest następnie przypisywana do atrybutu board.

* void generateStartBoard();

Metoda, bazuje na algorytmie backtrackingu, który usuwając kolejno zdefiniowane pola sprawdza czy plansza jest nadal jednoznacznie rozwiązywalna. Wywołuje w swoim ciele generateSolvedBoard() oraz countSolutions(int& number, int row, int col).

* void countSolutions(int& number, int row, int col);

Metoda zlicza ilość możliwych rozwiązań planszy, która jest przekazywana przez referencję parametrem number.

* bool solve(int row, int col, int\*\* tab);

Metoda sprawdzająca czy rozwiązanie Sudoku jest możliwe.

* bool gameOver() override;

Implementacja czysto wirtualnej metody sygnalizująca koniec gry.

* char getValue(int column, int row) override;

Implementacja czysto wirtualnej metody pozwalająca na łatwy dostęp do komórki planszy.

* void play() override;

Metoda umożliwiająca uzupełnianie Sudoku. Na początku generowana jest plansza, do momentu, aż zostanie utworzona taka, która odpowiada poziomem trudności – temu wskazanemu przez gracza, przy tworzeniu obiektu klasy Sudoku. Następnie prowadzi gracza przez całą rozgrywkę, aż do uzupełnienia wszystkich pól na planszy. Jest to redefinicja metody klasy abstrakcyjnej Games().

* void display() override;

Metoda wypisująca na ekran terminala, planszę do gry. Jest to redefinicja metody klasy abstrakcyjnej Games().

Klasa Crossword:

Klasa definiuje wszystkie metody umożliwiające rozwiązywanie krzyżówek. Baza haseł jest podawana w pliku z rozszerzeniem txt, w konstruktorze klasy. Klasa dziedziczy po klasie abstrakcyjnej Games.

Stałe oraz makra:

Za pomocą dyrektyw preprocesora w czasie kompilacji jest tworzona stała:

emptySquare '\_'- definicja pola pustego

Atrybuty prywatne:

* std::pair<std::string, std::string> crosswordClue;

Przechowuje hasło główne krzyżówki wraz z opisem.

* std::vector<std::pair<int, std::pair<std::string, std::string>>> crosswordClues;

Przechowuje hasła wykorzystane w krzyżówce wraz z ich numerem w krzyżówce oraz opisem.

* std::vector<std::pair<int, std::pair<std::string, std::string>>> crosswordCluesUser;

Przechowuje hasła wprowadzone przez użytkownika do krzyżówki, wraz z ich numerem w krzyżówce oraz opisem.

* std::vector<std::pair<std::string, std::string>> clues;

Przechowuje hasła pobrane z bazy haseł, wraz z ich opisem.

* std::vector<std::vector<char>> board;

Plansza przechowująca znaki w miejscach odpowiadających ich faktycznej pozycji

Metody publiczne:

* Crossword();

Konstruktor klasy.

* ~Crossword() = default;

Domyślny destruktor klasy.

* std::string getClue();

Metoda umożliwiająca dostęp do pierwszego pola atrybutu prywatnego crosswordClue.

* std::string getClueInfo();

Metoda umożliwiająca dostęp do drugiego pola atrybutu prywatnego crosswordClue.

* void chooseClueRandomly();

Metoda wybierająca losowo z bazy, hasło główne krzyżówki i przypisująca je do atrybutu crosswordClue.

* std::vector<std::pair<std::string, std::string>> getClues();

Metoda umożliwiająca dostęp atrybutu prywatnego clues.

* std::vector<std::pair<int, std::pair<std::string, std::string>>> getCrosswordClues();

Metoda umożliwiająca dostęp atrybutu prywatnego crosswordClues.

* std::vector<std::pair<int, std::pair<std::string, std::string>>> getCrosswordCluesUser();

Metoda umożliwiająca dostęp atrybutu prywatnego crosswordCluesUser.

* bool isEmpty(int& num);

Metoda sprawdzająca czy zdefiniowane za pomocą numeru hasło na planszy nie zostało uzupełnione.

* void addClue(std::string Clue, std::string ClueInfo);

Metoda umożliwiająca dodanie hasła do bazy haseł.

* bool findLetter(char letter, std::pair<std::string, std::string> word);

Metoda sprawdzająca czy podane za pomocą parametru hasło zawiera konkretną literę.

* int findLetterPos(char letter, std::string word);

Metoda sprawdzająca na której pozycji podane za pomocą parametru słowo zawiera konkretną literę.

* int findMaxLetterPos();

Metoda zwracająca maksymalne przesunięcie litery, tworzącej hasło główne krzyżówki od lewej strony. Metoda wykorzystywana do wypisywania na ekran.

* void chooseCrosswordClues();

Metoda wybierająca losowo hasła, z bazy haseł krzyżówek, które zawierają konkretne litery hasła głównego i ustawia je w kolejności wypisywania. Następnie przypisująca je do atrybutu crosswordClues. Wybieranie haseł następuje do momentu utworzenia krzyżówki.

* void insert(int num, std::string clueValue);

Metoda wstawiająca dane hasło w zdefiniowany za pomocą numeru rząd.

* std::string encryptClue(std::string clue);

Metoda szyfrująca dane hasło. Jej litery są zamieniane na znak zdefiniowany przez emptySquare. Metoda używana do wypisywania na ekran.

* friend std::ostream& operator<<(std::ostream& COUT, Crossword& crossword);

Funkcja zaprzyjaźniona klasy Crossword. Umożliwia wypisanie zawartości bazy haseł na ekran.

* friend std::istream& operator>>(std::istream& CIN, Crossword& crossword);

Funkcja zaprzyjaźniona klasy Crossword. Umożliwia wpisanie zawartości bazy haseł do strumienia.

* void display() override;

Metoda aktualizująca atrybut board, oryginalnie wypisywała hasła z przesunięciem do terminala.

* void solveCrossword();

Metoda rozwiązuje krzyżówkę. Wypełnia atrybut crosswordCluesUser poprawnymi hasłami.

* bool isSolved();

Metoda sprawdzająca czy wszystkie hasła w krzyżówce są uzupełnione (czy plansza jest wypełniona).

* bool checkCorrectness();

Metoda sprawdzająca czy wszystkie hasła w krzyżówce są uzupełnione poprawnie.

* bool gameOver() override;

Implementacja czysto wirtualnej metody, sygnalizuje zakończenie gry.

* char getValue(int column, int row) override;

Implementacja czysto wirtualnej metody, pozwala na dostęp do komórki planszy.

* void play() override;

Metoda umożliwiająca uzupełnianie krzyżówek. Prowadzi gracza przez całą rozgrywkę, aż do uzupełnienia wszystkich pól na planszy. Jest to redefinicja metody klasy abstrakcyjnej Games().

* 1. Użytkownik [Bartek]
  2. Wyświetlanie

Klasa Tile:

Klasa reprezentująca najmniejszy element gry planszowej – kwadrat posiadający wartość. Kolor tła, obwódki oraz wyświetlanej wartości może być modyfikowany. W razie potrzeby Tile może nie być renderowany. Klasa ta dziedziczy po klasie RectangleShape z biblioteki SFML/Graphics.

Atrybuty prywatne:

* bool visibility;

Widoczność obiektu – determinuje czy będzie renderowany.

* sf::Vector2f position;

Położenie lewego górnego rogu obiektu wyrażone w przesunięciu od lewego górnego rogu okna.

* char value;

Wyświetlana wartość.

* sf::Text displayValue;

Obiekt tekstowy pozwalający na wyświetlenie przechowywanej wartości.

Metody publiczne:

* Tile() = default;

Domyślny konstruktor.

* Tile(

sf::Vector2f position,

sf::Color frameColor,

sf::Color backgroundColor,

sf::Color displayColor,

sf::Font& font,

double size,

bool isVisible,

char value = 0

);

Zparametryzowany konstruktor.

* void draw(sf::RenderWindow& window);

Metoda rysująca obiekt na przekazanym oknie.

* char getValue();

Getter pola value.

* void setValue(char);

Setter pola value.

* bool isVisible();

Getter pola visibility.

* void setVisibility(bool);

dSetter pola visibility.

Klasa Board:

Klasa reprezentująca kwadratową grę planszową. Składa się z obiektów klasy Tile. Ilość elementów (bok) jest zmienna.

Prywatne:

* int elemNum;

Ilość elementów w boku planszy.

Chronione:

* sf::Vector2f position;

Położenie lewego górnego rogu obiektu wyrażone w przesunięciu od lewego górnego rogu okna.

* double size;

Długość boku.

* Tile \*\* board;

Dynamicznie alokowana dwuwymiarowa tablica obiektów Tile.

* std::pair<int, int> selected;

Indeks aktualnie zaznaczonej komórki.

Publiczne:

* Board(

sf::Vector2f position,

sf::Color frameColor,

sf::Color backgroundColor,

sf::Color displayColor,

sf::Font& font,

double size,

int elemNum,  
 sf::RenderWindow &window

);

Zparametryzowany konstruktor.

* sf::RenderWindow &m\_window

Referencja na obiekt okna w którym rysowane będą elementy.

* virtual std::pair<int, int> getIndex(sf::Vector2i);

Zwraca indeks komórki wskazywanej przez wektor (myszkę).

* virtual Tile& getTile(std::pair<int, int> index);

Getter komórek planszy.

* virtual void draw();

Rysuje wszystkie przechowywane komórki Tile.

* virtual void update();

Pojedyńczy cykl “wyczyść okno – narysuj – wyświetl na oknie”.

* virtual void display(Games &game);

Pobranie danych o planszy z dowolnej gry dziedziczącej po Games, wywołanie update().

* void display2(Games& game);

Metoda używana do wyświetlenia instancji Crossword (docelowo nie powinna być używana).

* virtual void set\_selected(std::pair<int, int> index);

Setter pola selected.

* 1. Wykorzystane moduły

Do stworzenia programu wykorzystaliśmy biblioteki:

* iostream
* random
* vector
* utility
* fstream
* algorithm
* string
* CppUnitTestFramework
* SFML/Graphics
* SFML/System

1. Wyjątki[Kieru][Bartek]

Zdecydowaliśmy się na utworzenie własnych wyjątków, które w naszej opinii dokładniej wskazują, dlaczego użytkownik wykonał operacje niedozwoloną przez nasz program. Te wyjątki to:

* ?

1. Testy

Stworzony przez nas program przeszedł [ilość testów] testów, które dotyczą każdego modułu naszego programu. Ponadto postanowiliśmy wykonać testy manualne. W ten sposób uzyskaliśmy pierwsze zastrzeżenia co do działania programu.

Testowanie 1 etap:

Informacja, która otrzymaliśmy wskazywała, że w grze TicTacToe zabrakło większej różnorodności w strategiach gry wybieranych przez bota. Początkowo jego jedyną strategią było dążenie do wygranej w jak najmniejszej liczbie ruchów. Ponadto okazało się, że dla Crossword baza haseł była zbyt mała i w jednym na około dwadzieścia przypadków, nie udawało się utworzyć tablicy z krzyżówkami. Wtedy doszło do najważniejszych poprawek w działaniu bota – metodzie moveAI, która została rozbudowana na trzy kolejne MoveAIHard, moveAIMedium, moveAIEasy. Następnie zdecydowaliśmy się znacznie poszerzyć bazę haseł, aby nie było możliwości błędu przy tworzeniu obiektu klasy Crossword.

Testowanie 2 etap:

Testowanie 3 etap:

Początkowe testowanie elementów odbywało się wyłącznie manualnie, by nabrać doświadczenia w posługiwaniu się bibliotekami SFML. Wygląd komponentów nie jest elementem szczególnie odpowiednim do testów automatycznych. Testowana z tego powodu jest głównie logika klas.

Następnie dopisaliśmy odpowiednie testy jednostkowe i pozwoliliśmy przetestować użytkownikowi nasz program. Okazało się, że projekt odpowiada wymaganiom oraz wykazuje się dużą różnorodnością strategii i możliwych rozgrywek na co wskazywali nasi testerzy.

1. Problemy

Tworząc logikę zauważyliśmy, że przechowywanie wszystkich gier w tablicy składającej się z takich samych typów, może nie być odpowiednim podejściem programistycznym. Aby stworzyć gry, które wyglądają i działają podobnie jak te do, których użytkownik przywykł grając na kartce papieru, trzeba było zdecydować się na pewne kompromisy. Największym z nich z pewnością jest decyzja, aby użytkownik uzupełniając krzyżówki wpisywał całe słowa zamiast pojedynczych liter. Możliwe, że w przyszłości dla większej spójności logiki gier można by, uzupełniać tablicę elementów typu char, natomiast wymagałoby to przebudowy części programu związanej z logiką gier. [Bartek][Kieru]

Z pewnością nie utrudnia to korzystania z programu, ale jeśli kiedyś byłby to produkt, który miałby być wprowadzony na rynek, mógłby przegrać z konkurencją w związku z [?? ].

1. Podsumowanie

Cały projekt został przez nas napisany w języku C++. Jednakże nie wszyscy członkowie naszego zespołu mieli wcześniejsze doświadczenie z programowaniem w języku statycznie typowanym. Z tego powodu napotkaliśmy pewne przeszkody, ale dzięki zróżnicowanej wiedzy na dane tematy byliśmy w stanie pomóc sobie nawzajem i je przezwyciężyć. Jesteśmy zadowoleni z podziału obowiązków, ponieważ był to projekt, w którego tworzenie był zdecydowanie zaangażowany każdy członek naszej grupy. Efektem naszej pracy jest produkt, który mamy nadzieje, że odpowiada na temat, który został nam przydzielony. Wierzymy, że może być on konkurencyjny w porównaniu do rozwiązań proponowanych przez studentów z podobnym do naszego doświadczeniem.