Dokumentacja projektu z Programowania Obiektowego

Nazwa projektu: Kolokwiarz

Prowadzący: mgr inż. Maciej Radecki

Krótki opis projektu:

Aplikacja okienkowa będąca grą quizową, której głównym celem jest pomoc w nauce na kolokwia EKA W12N.

Spis Treści

1. Ważne informacje wstępne

1.1. Specyfikacje i wskazania do uruchamiania programu

Projekt był robiony na systemie Windows 10 i nie był testowany na innych systemach operacyjnych. Program nie używa wywołań systemowych, więc po skompilowaniu projektu na komputerze z Linux lub MacOS powinien działać, ale jednak Windows jest pewny.

Po kilku testach wynika, że aplikacja aktualnie **nie potrzebuje więcej niż 20 MB RAM**.

Folder główny projektu zawiera foldery:

- *base* – zawiera pliki baz pytań w formacie .json;
- *bin/release* – zawiera plik .exe programu oraz rozszerzenia umożliwiające jego aktywacje.
- *core* – zawiera pliki klas odpowiadających za logikę programu;
- *gui* – zawiera pliki klas Qt integrujące logikę z interfejsem graficznym aplikacji;
- *styles* – zawiera pliki tekstowe .qss będące arkuszami stylu (motywami) aplikacji;
- *utils* – WAŻNE plik *utils(.h i .cpp)* zawiera funkcje pomocnicze do szukania ścieżek pewnych plików i każda z tych funkcji szuka ich 2 foldery wyżej względem pliku .exe. Jeśli zostanie on przeniesiony do lokalizacji nie zachowującej tej odległości, program przestanie działać.

  Jakie funkcje dostarcza utils:

  - **QString** getUsersFilePath() – funkcja zwracająca ścieżkę do pliku users.json jako QString
  - **QString** getQuestionsFilePath(const QString &topicName) – funkcja zwracająca ścieżkę do odpowiedniego pliku bazy pytań jako QString [❗ nazwy plików z pytaniami muszą być w formie „to_jest_nazwa.json” inaczej funkcja nie znajdzie ich ❗]
  - **QString** getStyleFilePath(const QString &styleName) – funkcja zwraca ścieżkę do odpowiedniego arkusza stylu jako QString [nazwy tych arkuszy muszą być nazwane w ten sam sposób co bazy pytań ale z rozszerzeniem „.qss”]

Folder główny zawiera również:

- *CMakeLists.txt;*
- *Doxyfile* – skonfigurowany na generacje dokumentacji w HTML;
- Pliki *main* (*main.cpp i mainwindow*);
- *users.json* – plik użytkowników.

1.2. Założenia funkcjonalne i niefunkcjonalne

Funkcjonalne:

- Program powinien obsługiwać pliki .json;
- Program powinien być łatwo rozszerzalny o nowe bazy pytań;
- Program powinien obsługiwać co najmniej 10 użytkowników (system logowania);
- Program powinien służyć do nauki;

Niefunkcjonalne:

- Program powinien zawierać tryb treningowy i rankingowy;
- Program powinien zawierać ranking użytkowników według punktów;
- Program powinien informować użytkownika czy udzielił poprawnej odpowiedzi podczas quizu;
- Program powinien mieć specjalnego użytkownika upoważnionego do edycji baz pytań z poziomu aplikacji;
- Program powinien mieć co najmniej 2 motywy (jasny i ciemny), możliwe do zmiany w każdej chwili;

2. Klasy zarządzające logiką (*core*)

2.1. User

Klasa definiująca użytkownika.

Konstruktor:

User(**QString** name, **QString** pass, **int** points = 0, **int** games = 0, **QDateTime** login = QDateTime::currentDateTime());

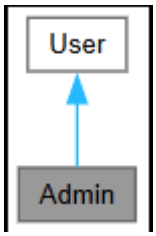
Pola (specyfikator dostępu **protected**):

- **QString** username – nazwa użytkownika;
- **QString** password – hasło;
- **int** totalPoints – suma punktów zdobytych podczas używania aplikacji;
- **int** totalGames – ilość zakończonych gier w aplikacji;
- **QDateTime** lastLogin – zmienna przechowująca dokładną datę ostatniego logowania;

Metody publiczne:

- **void** addPoints(int pts) – dodaje punkty do pola *totalPoints*;
- **void** incrementGames() – dodaje +1 do pola *totalGames*;
- **QString** getStats() const – zwraca wszystkie pola obiektu jako sformatowany QString;
- **QString** getUsername() const – zwraca nazwę użytkownika;
- **QString** getPassword() const – zwraca hasło;
- **int** getTotalPoints() const – zwraca ilość punktów użytkownika;
- **int** getTotalGames() const – zwraca ilość spędzonych gier użytkownika;
- **QDateTime** getLastLogin() const – zwraca czas ostatniego zalogowania;
- **void** setLastLogin(QDateTime dateTime) – ustawia pole *lastLogin*;
- **void** resetStats() – ustawia pola *totalPoints* i *totalGames* na zero;
- **bool** operator<(const User &other) const – przeładowanie operatora '<' w celu porównywania użytkowników względem punktów;
- **bool** isAdmin() – dla obiektów klasy User zwraca **false**;

2.2. Admin



Dziedziczy po klasie User, definiuje specjalnego użytkownika – administratora.

Zawiera te same pola, metody i konstruktor o takich samych poziomach dostępu. Różni się jedynie metodą isAdmin(), bo dla obiektu klasy Admin zwraca **true**.

2.3. Question

Klasa abstrakcyjna definiująca pytanie. Służy jako szablon do klas bardziej specyficznych typów pytań.

Pola:

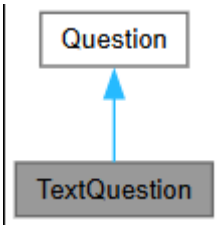
- **QString** text – tekst pytania;
- **QStringList** options – odpowiedzi do wyboru;
- **int** correctOprionIndex – index poprawnej odpowiedzi;

Metody publiczne:

- **bool** isCorrect(int optionIndex) – zwraca prawdę jeśli zaznaczono poprawną odpowiedź;
- **QString** getQuestionText() const – zwraca pole *text*;
- **QStringList** getOprions() const – zwraca pole *options*;

- **Int** getCorrectIndex() const – zwraca pole *correctOptionIndex*;

2.4. TextQuestion



Dziedziczy po Question. Jest konkretną implementacją pytania najzwyklejszego pytania zamkniętego.

Zawiera dodatkowy konstruktor (oprócz domyślnego dziedziczonego z klasy Question): TextQuestion(const QString &text, const QStringList &options, int correctOptionIndex)

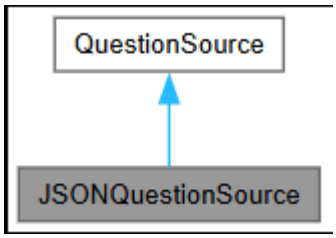
Oraz metodę **void** shuffleOptions(), która korzystając z generatora liczb losowych przetasowuje elementy w polu *options*.

2.5. QuestionSource

Klasa abstrakcyjna zawierająca tylko metodę **QVector<TextQuestion>** getQuestions() oraz destruktor.

Istnieje tylko jako szablon w razie gdyby bazy pytań miały być implementowane z innych plików niż .json.

2.6. JSONQuestionSource



Klasa obsługująca pliki .json. Dziedziczy po QuestionSource. Zaprzyjaźniona z klasą QuestionEditor.

Pola:

- **QString** filePath – ścieżka do pliku;
- **QVector<TextQuestion>** questions – wektor przechowujący pytania;

Metody prywatne:

- **void** loadQuestionsFromFile() – wczytuje plik z .json i zapisuje zawarte tam pytania jako obiekty TextQuestion do pola klasy *questions*;
- **bool** saveQuestionsToFile(const QVector<TextQuestion> &questions) – metoda do zapisywania zmian w plikach baz pytań;

Klasa zawiera także metodę getQuestions() odziedziczoną od przodka oraz posiada swój własny konstruktor:

explicit JSONQuestionSource(const QString &path);

2.7. QuestionEditor

Zaprzyjaźniona z klasą QuestionAdder (klasą GUI). Zawiera same statyczne metody służące do edytowania konkretnych plików baz pytań .json.

Metody:

- **static bool** addQuestion(const Admin& admin, const TextQuestion& question, JSONQuestionSource& source);
- **static bool** removeQuestion(const Admin& admin, int index, JSONQuestionSource& source);
- **static bool** editQuestion(const Admin& admin, int index, const TextQuestion& updatedQuestion, JSONQuestionSource& source);

Te metody mogą być wywołanie jedynie przez aktualnie zalogowanego admin.

2.8. QuizManager

Pierwsza duża klasa tego programu. Ogólnie służy do zarządzania quizem.

Konstruktor domyślny.

Pola:

- **int** currentScore = 0 – pole zapisujące aktualny wynik podczas grania w quiz, domyślnie ustawione na 0;
- **int** currentQuestionIndex = 0 – pole wskazujące, na którym aktualnie pytaniu ze zbioru jesteś;
- **int** correctAnswers = 0 – pole przechowujące ilość poprawnie odpowiedzianych pytań w ciągu quizu;
- **QVector<TextQuestion>** questions – pole (wektor) przechowujący zbiór pytań do aktualnego quizu;
- **std::unique_ptr<QuestionSource>** QuestionSource – pole (inteligentny wskaźnik) wskazujące na odpowiedni plik, z którego będą brane pytania;
- **QElapsedTimer** timer – pole odliczające czas od rozpoczęcia quizu;
- **QString** quizTopic – pole zawierające nazwę tematu. Używana głównie w celu wyznaczenia pola *QuestionSource*;
- **qint64** startTime – pole zapisujące czas w milisekundach kiedy zaczęło się nowe pytanie. Używane tylko przy wybraniu trybu rankingowego by obliczać punkty;
- **int** timeLimit = 30 – pole ustawiające limit czasu na pytanie w trybie rankingowym;

Metody:

- **void** setQuestionSource(std::unique_ptr<QuestionSource> source) – ustawia pole *QuestionSource*;
- **QVector<TextQuestion>** randomizeQuestions(const QVector<TextQuestion>& questions) – używając generatora liczb losowych przetasowuje kolejność pytań w wektorze pytań i zwraca pełny wektor;
- **void** randomizeQuestions(int questionAmount) – obcina wektor pytań do określonej ilości;
- **void** loadQuestions() – załadowuje wektor pytań ze źródła określonego przez pole *QuestionSource*;
- **void** startQuiz() – rozpoczyna quiz (resetuje pola currentScore, currentQuestionIndex i zaczyna wywołuje metodę startTimer());
- **void** startTimer() – rozpoczyna timer;
- **double** getTimeInSeconds() const – zwraca czas od ropoczęcia pytania w sekundach;
- **void** markQuestionStart() – ustawia pole *startTime*;
- **double** getTimeSinceQuestionStart() const;
- **void** setTopicName(const QString& topicName) – ustawia pole *quizTopic*;
- **QString** getTopicName() const – zwraca pole *quizTopic* jako QString;
- **TextQuestion** getCurrentQuestion() const – zwraca pytanie (obiekt TextQuestion) wskazane przez pole currentQuestionIndex;
- **void** nextQuestion() – inkrementuje o 1 pole currentQuestionIndex jeśli metoda hasNextQuestion() zróci prawdę,;
- **bool** checkAnswer(int answerIndex) – sprawdza odpowiedź użytkownika;
- **bool** hasNextQuestion() const -sprawdza czy jest jeszcze jedno pytanie;
- **void** submitAnswer(int answerIndex) – zatwierdza odpowiedź użytkownika i wywołuje metodę checkAnswer(int answerIndex);
- int getCurrentScore() const – zwraca pole *currentScore*;
- int getCurrentQuestionIndex() const – zwraca pole *currentQuestionIndex*;
- int getTotalQuestions() const – zwraca ilość pytań w wektorze *questions*;
- int getCorrectAnswerCount() const – zwraca pole *correctAnswers*;
- void calculateScore(int selectedIndex) – oblicza i dodaje punkty dla odpowiedniego pytania na bazie prędkości odpowiedzi;
- void resetQuiz() – zeruje pola *currentScore*, *currentQuestionIndex*, *correctAnswers* oraz wyłącza timer;
- void setTimeLimit(int seconds) – ustawia pole *timeLimit*;

- int getTimeLimit() const – zwraca pole *timeLimit*;

2.9. UserManager

Klasa zarządzająca zbiorami użytkowników: odpowiedzialna za wczytywanie i zapisywanie listy użytkowników, uwierzytelnianie i rejestrację.

Konstruktor domyślny.

Pola:

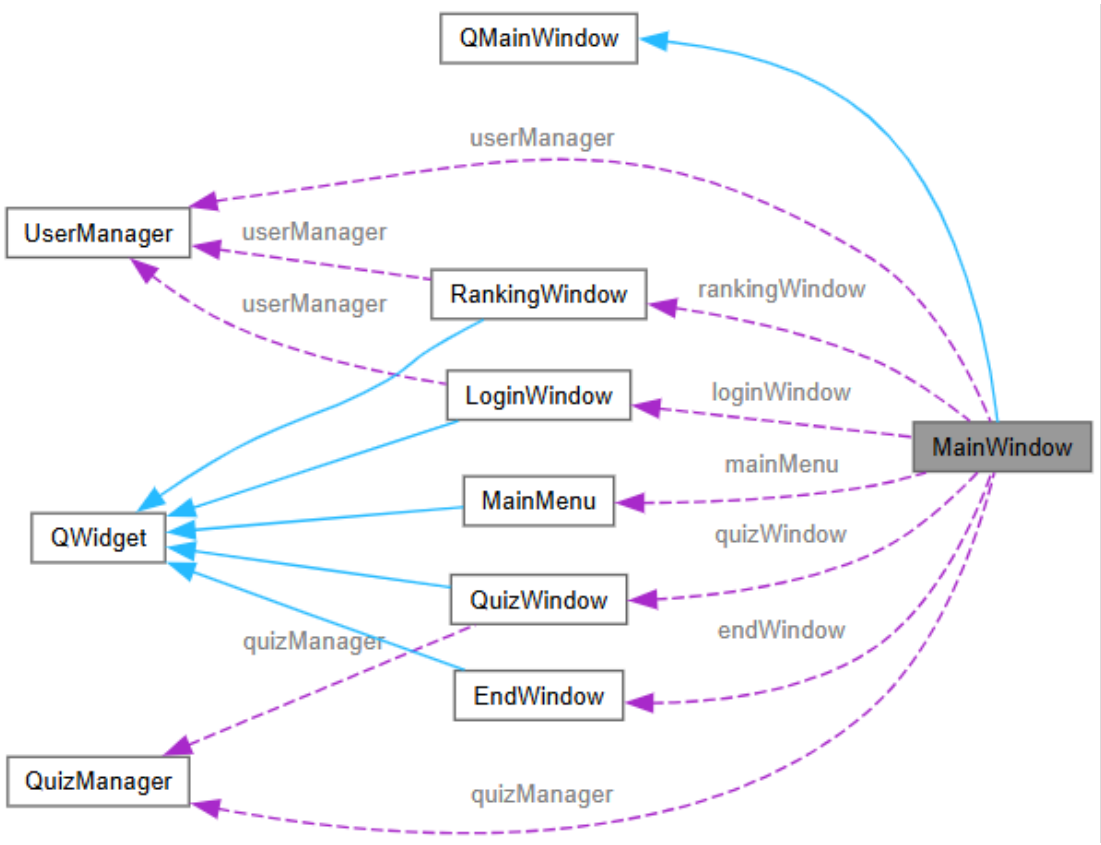
- **QVector<std::shared_ptr<User>>** users – wektor wskaźników do obiektów User przechowujący załadowanych użytkowników;

Metody publiczne:

- **QVector<std::shared_ptr<User>>** getUsers() const – zwraca wektor wszystkich użytkowników aktualnie przechowywanych w menedżerze;
- **void** loadUsersFromFile(const QString& filepath) – wczytuje użytkowników z pliku JSON o ścieżce filepath i tworzy obiekty User; w przypadku błędu parsowania lub braku pliku – odpowiednia obsługa błędu;
- **void** saveUsersToFile(const QString& filepath) – zapisuje aktualny wektor użytkowników do pliku JSON o ścieżce filepath; zwraca lub loguje (qDebug) błąd w razie niepowodzenia zapisu;
- **std::shared_ptr<User>** login(const QString& username, const QString& password) – próbuje uwierzytelnić użytkownika o podanej nazwie i haśle; jeżeli dane poprawne, zwraca wskaźnik do obiektu User (aktualizując lastLogin); w przeciwnym razie zwraca nullptr;
- **std::shared_ptr<User>** registerUser(const QString& username, const QString& password, QString* error = nullptr) – próbuje zarejestrować nowego użytkownika; sprawdza unikalność nazwy użytkownika, tworzy nowy obiekt User i dodaje do wektora; w przypadku błędu (np. nazwa zajęta, złe hasło) zwraca nullptr i (opcjonalnie) wpisuje komunikat błędu w `*error`; w razie powodzenia zwraca wskaźnik do nowego obiektu User; Jeśli rejestrowany jest pierwszy użytkownik, staje się on automatycznie obiektem klasy Admin;
- **bool** validatePassword(const QString& username, const QString& password) const – sprawdza, czy podane hasło pasuje do danego użytkownika; zwraca true, jeżeli hasło jest poprawne; używana wewnątrz login lub przy innych operacjach wymagających weryfikacji.

3. Klasy integrujące logikę z GUI (gui)

3.1. MainWindow



Klasa reprezentująca główne okno aplikacji. Odpowiada za zarządzanie ekranami interfejsu graficznego (GUI) – logowania, quizu, menu głównego, rankingu oraz podsumowania. Obsługuje również logikę przełączania widoków, zmianę motywu oraz interakcję z użytkownikiem.

Konstruktor domyślny klasy Qt:

explicit MainWindow(QWidget *parent = nullptr) – inicjalizuje wszystkie składniki interfejsu, tworzy obiekty podrzędnych okien (LoginWindow, QuizWindow, itd.), ustawia początkowy styl graficzny oraz łączy odpowiednie sygnały i sloty.

Destruktor usuwa wskaźniki do UserManager, QuizManager i ui.

Pola:

- **Ui::MainWindow** *ui – wskaźnik do wygenerowanego interfejsu użytkownika;
- **QActionGroup** *stylesGroup – grupa akcji odpowiadająca za wybór stylu aplikacji;
- **QStackedWidget** *stackedWidget – widżet umożliwiający przełączanie między różnymi ekranami GUI;

- **UserManager*** userManager – wskaźnik do menedżera użytkowników;
- **QuizManager*** quizManager – wskaźnik do menedżera quizu;
- **std::shared_ptr<User>** currentUser – wskaźnik do aktualnie zalogowanego użytkownika (lub nullptr, jeśli niezalogowany);
- **LoginWindow*** loginWindow – okno logowania użytkownika;
- **QuizWindow*** quizWindow – okno przeprowadzania quizu;
- **MainMenu*** mainMenu – ekran menu głównego;
- **RankingWindow*** rankingWindow – ekran rankingu użytkowników;
- **EndWindow*** endWindow – ekran podsumowania quizu (tworzony dynamicznie po zakończeniu quizu);
- **bool** userLoginStatus – flaga informująca, czy użytkownik jest aktualnie zalogowany.

Metody prywatne:

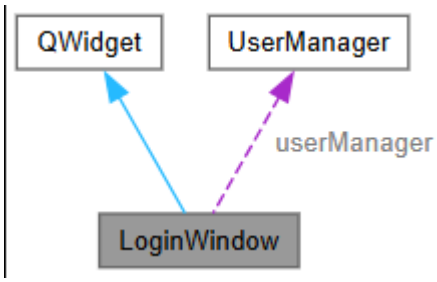
- **void** showLoginWindow() – przełącza widok na okno logowania;
- **void** showMainMenu() – przełącza widok na menu główne, jeśli nie jest puste;
- **void** showRankingWindow() – przełącza widok na ranking użytkowników;
- **void** logoutUser() – wylogowuje użytkownika, czyści dane i wraca do ekranu głównego;
- **void** setUserLoginStatus(bool status) – ustawia status logowania oraz aktualizuje napis przycisku logowania;
- **bool** isLoggedIn() const – sprawdza, czy użytkownik jest aktualnie zalogowany.

Sloty:

- **void** onLoginSuccess(std::shared_ptr<User> loggedInUser) – reakcja na poprawne logowanie; ustawia aktualnego użytkownika, zmienia przycisk na „WYLOGUJ” i zapisuje dane;
- **void** onQuizCompleted(const QString& topicName, int score, int correctAnswers) – reakcja na zakończenie quizu; tworzy i pokazuje okno podsumowania quizu (EndWindow);
- **void** on_loginButton_clicked() – obsługa kliknięcia przycisku „ZALOGUJ/WYLOGUJ”; w zależności od stanu zmienia widok na okno logowania lub wylogowuje użytkownika;
- **void** handleQuizStart(QString selectedTopic, bool isTrainingMode, int questionAmount = 10) – rozpoczyna nowy quiz w trybie treningowym lub rankingowym z wybraną liczbą pytań;

- **void** on_playButton_clicked() – sprawdza, czy użytkownik jest zalogowany; jeśli tak – otwiera menu główne, w przeciwnym wypadku pokazuje komunikat błędu;
- **void** on_rankingWindow_clicked() – przełącza widok na ranking;
- **void** handleQuizFinished(int score) – aktualizuje statystyki użytkownika po zakończonym quizie, resetuje quiz oraz wraca do menu;
- **void** on_actionDefaultTheme_triggered() – wczytuje i stosuje domyślny motyw graficzny aplikacji;
- **void** on_actionDodaj_Pytanie_triggered() – otwiera okno dodawania pytania (QuestionAdder), jeśli użytkownik jest administratorem i znajduje się na ekranie głównym lub rankingu;
- **void** on_actionDarkTheme_triggered() – wczytuje i stosuje ciemny motyw graficzny aplikacji.

3.2. LoginWindow



Klasa reprezentująca okno logowania użytkownika. Dziedziczy po QWidget; pozwala na wprowadzenie nazwy i hasła, próbę logowania lub rejestracji oraz powrót do głównego okna.

Konstruktor:

explicit LoginWindow(UserManager* userManager, QWidget *parent = nullptr) – inicjalizuje okno, ustawia wskaźnik do menedżera użytkowników (userManager), łączy sygnały i sloty GUI.

Pola:

- **Ui::LoginWindow** *ui – wskaźnik na wygenerowany interfejs użytkownika (pola QLineEdit, QPushButton itd.);
- **UserManager*** userManager – wskaźnik na instancję menedżera użytkowników dla operacji logowania/rejestracji;

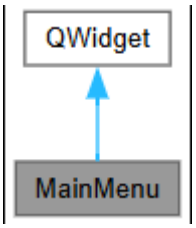
Sygnały:

- **void** loginSuccess(std::shared_ptr<User> loggedInUser) – emitowany, gdy logowanie się powiodło, przekazuje wskaźnik do zalogowanego użytkownika;
- **void** backToMainWindowRequested() – emitowany, gdy użytkownik chce powrócić do głównego menu bez logowania.

Sloty / Metody prywatne:

- **void** attemptLogin() – próbuje uwierzytelnić użytkownika na podstawie wprowadzonych danych; w razie sukcesu emituje loginSuccess, w razie błędu wyświetla komunikat;
- **void** attemptRegister() – próbuje zarejestrować nowego użytkownika; w razie sukcesu emituje loginSuccess, w razie błędu wyświetla komunikat;
- **void** on_backButton_clicked() – slot wywoływany po kliknięciu przycisku „Powrót”; emituje backToMainWindowRequested();
- void on_registerButton_clicked() – slot wywoływany po kliknięciu przycisku „Zarejestruj”; zwykle wywołuje attemptRegister();
- **void** on_loginButton_clicked() – slot wywoływany po kliknięciu przycisku „Zaloguj”; zwykle wywołuje attemptLogin();
- **void** clearFields() – czyści pola edycyjne (np. QLineEdit) po operacji lub przy inicjalizacji.

3.3. MainMenu



Klasa reprezentująca główne menu wyboru quizu/trybu. Dziedziczy po QWidget; umożliwia wybór tematu quizu, trybu (treningowy/rankingowy) i liczby pytań.

Konstruktor:

explicit MainMenu(QWidget *parent = nullptr) – Kod konstruktora wykorzystuje strukturę QuizTopic dostarczoną przez *utils.h*, by uzupełnić obiekty QComboBox.

Pola:

- **Ui::MainMenu** *ui – wskaźnik na wygenerowany interfejs użytkownika (np. QComboBox z tematami, przyciski startu, przycisk powrotu);
- **QList<QuizTopic>** quizTopics – lista dostępnych tematów quizu;
- **QButtonGroup** *modeButtonGroup – grupa przycisków wyboru trybu;

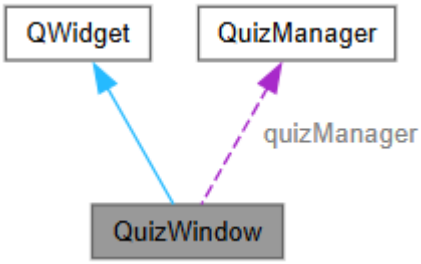
Sygnały:

- **void** startQuiz(QString selectedTopic, bool isTrainingMode, int questionAmount = 10) – emitowany, gdy użytkownik wybierze temat i tryb oraz kliknie „Start”; przekazuje wybrane parametry do uruchomienia quizu;
- **void** backToMainWindowRequested() – emitowany, gdy użytkownik chce wrócić do okna głównego.

Sloty / Metody prywatne:

- **void** setTopicsByCategory() – ustawia lub filtruje listę tematów według wybranej kategorii;
- **void** on_backButton_clicked() – obsługuje powrót do poprzedniego ekranu; emituje backToMainWindowRequested();
- **void** on_startGameButton_clicked() – obsługuje kliknięcie przycisku „Start”; zbiera parametry i emituje startQuiz(...);
- **void** onTopicChanged(const QString &topicName) – reaguje na zmianę wyboru tematu w QComboBox; ewentualnie aktualizuje podgląd opisu tematu lub dostępne liczby pytań.

3.4. QuizWindow



Klasa reprezentująca okno przebiegu quizu. Dziedziczy po QWidget; wyświetla pytania, odpowiedzi, timer oraz obsługuje logikę zaznaczania, zatwierdzania odpowiedzi, przejścia do kolejnego pytania.

Konstruktor:

explicit QuizWindow(QuizManager* quizManager, std::shared_ptr<User> loggedInUser, bool isTrainingMode, QWidget *parent = nullptr) – inicjalizuje GUI quizu z wskaźnikiem do QuizManagera i zalogowanym użytkownikiem, ustawia tryb treningowy/rankingowy; przygotowuje timer QTimer.

Pola:

- **Ui::QuizWindow** *ui – wskaźnik do wygenerowanego interfejsu (np. QLabel pytania, QRadioButtony opcji, QPushButton „Potwierdź”);
- **QuizManager*** quizManager – wskaźnik do menedżera quizu sterującego kolejnością pytań i punktacją;
- **std::shared_ptr<User>** loggedInUser – wskaźnik do zalogowanego użytkownika, do aktualizacji statystyk po zakończeniu quizu;
- **bool** isTrainingMode – flaga trybu; jeśli true, brak limitu czasu/punktacji rankingowej, jeśli false – liczenie punktów na podstawie czasu;
- **QTimer*** countdownTimer – timer odliczający czas pozostały na odpowiedź w trybie rankingowym, a w trybie treningowym tylko stoi na ustalonej liczbie;
- **bool** hasAnswered – flaga pomocnicza, czy użytkownik już odpowiedział na bieżące pytanie (aby nie przyjmować kolejnych kliknięć);

Metody publiczne:

- **void** setMode(bool isTraining) – ustawia tryb quizu;
- **void** startQuiz(const QString& topicName, int questionAmount = 10) – rozpoczyna quiz o podanym temacie i ilości pytań (domyślnie 10);
- **void** showCurrentQuestion() – metoda obsługująca pokazanie aktualnego pytania;

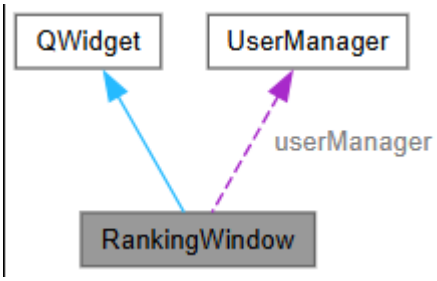
Sygnały:

- **void** backToMainMenuRequested() – emitowany, gdy użytkownik chce wyjść z okna i wrócić do ekranu MainMenu;
- **void** quizCompleted(const QString& topicName, int score, int correctAnswers) – emitowany, gdy metoda QuizManager::hasNextQuestion() zwróci fałsz; przekazuje odpowiednie parametry do wyświetlenia na ekranie końcowym;

Sloty / Metody prywatne:

- void on_quitButton_clicked() – obsługuje naciśnięcie przycisku ‘X’, emituje backToMainMenuRequested();
- void handleAnswerTimeout() – wywołuje on_confirmButton_clicked();
- void on_confirmButton_clicked() – obsługuje kliknięcie przycisku „Potwierdź”: wywołuje checkAnswerAndColorize(), zatrzymuje timer;
- void checkAnswerAndColorize() – sprawdza odpowiedź i zaznacza poprawną na zielono, a niepoprawną na czerwono (w przypadku gdy skończy się czas w trybie rankingowym i żadna odpowiedź nie została wybrana, zaznacza wszystkie błędne odpowiedzi na czerwono);
- void updateTimerDisplay() – reaguje na sygnał wysyłany przez timer (pole) co sekundę, by zaktualizować co on pokazuje (obiekt na interfejsie);

3.5. RankingWindow



Klasa reprezentująca okno wyświetlania rankingu użytkowników. Dziedziczy po QWidget; pokazuje listę użytkowników posortowaną według punktów.

Konstruktor:

explicit RankingWindow(UserManager* userManager, QWidget *parent = nullptr) – inicjalizuje GUI rankingu, pobiera listę użytkowników z userManager, sortuje je (operator< w User) i wyświetla w QTableWidget.

Pola:

- **Ui::RankingWindow** *ui – wskaźnik do wygenerowanego interfejsu (np. tabela rankingu, przycisk powrotu);
- **UserManager** *userManager – wskaźnik do menedżera użytkowników, aby pobrać listę i ewentualnie zaktualizować wyniki;

Sygnały:

- backToMainWindow() – emitowany, gdy użytkownik chce wrócić do ekranu głównego;

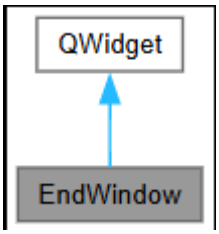
Metody publiczne:

- **void** updateRanking() – czyta plik *users.json* i aktualizuje ranking w tabeli;
- **void** setPodiumLabels() – aktualizuje obiekty QLabel na grafice podium na bazie pierwszych 3 pozycji w tabeli;

Sloty / Metody prywatne:

- **void** on_pushButton_clicked() – obsługuje kliknięcie przycisku „↩”; emituje backToMainWindow();

3.6. EndWindow



Klasa reprezentująca okno podsumowania po zakończeniu quizu. Dziedziczy po QWidget; pokazuje wynik i statystyki.

Konstruktor:

explicit EndWindow(std::shared_ptr<User> loggedInUser, QWidget *parent = nullptr) – inicjalizuje GUI podsumowania, ustawia pola statystyczne.

Pola:

- **Ui::EndWindow** *ui – wskaźnik do wygenerowanego interfejsu (np. QLabel z wynikiem);
- **std::shared_ptr<User>** loggedInUser – wskaźnik do zalogowanego użytkownika, do aktualizacji jego statystyk globalnych (totalPoints, totalGames) i wyświetlenia jego nazwy (*username*) w statystykach końcowych;
- **QString** topicName – nazwa tematu quizu, używana w wyświetleniu;
- **int** score – uzyskany wynik punktowy w danym quizie;
- **int** correctAnswers – liczba poprawnych odpowiedzi w danym quizie;

Sygnały:

- **void** quizFinished(int score) – emitowany po kliknięciu przycisku „Zakończ”; przekazuje uzyskany wynik, by wrócić do menu i zaktualizować statystyki zalogowanego użytwkownika.

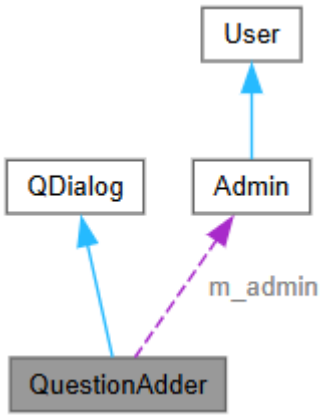
Metody publiczne:

- **void** setResults(const QString& topicName, int score, int correctAnswers) – ustawia QLabel z informacjami końcowymi;

Metody prywatne:

- **void** on_endButton_clicked() – obsługuje przycisk “Zakończ”, emituje quizFinished(score);

3.7. QuestionAdder



Klasa reprezentująca okno/dialog do dodawania nowych pytań do bazy. Dziedziczy po QDialog; pozwala administratorowi na wprowadzenie tekstu pytania, opcji odpowiedzi i zapisanie do pliku JSON.

Konstruktor:

explicit QuestionAdder(const Admin& admin, QWidget *parent = nullptr) – inicjalizuje okno, ustawia referencję do administratora (m_admin), wypełnia listę tematów (wywołując fillTopics()).

Pola:

- **Ui::QuestionAdder *ui** – wskaźnik do wygenerowanego interfejsu;
- **Admin** m_admin – kopia lub referencja do administratora; służy do weryfikacji uprawnień przed zapisaniem;

Sloty / Metody prywatne:

- **void** on_buttonBox_accepted() – slot wywoływany po kliknięciu „OK” w QDialogButtonBox; waliduje dane (np. czy jest co najmniej 2 opcje, czy indeks poprawnej w zakresie), tworzy TextQuestion i wywołuje QuestionEditor::addQuestion(m_admin, question, source); w razie sukcesu zamyka dialog lub czyści pola; w razie błędu wyświetla komunikat.
- **void** on_buttonBox_rejected() – slot wywoływany po kliknięciu „Anuluj”; zamyka dialog bez zmian.

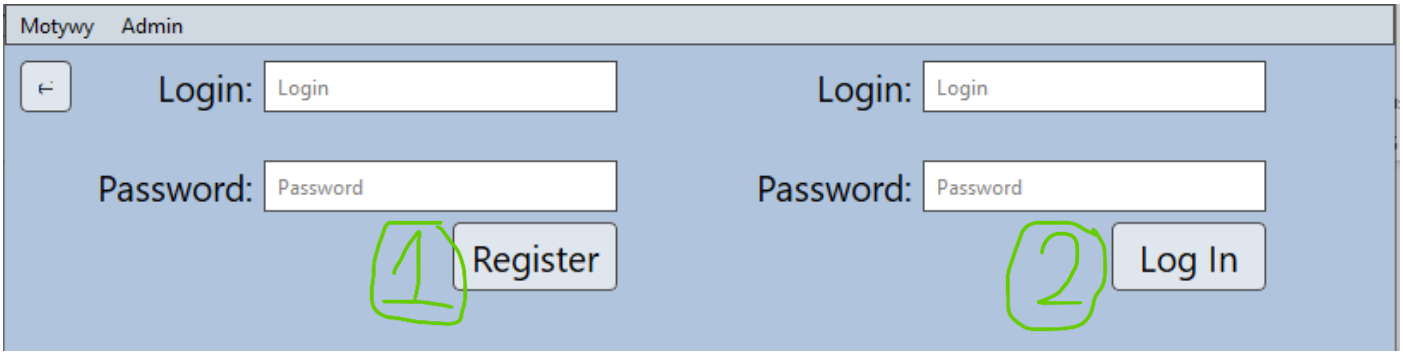
- **void** fillTopics() – uzupełnia QComboBox tematami dostępnymi w projekcie tak aby pytanie było dodane do właściwej bazy.

4. Instrukcja

Przycisk 1. przełącza okno na widok logowania. Użytkownik musi być zalogowany żeby zagrać, ale nie by zobaczyć ranking przyciskiem 2.. Przycisk 3. przełącza okno na menu główne jeśli jesteś zalogowany.

Opcją oznaczona przez 4. pozwala na zmianę motywu programu w każdej chwili.

Opcja zaznaczona nr 5. jest dostępna tylko dla Admina.

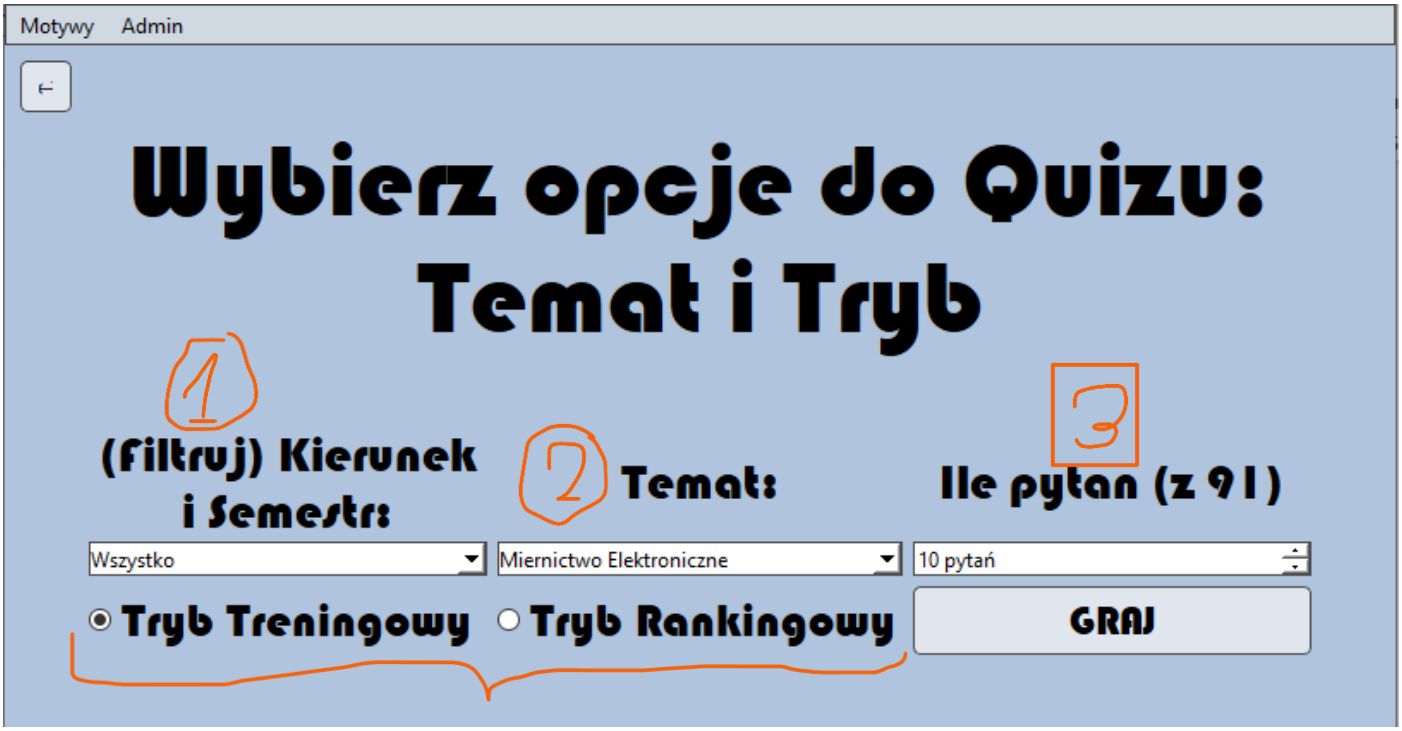


Na początku nie ma żadnych kont, więc trzeba pierwsze założyć. Będzie ono automatycznie administratorem.

Wybieramy nazwę użytkownika i hasło (bez spacji) i wciskamy przycisk ‘Register’ (nr 1.).

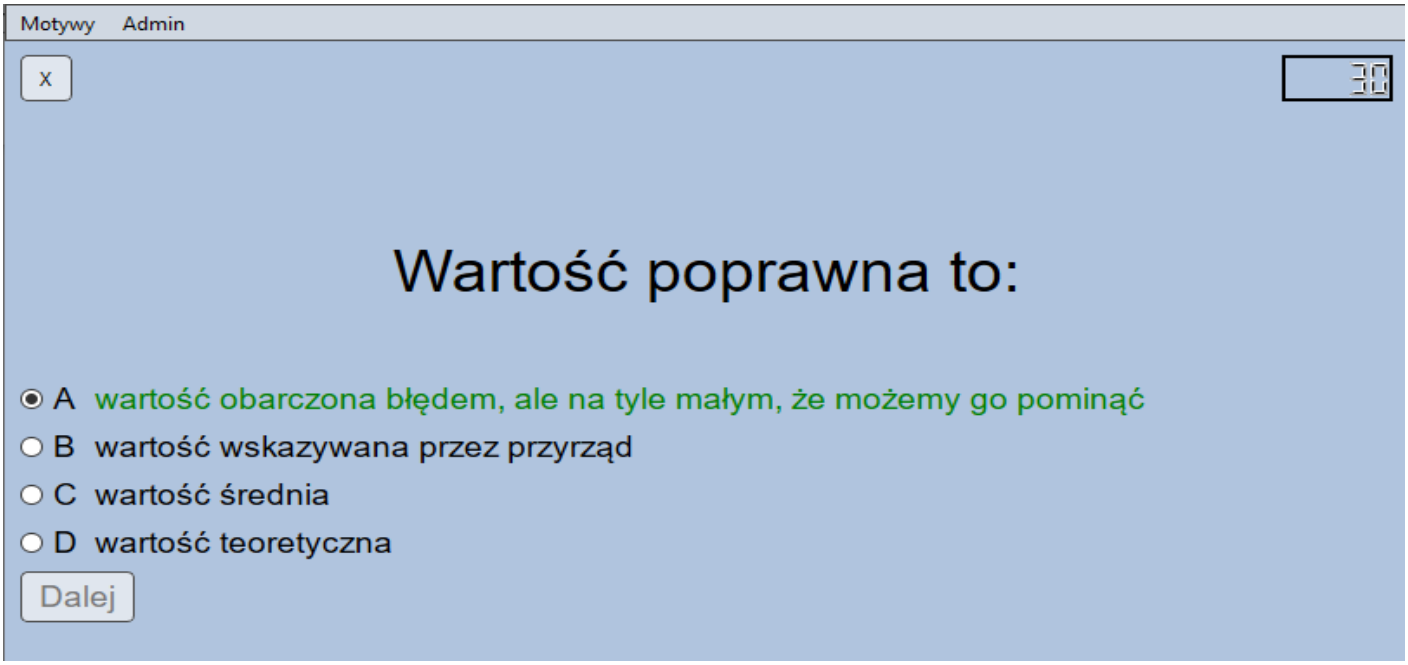
Należy pamiętać, że przy przyszłych rejestracjach nie może być znowu dokładnie taka sama nazwa użytkownika i nie będzie on administratorem.

Jeśli już masz konto i chcesz się zalogować – wpisz odpowiednie dane do pól po prawej stronie i zatwierdź klikając ‘Log In’

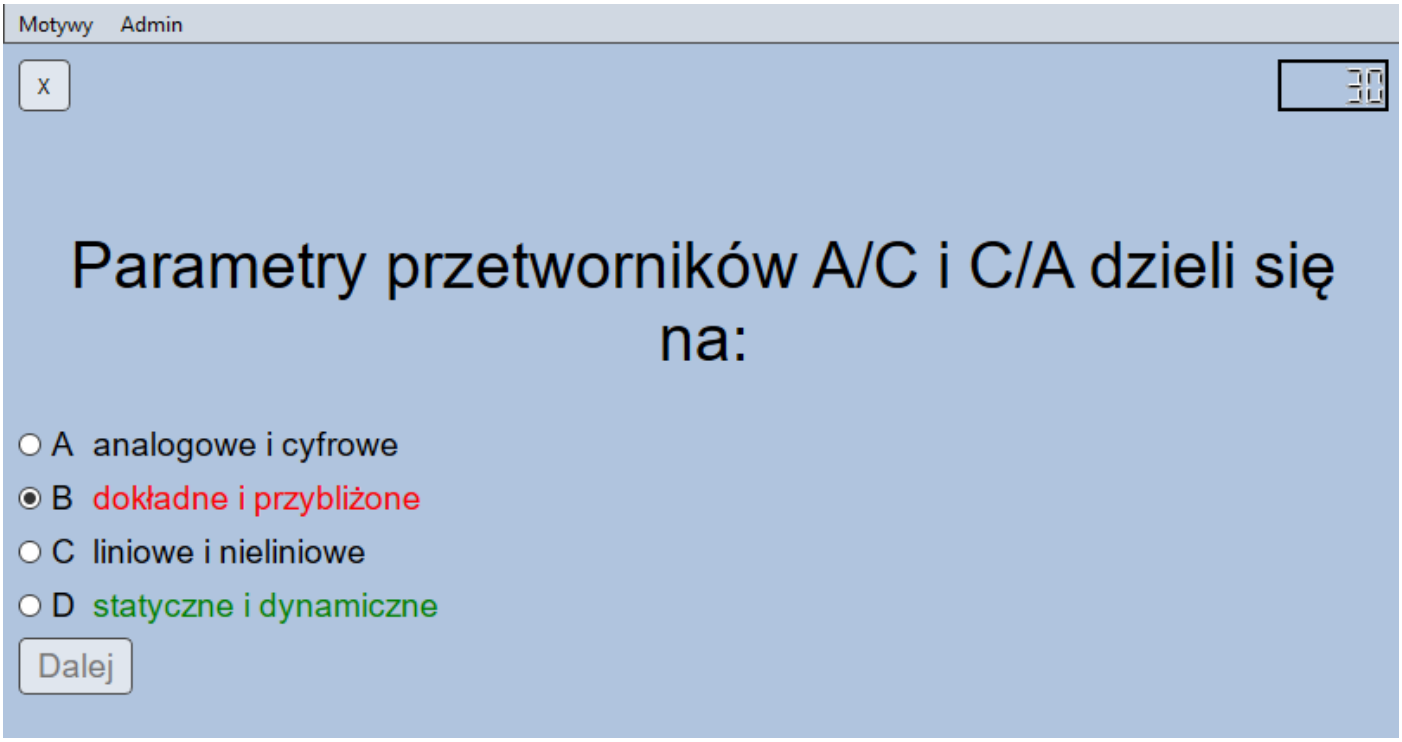


W oknie menu głównego możemy sobie wybrać temat (2.) nawet filtrując (1.) i ile pytań z niego chcemy (3.).

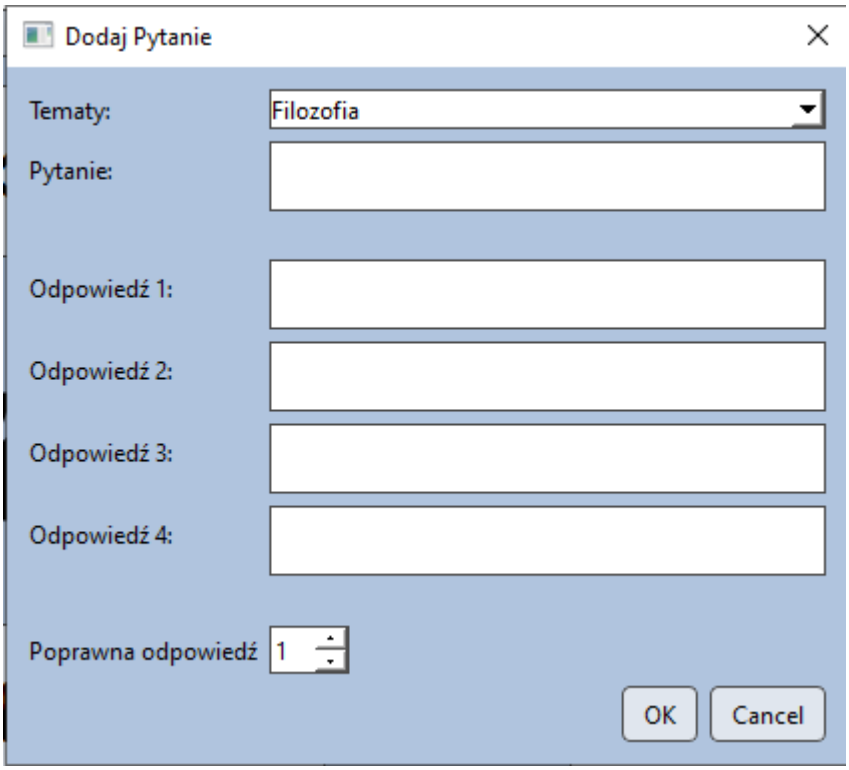
Domyślnie wybierany jest tryb treningowy, czyli taki w którym nie ma presji czasu i nie ma punktów.



Jeśli zaznaczono dobrą odpowiedź – podświetli się na zielono.



W przypadku błędnej odpowiedzi, jest ona zaznaczana na czerwono i pokazuje zielonym kolorem która była poprawna.



Jako administrator można dodać pytanie do dostępnego tematu.

Trzeba uzupełnić wszystkie pola i mieć pewność, że podany numer prawidłowej odpowiedzi jest poprawny.