|  |  |
| --- | --- |
| **Politechnika Świętokrzyska**  Wydział Elektrotechniki, Automatyki i Informatyki | |
| **Modelowanie i analiza systemów informatycznych** – Projekt | |
| **Temat:** | **Opracowanie:** |
| Zmiana unitermu poziomej operacji | Mateusz Misiak (91297) 1ID25A |
| eliminowania unitermów na pionową | **Data:** |
| operację eliminowania unitermów. | 19.05.2025 |

Spis treści

[1. Wstęp 4](#_Toc198547088)

[2. Opis zagadnienia 4](#_Toc198547089)

[2. Technologie użyte w projekcie 7](#_Toc198547090)

[2.1 Qt i C++ 7](#_Toc198547091)

[2.2 Struktura projektu 7](#_Toc198547092)

[2.3 Baza danych 7](#_Toc198547093)

[2.4 Kompilacja i uruchomienie 8](#_Toc198547094)

[3. Założenia projektu 8](#_Toc198547095)

[3.1 Założenia funkcjonalne 8](#_Toc198547096)

[3.2 Założenia interfejsowe 8](#_Toc198547097)

[3.3 Założenia techniczne 8](#_Toc198547098)

[4. Implementacja aplikacji 9](#_Toc198547099)

[4.1. Główna logika aplikacji – plik *main.cpp* 9](#_Toc198547100)

[4.1.1. Inicjalizacja aplikacji 9](#_Toc198547101)

[4.1.2. Budowa interfejsu użytkownika 9](#_Toc198547102)

[4.1.3. Obsługa interakcji użytkownika 10](#_Toc198547103)

[4.1.4. Obsługa transformacji unitermów 10](#_Toc198547104)

[4.1.5. Zapis danych do bazy 10](#_Toc198547105)

[4.1.6. Przegląd danych 11](#_Toc198547106)

[4.1.7. Rozmiar i uruchomienie aplikacji 11](#_Toc198547107)

[4.2. Komponent rysujący pionowy uniterm – pliki *verticalwidget.cpp* i *verticalwidget.h* 11](#_Toc198547108)

[4.2.1. Deklaracja klasy – *verticalwidget.h* 11](#_Toc198547109)

[4.2.2. Konstruktor – *verticalwidget.cpp* 11](#_Toc198547110)

[4.2.3. Ustawianie danych – metoda *setLabels(...)* 12](#_Toc198547111)

[4.2.4. Metoda *paintEvent(...)* – rysowanie widżetu 12](#_Toc198547112)

[4.3. Komponent rysujący poziomy uniterm – pliki *horizontalwidget.cpp* i *horizontalwidget.h* 13](#_Toc198547113)

[4.3.1. Deklaracja klasy – *horizontalwidget.h* 13](#_Toc198547114)

[4.3.2. Konstruktor – horizontalwidget.cpp 14](#_Toc198547115)

[4.3.3. Metoda *setLabels(...)* 14](#_Toc198547116)

[4.3.4. Metoda *paintEvent(...)* 14](#_Toc198547117)

[4.4. Komponenty transformacji unitermu – *ASwapWidget* i *BSwapWidget* 15](#_Toc198547118)

[4.4.1. Konstruktor 15](#_Toc198547119)

[4.4.2. Metoda *setLabels(...)* 16](#_Toc198547120)

[4.4.3. Rysowanie poziomego unitermu (część wspólna) 17](#_Toc198547121)

[4.4.4. Rysowanie pionowej linii (różnice!) 17](#_Toc198547122)

[4.4.5. Rysowanie pionowego unitermu 17](#_Toc198547123)

[4.6. Pliki *globals.cpp* i *globals.h* 18](#_Toc198547124)

[4.7. Pliki *save.cpp* i *save.h* 18](#_Toc198547125)

[Konstruktor klasy 18](#_Toc198547126)

[Obsługa przycisków 18](#_Toc198547127)

[4.8. Pliki *database.cpp* i *database.h* 19](#_Toc198547128)

[Inicjalizacja bazy danych 19](#_Toc198547129)

[Zapis danych 19](#_Toc198547130)

[Przegląd danych – *DatabaseViewer* 19](#_Toc198547131)

[5. Diagramy systemu 20](#_Toc198547132)

[5.1 Diagramy systemu – diagram przypadków użycia 20](#_Toc198547133)

[5.2. Diagram klas 21](#_Toc198547134)

[5.2.1. Klasa sterująca (kontroler główny) 21](#_Toc198547135)

[5.2.2. Komponenty wizualne (widżety graficzne) 21](#_Toc198547136)

[5.2.3. Klasy obsługi zapisu i bazy danych 21](#_Toc198547137)

[5.2.4. Klasa globalnych danych 22](#_Toc198547138)

[5.3. Diagram aktywności 24](#_Toc198547139)

[5.3.1. Etap 1: Uruchomienie i przygotowanie aplikacji 24](#_Toc198547140)

[5.3.2. Etap 2: Wprowadzanie danych 24](#_Toc198547141)

[5.3.3. Etap 3: Operacja zamiany 24](#_Toc198547142)

[5.3.4. Etap 4: Zapis i przegląd danych 25](#_Toc198547143)

[5.4. Diagram sekwencji 27](#_Toc198547144)

[5.4.1. Uruchomienie aplikacji 27](#_Toc198547145)

[5.4.2. Wprowadzanie danych 27](#_Toc198547146)

[5.4.3. Zamiana unitermów 27](#_Toc198547147)

[5.4.4. Zapis danych do bazy 27](#_Toc198547148)

[5.4.5. Przegląd zapisanych danych 27](#_Toc198547149)

[5.5. Diagram warstw systemu 30](#_Toc198547150)

[5.5.1. Warstwa prezentacji (UI) 30](#_Toc198547151)

[5.5.2. Warstwa logiki aplikacji 30](#_Toc198547152)

[5.5.3. Warstwa dostępu do danych 30](#_Toc198547153)

[5.5.4. Warstwa bazy danych 31](#_Toc198547154)

[5.6. Diagram komponentów 33](#_Toc198547155)

[5.6.1. MainWindow Controller 33](#_Toc198547156)

[5.6.2. Widgets 33](#_Toc198547157)

[5.6.3. Dialog 33](#_Toc198547158)

[5.6.4. Qt Framework 33](#_Toc198547159)

[6. Struktura bazy danych 35](#_Toc198547160)

[7. Zrzuty ekranu z działania aplikacji 35](#_Toc198547161)

[7.1. Ekran główny aplikacji 35](#_Toc198547162)

[7.2. Okno dialogowe – wprowadzanie danych 36](#_Toc198547163)

[7.3. Graficzna prezentacja unitermu poziomego 37](#_Toc198547164)

[7.4. Równoczesna prezentacja unitermów poziomego i pionowego 38](#_Toc198547165)

[7.5. Okno dialogowe – wybór sposobu zamiany unitermu 39](#_Toc198547166)

[7.6. Efekt zamiany unitermu – przypadek „Zamień za A” 40](#_Toc198547167)

[7.7. Efekt zamiany unitermu – przypadek „Zamień za B” 41](#_Toc198547168)

[7.8. Okno zapisu danych – formularz metadanych 42](#_Toc198547169)

[7.9. Podgląd zapisanych danych – widok bazy 43](#_Toc198547170)

[8. Wnioski i podsumowanie 43](#_Toc198547171)

# 1. Wstęp

Niniejsze sprawozdanie dotyczy realizacji projektu informatycznego z zakresu modelowania i analizy systemów, którego przedmiotem było stworzenie aplikacji graficznej umożliwiającej wizualne przekształcanie struktur algebraicznych zwanych unitermami. Zadanie polegało na zaprogramowaniu mechanizmu zamiany reprezentacji unitermów z formy liniowej (poziomej) na strukturę pionową, odzwierciedlającą operację eliminowania w alternatywnym układzie.

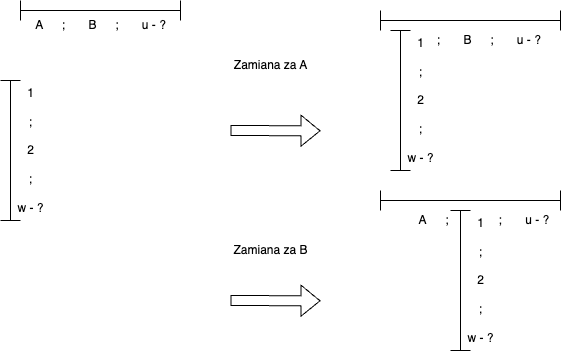
# 2. Opis zagadnienia

Celem niniejszego projektu było opracowanie oraz zaimplementowanie programu komputerowego, który realizuje proces przekształcenia unitermu z poziomej operacji eliminowania unitermów na postać pionową. Projekt wpisuje się w zagadnienia teoretycznej i stosowanej informatyki, a w szczególności w tematykę algebry algorytmów i ich graficznej reprezentacji.

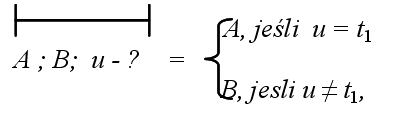
Unitermy stanowią elementarną jednostkę opisu algorytmów w podejściu algebraicznym, a operacje na nich — takie jak sekwencjonowanie, eliminowanie czy zrównoleglenie — pozwalają odwzorować strukturę i logikę działania algorytmu. W ramach projektu skupiono się na dwóch typach eliminowania unitermów:

* **poziomym**, w którym dane są reprezentowane i przetwarzane w jednej linii,
* **pionowym**, w którym struktura logiczna algorytmu jest przedstawiona w układzie kolumnowym.

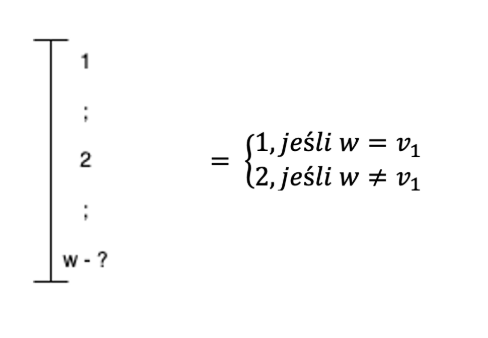
Przedstawiony poniżej diagram prezentuje graﬁczną reprezentację przebiegu tej operacji.



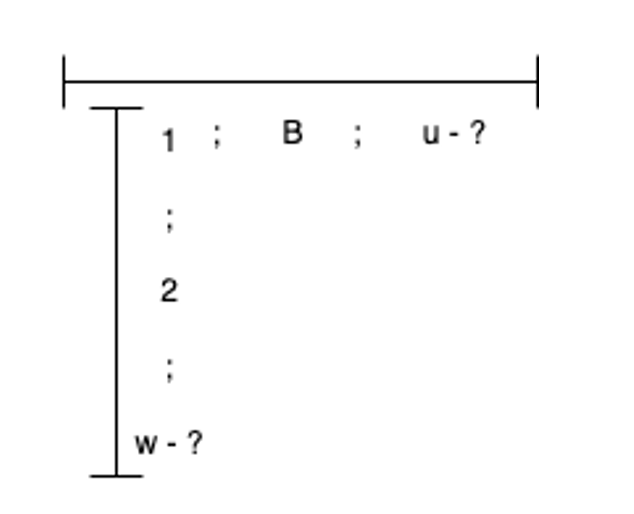
*Ilustracja graficzna zamiany unitermów*

**

gdzie *t1* – wartość unitermu warunkowego *u*.

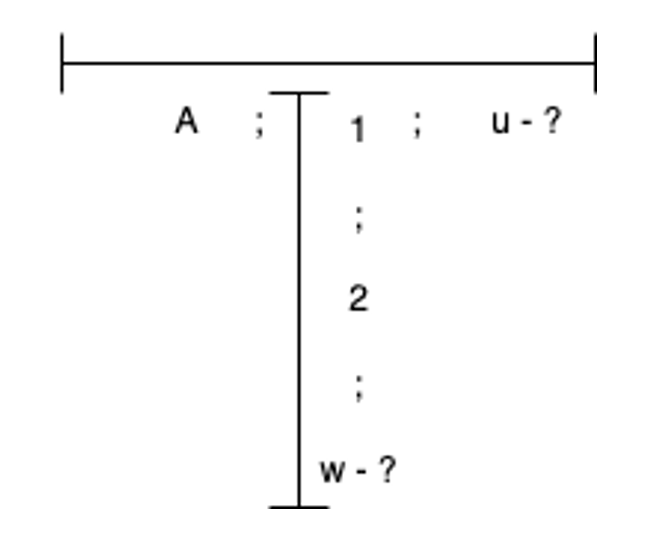


gdzie *v1* – wartość unitermu warunkowego *w*.



gdzie *v1* – wartość unitermu warunkowego *w*.

gdzie *t1* – wartość unitermu warunkowego *u*.



gdzie *v1* – wartość unitermu warunkowego *w*.

gdzie *t1* – wartość unitermu warunkowego *u*.

# 2. Technologie użyte w projekcie

Do realizacji projektu wykorzystano framework **Qt 6** w połączeniu z językiem programowania **C++**. Wybór tej technologii został podyktowany koniecznością stworzenia aplikacji graficznej typu desktop, która umożliwia intuicyjną obsługę oraz dynamiczną prezentację danych w postaci graficznej — zarówno poziomej, jak i pionowej.

## 2.1 Qt i C++

Qt to nowoczesny framework służący do tworzenia aplikacji wieloplatformowych z graficznym interfejsem użytkownika (GUI). W ramach projektu użyto modułu *QtWidgets*, który umożliwia tworzenie klasycznych komponentów okienkowych takich jak przyciski, pola tekstowe, ramki, widżety oraz obsługę zdarzeń użytkownika.

Język C++ zapewnił wydajność oraz pełną kontrolę nad logiką programu. Dzięki klasom dziedziczącym po *QWidget*, możliwe było samodzielne zaprojektowanie i narysowanie komponentów reprezentujących unitermy oraz operacje na nich.

## 2.2 Struktura projektu

Projekt został podzielony na następujące główne komponenty:

* **main.cpp** – odpowiada za utworzenie okna głównego i połączenie wszystkich elementów interfejsu.
* **VerticalWidget** i **HorizontalWidget** – klasy odpowiedzialne za rysowanie unitermów w układzie pionowym i poziomym.
* **ASwapWidget** i **BSwapWidget** – klasy odpowiedzialne za wizualizację operacji zamiany unitermów.
* **save.cpp / save.h** – komponent do wprowadzania i zapisywania nazw oraz opisów danych.
* **database.cpp / database.h** – klasa do obsługi lokalnej bazy danych SQLite.
* **globals.h / .cpp** – przechowuje zmienne globalne związane z wartościami unitermów.

## 2.3 Baza danych

Dane wprowadzone przez użytkownika mogą być zapisywane w lokalnej bazie danych **SQLite** pod nazwą *moja\_baza.db*. Każdy rekord zawiera nazwę, opis oraz wszystkie wartości związane z konkretnym przypadkiem zamiany unitermów. Obsługa bazy została zrealizowana przy użyciu klasy *QSqlDatabase* oraz mechanizmów *QSqlQuery*.

## 2.4 Kompilacja i uruchomienie

Projekt można skompilować za pomocą **Qt Creator** lub narzędzi wiersza poleceń (*qmake, make*). Do uruchomienia aplikacji wymagane jest środowisko Qt 6 oraz biblioteki *QtSql* i QtWidgets.

# 3. Założenia projektu

Celem projektowym było stworzenie aplikacji umożliwiającej użytkownikowi wizualizację i transformację unitermu z poziomej operacji eliminowania na postać pionową, zgodnie z rozszerzoną algebrą algorytmów. Aby to osiągnąć, przyjęto następujące założenia funkcjonalne i techniczne:

## 3.1 Założenia funkcjonalne

* Aplikacja powinna umożliwiać **wprowadzanie danych wejściowych** przez użytkownika, w tym wartości elementów unitermów.
* Dane mogą być reprezentowane w dwóch formach:
  + **poziomej** (np. A ; B ; u - ?),
  + **pionowej** (np. 1 ; 2 ; w - ?).
* Użytkownik powinien mieć możliwość dokonania **zamiany struktury unitermu** – z poziomej na pionową – za pomocą przycisku „Zamień”.
* Przed dokonaniem zamiany aplikacja zapyta, **czy zamiana ma dotyczyć symbolu A czy B**, co pozwala na dwa różne schematy transformacji.
* Aplikacja powinna oferować możliwość:
  + **zapisania danych** (nazwa i opis) do bazy danych,
  + **przeglądania historii** zapisanych przypadków (widok tabeli),

## 3.2 Założenia interfejsowe

* Interfejs użytkownika powinien być **czytelny i intuicyjny**, a wszystkie komponenty graficzne mają mieć estetyczny wygląd.
* Aplikacja powinna dynamicznie **aktualizować widok** po każdej zmianie danych wejściowych.
* Zastosowane zostały kolory neutralne (np. białe tło) dla poprawy przejrzystości rysunków.

## 3.3 Założenia techniczne

* Aplikacja powinna być **samodzielna**, nie wymagająca dostępu do Internetu ani zewnętrznych usług.
* Wymagane środowisko: Qt 6, biblioteki *QtWidgets*, *QtSql*, QtGui.
* Baza danych *SQLite* powinna być **lokalna**, automatycznie tworzona przy pierwszym uruchomieniu.
* Struktura bazy danych powinna zawierać kolumny odpowiadające wszystkim zmiennym używanym w transformacji (val1, val2, valW, valA, valB, valU).

# 4. Implementacja aplikacji

## 4.1. Główna logika aplikacji – plik *main.cpp*

Plik *main.cpp* zawiera główny punkt wejścia aplikacji oraz implementację interfejsu użytkownika w całości zbudowanego przy użyciu biblioteki Qt6. To właśnie w tym miejscu tworzona jest struktura okna głównego, dodawane są wszystkie widżety rysujące oraz definiowane są interakcje – od przycisków wprowadzających dane po obsługę zapisu do bazy i transformacji unitermów.

### 4.1.1. Inicjalizacja aplikacji

Na początku tworzone jest środowisko aplikacji Qt:

QApplication app(argc, argv);

Następnie wyświetlana jest lokalizacja pliku bazy danych w terminalu (dla celów diagnostycznych):

Aplikacja próbuje zainicjować połączenie z lokalną bazą SQLite:

qDebug() << "Ścieżka do bazy danych:" << QDir::current().absoluteFilePath("moja\_baza.db");

if (!Database::initialize()) {

return -1;

}

Jeśli połączenie się nie powiedzie, aplikacja natychmiast się kończy.

### 4.1.2. Budowa interfejsu użytkownika

Tworzone jest główne okno (*QWidget*) z tytułem:

QWidget window;

window.setWindowTitle("MASI PROJEKT MATEUSZ MISIAK 1ID25A");

Użyty zostaje układ pionowy (*QVBoxLayout*) oraz ramka (*QFrame*) pełniąca rolę obszaru roboczego (canvasu). Ramka ta zawiera wszystkie graficzne komponenty (*VerticalWidget*, *HorizontalWidget*, *ASwapWidget*, *BSwapWidget*) służące do prezentacji unitermów.

Każdy z tych widżetów jest ukrywany domyślnie przy uruchomieniu:

horizontalDrawing->hide();

verticalDrawing->hide();

aSwapDrawing->hide();

bSwapDrawing->hide();

Interfejs zawiera również zestaw przycisków:

* **Wprowadź dane pionowo**
* **Wprowadź dane poziomo**
* **Zamień**
* **Zapisz**
* **Zobacz dane**

Dodawane są one poniżej obszaru rysowania.

### 4.1.3. Obsługa interakcji użytkownika

#### Wprowadzanie danych pionowych

Kliknięcie w przycisk „Wprowadź dane pionowo” powoduje otwarcie trzech okien dialogowych. Dane są zapisywane w zmiennych globalnych *val1*, *val2*, *valW*, a następnie przekazywane do *verticalDrawing*:

val1 = QInputDialog::getText(&window, "Dane", "Wprowadź wartość 1:");

...

verticalDrawing->setLabels(val1, val2, valW);

#### Wprowadzanie danych poziomych

Analogiczna obsługa zachodzi przy przycisku „Wprowadź dane poziomo”, gdzie dane trafiają do *valA*, *valB*, *valU* oraz widżetu *horizontalDrawing*.

### 4.1.4. Obsługa transformacji unitermów

Kliknięcie przycisku **Zamień** wywołuje *QMessageBox*, który pyta użytkownika, czy zamiana ma być wykonana za *A* czy za *B*.

W zależności od wyboru:

* Pokazywany jest *ASwapWidget* (transformacja przez A)
* Lub *BSwapWidget* (transformacja przez B)

Dane są przekazywane do odpowiedniego widżetu przy użyciu metody *setLabels(...)*:

aSwapDrawing->setLabels(val1, val2, valW, valA, valB, valU);

### 4.1.5. Zapis danych do bazy

Kliknięcie w przycisk **Zapisz** otwiera okno dialogowe (*save*) z dwoma polami: nazwa i opis. Po potwierdzeniu, dane te – wraz z aktualnymi wartościami *val1*, *val2*, *valW*, *valA*, *valB*, *valU* – są zapisywane do bazy SQLite:

if (Database::insertRecord(nazwa, opis, val1, val2, valW, valA, valB, valU)) {

QMessageBox::information(...);

}

### 4.1.6. Przegląd danych

Przycisk **Zobacz dane** uruchamia nowy dialog (*DatabaseViewer*), który wczytuje i prezentuje wszystkie rekordy z tabeli *zapisy* w formie tabelarycznej.

### 4.1.7. Rozmiar i uruchomienie aplikacji

Po zakończeniu konfiguracji, aplikacja ustawia rozmiar okna na *800x800* i je pokazuje:

window.resize(800, 800);

window.show();

return app.exec();

## 4.2. Komponent rysujący pionowy uniterm – pliki *verticalwidget.cpp* i *verticalwidget.h*

Komponent *VerticalWidget* odpowiada za graficzną reprezentację unitermu pionowego – jednej z form operacji eliminowania unitermów przedstawionej w projekcie. Ten widżet umożliwia dynamiczne wizualizowanie danych wprowadzonych przez użytkownika w kontekście operacji logiczno-algorytmicznej.

### 4.2.1. Deklaracja klasy – *verticalwidget.h*

**Opis:**

class VerticalWidget : public QWidget {

Q\_OBJECT

public:

explicit VerticalWidget(QWidget \*parent = nullptr);

void setLabels(const QString &a, const QString &b, const QString &c);

protected:

void paintEvent(QPaintEvent \*event) override;

private:

QStringList labels;

};

* Klasa dziedziczy po *QWidget*, co pozwala na jej bezpośrednie użycie w interfejsie graficznym.
* Metoda *setLabels* umożliwia ustawienie treści, które zostaną wyświetlone obok pionowej linii.
* Przesłonięta metoda *paintEvent* odpowiada za rysowanie widżetu.

### 4.2.2. Konstruktor – *verticalwidget.cpp*

**Opis:**  
Domyślne etykiety są inicjalizowane zgodnie z ogólną postacią logiczną unitermu: argument1, separator, argument2, separator, warunek. Użycie *QStringList* umożliwia łatwe iterowanie podczas rysowania.

VerticalWidget::VerticalWidget(QWidget \*parent) : QWidget(parent) {

// Domyślne etykiety

labels = { "1", ";", "2", ";", "w - ?" };

}

### 4.2.3. Ustawianie danych – metoda *setLabels(...)*

**Opis:**

void VerticalWidget::setLabels(const QString &a, const QString &b, const QString &c) {

labels = { a, ";", b, ";", c + " - ?" };

update(); // odświeżenie widoku

}

* Metoda przyjmuje trzy parametry (np. *"1"*, *"2"*, *"w"*) i formatuje je jako sekwencję tekstową.
* Dodawany jest znak *"- ?"* przy trzeciej wartości, co jest konwencją wskazującą na niewiadomą w warunku eliminacji.
* Wywołanie *update()* powoduje ponowne narysowanie komponentu, wyzwalając *paintEvent*.

### 4.2.4. Metoda *paintEvent(...)* – rysowanie widżetu

void VerticalWidget::paintEvent(QPaintEvent \*event) {

QPainter painter(this);

painter.setRenderHint(QPainter::Antialiasing);

Rysowanie rozpoczyna się od utworzenia obiektu *QPainter* z wygładzaniem krawędzi (*Antialiasing*).

#### Rysowanie pionowej linii

**Opis:**

int margin = 30;

int x = width() / 2;

painter.drawLine(x, margin, x, height() - margin); // pionowa linia

painter.drawLine(x - 20, margin, x + 20, margin); // poprzeczka górna

painter.drawLine(x - 20, height() - margin, x + 20, height() - margin); // dolna

* Linia rysowana jest w centrum widżetu – z górnym i dolnym ograniczeniem.
* Poprzeczki (znaczniki) na obu końcach linii poprawiają estetykę i czytelność.

#### Ustawienie czcionki

QFont font = painter.font();

font.setPointSize(30); // duża czcionka

painter.setFont(font);

Widoczne etykiety są renderowane dużą czcionką, co zwiększa ich czytelność w kontekście graficznej prezentacji algorytmu.

#### Rysowanie tekstu – etykiet

int baseY = height() / 2 - 60;

int spacing = 40;

for (int i = 0; i < labels.size(); ++i) {

int y = baseY + i \* spacing;

painter.drawText(x + 25, y, labels[i]);

}

**Opis:**

* Tekst (czyli elementy unitermu) rysowany jest w kolumnie po prawej stronie pionowej linii.
* *baseY* określa pozycję początkową, a *spacing* – odstęp między wierszami.
* Dynamiczne wykorzystanie *QStringList labels* sprawia, że komponent może łatwo zostać użyty do wyświetlenia różnych konfiguracji danych.

## 4.3. Komponent rysujący poziomy uniterm – pliki *horizontalwidget.cpp* i *horizontalwidget.h*

Komponent *HorizontalWidget* odpowiada za graficzną prezentację unitermu w jego klasycznej postaci poziomej. Przedstawia sekwencję składającą się z dwóch operandów oddzielonych średnikami oraz warunku *u*. Komponent ten służy jako punkt wyjścia dla dalszej transformacji na postać pionową w ramach projektu.

### 4.3.1. Deklaracja klasy – *horizontalwidget.h*

**Opis:**

class HorizontalWidget : public QWidget {

Q\_OBJECT

public:

explicit HorizontalWidget(QWidget \*parent = nullptr);

void setLabels(const QString &a, const QString &b, const QString &c);

protected:

void paintEvent(QPaintEvent \*event) override;

private:

QStringList labels;

};

* Klasa dziedziczy po *QWidget*.
* Przechowuje listę *labels*, która zawiera elementy do wyświetlenia: operand1, średnik, operand2, średnik, warunek *u - ?*.
* *setLabels* aktualizuje zawartość tej listy, a *paintEvent* odpowiada za wizualizację danych.

### 4.3.2. Konstruktor – horizontalwidget.cpp

**Opis:**

HorizontalWidget::HorizontalWidget(QWidget \*parent) : QWidget(parent) {

labels = { "A", ";", "B", ";", "u - ?" };

}

* Komponent inicjalizowany jest z domyślnymi danymi symbolizującymi strukturę *A ; B ; u - ?*.
* Domyślne wartości pomagają w prezentacji szkieletu operacji przed wprowadzeniem danych użytkownika.

### 4.3.3. Metoda *setLabels(...)*

**Opis:**

void HorizontalWidget::setLabels(const QString &a, const QString &b, const QString &c) {

labels = { a, ";", b, ";", c + " - ?" };

update();

}

* Metoda przyjmuje trzy tekstowe wartości: dwa operandy i warunek logiczny.
* Tworzona lista *QStringList* pozwala na elastyczne zarządzanie rysowanym tekstem.
* *update()* powoduje natychmiastowe odświeżenie widżetu i ponowne wywołanie *paintEvent*.

### 4.3.4. Metoda *paintEvent(...)*

void HorizontalWidget::paintEvent(QPaintEvent \*event) {

QPainter painter(this);

painter.setRenderHint(QPainter::Antialiasing);

**Opis:**

* Inicjalizacja rysowania z włączoną opcją wygładzania (*Antialiasing*), co zapewnia wysoką jakość linii i tekstów.

#### Rysowanie linii i znaczników

**Opis:**

int margin = 30;

int y = height() / 2;

painter.drawLine(margin, y, width() - margin, y);

painter.drawLine(margin, y - 20, margin, y + 20);

painter.drawLine(width() - margin, y - 20, width() - margin, y + 20);

* Główna pozioma linia symbolizuje sekwencję unitermu.
* Na obu końcach dodano znaczniki (poprzeczki), aby wizualnie domknąć linię.

#### Konfiguracja czcionki

QFont font = painter.font();

font.setPointSize(30);

painter.setFont(font);

**Opis:**

* Czcionka o dużym rozmiarze (30pt) gwarantuje czytelność nawet w przypadku długich etykiet.

#### Rysowanie tekstu nad linią

int baseX = width() / 2 - 160;

int spacing = 60;

for (int i = 0; i < labels.size(); ++i) {

int x = baseX + i \* spacing;

painter.drawText(x, y + 50, labels[i]);

}

**Opis:**

* *baseX* ustala początek rysowania tekstów (punkt odniesienia).
* Każda kolejna etykieta rysowana jest w odstępie 60 pikseli.
* Tekst wyświetlany jest **nad** linią (dokładnie 50 pikseli poniżej punktu linii poziomej).

## 4.4. Komponenty transformacji unitermu – *ASwapWidget* i *BSwapWidget*

Oba komponenty dziedziczą po *QWidget* i są odpowiedzialne za graficzną wizualizację transformacji unitermu z poziomego na pionowy. Różnią się jedynie **punktem podstawienia**:

* *ASwapWidget* — podstawienie odbywa się względem operandu *A*,
* *BSwapWidget* — podstawienie względem *B*.

### 4.4.1. Konstruktor

**ASwapWidget:**

ASwapWidget::ASwapWidget(QWidget \*parent) : QWidget(parent) {

labelA = "1";

labelB = "B";

labelC = "u";

verticalLabels = { ";", "2", ";", "w - ?" };

}

**BSwapWidget:**

BSwapWidget::BSwapWidget(QWidget \*parent) : QWidget(parent) {

labelA = "A";

labelB = "1";

labelC = "u";

verticalLabels = { ";", "2", ";", "w - ?" };

}

**Wyjaśnienie:**

* W obu konstruktorach ustawiane są domyślne wartości, które mają znaczenie tylko przy uruchomieniu komponentu bez danych od użytkownika.
* *labelA*, *labelB*, *labelC* – przechowują dane poziome.
* *verticalLabels* – lista tekstów do wyświetlenia pionowo pod miejscem podstawienia.

### 4.4.2. Metoda *setLabels(...)*

**ASwapWidget:**

void ASwapWidget::setLabels(const QString &a, const QString &b, const QString &c,

const QString &d, const QString &e, const QString &f) {

labelA = a; // operand podstawiający (np. "1")

labelB = e; // drugi operand (np. "B")

labelC = f; // warunek (np. "u")

verticalLabels = { ";", b, ";", c + " - ?" }; // np. [";", "2", ";", "w - ?"]

update();

}

**BSwapWidget:**

void BSwapWidget::setLabels(const QString &a, const QString &b, const QString &c,

const QString &d, const QString &e, const QString &f) {

labelA = d; // operand A (np. "A")

labelB = a; // operand podstawiający (np. "1")

labelC = f; // warunek (np. "u")

verticalLabels = { ";", b, ";", c + " - ?" }; // np. [";", "2", ";", "w - ?"]

update();

}

**Wyjaśnienie:**

* Główna różnica: w *ASwapWidget* operand *val1* (czyli *a*) zastępuje *A*, a w *BSwapWidget* – *val1* zastępuje *B*.
* *update()* powoduje ponowne wywołanie *paintEvent*, co skutkuje przerysowaniem widżetu.

### 4.4.3. Rysowanie poziomego unitermu (część wspólna)

int baseX = widthMid - 100;

int spacingX = 40;

painter.drawText(baseX + 0 \* spacingX , topY + 20, labelA);

painter.drawText(baseX + 1 \* spacingX, topY + 20, ";");

painter.drawText(baseX + 2 \* spacingX, topY + 20, labelB);

painter.drawText(baseX + 3 \* spacingX, topY + 20, ";");

painter.drawText(baseX + 4 \* spacingX, topY + 20, labelC + " - ?");

**Wyjaśnienie:**

* Wyświetlane są kolejno: operand A, separator, operand B, separator, warunek z sufiksem *" - ?"*.
* Umożliwia to odwzorowanie formy *A ; B ; u - ?*.

### 4.4.4. Rysowanie pionowej linii (różnice!)

**ASwapWidget:**

int verticalX = baseX - 10; // linia pod operandem A

**BSwapWidget:**

int verticalX = baseX + 50; // linia pod operandem B

**Wyjaśnienie:**

* Pozycja pionowej linii wskazuje, który operand (*A* czy *B*) jest „zastępowany” (transformowany).
* W *ASwapWidget* linia jest bardziej z lewej – pod *labelA*,
* W *BSwapWidget* linia przesunięta bardziej w prawo – pod *labelB*.

### 4.4.5. Rysowanie pionowego unitermu

int baseY = verticalTop + 30;

int spacingY = 35;

for (int i = 0; i < verticalLabels.size(); ++i) {

int y = baseY + i \* spacingY;

painter.drawText(verticalX + 15, y, verticalLabels[i]);

}

**Wyjaśnienie:**

* Pod pionową linią wypisywane są teksty symbolizujące uniterm pionowy po transformacji.
* Zawartość *verticalLabels* to: separator *;*, drugi operand (np. *2*), separator *;*, warunek *w - ?*.

## 4.6. Pliki *globals.cpp* i *globals.h*

Te pliki przechowują zmienne globalne – wartości wpisane przez użytkownika. Dzięki deklaracjom *extern* w pliku nagłówkowym *globals.h*, są one współdzielone między różnymi plikami źródłowymi:

extern QString val1;

extern QString val2;

extern QString valW;

extern QString valA;

extern QString valB;

extern QString valU;

Wartości są inicjalizowane w *globals.cpp*:

QString val1 = "1";

QString val2 = "2";

QString valW = "W";

QString valA = "A";

QString valB = "B";

QString valU = "U";

## 4.7. Pliki *save.cpp* i *save.h*

Klasa *save* definiuje własne okno dialogowe służące do zapisu danych do bazy danych. Dziedziczy po *QDialog* i zawiera dwa pola wejściowe: jedno na nazwę, drugie na opis operacji.

### Konstruktor klasy

Tworzy komponenty GUI: dwa pola (jednoliniowe i wieloliniowe) oraz dwa przyciski (*Zapisz*, *Anuluj*):

nameEdit = new QLineEdit();

descriptionEdit = new QTextEdit();

Układ przycisków tworzony jest z wykorzystaniem *QHBoxLayout* i dodawany do głównego układu *QVBoxLayout*.

### Obsługa przycisków

Przyciski są połączone z metodami *accept()* i *reject()*:

connect(saveButton, &QPushButton::clicked, this, &QDialog::accept);

connect(cancelButton, &QPushButton::clicked, this, &QDialog::reject);

#### Dostęp do danych

Dane wpisane przez użytkownika można pobrać za pomocą metod:

QString getName() const;

QString getDescription() const;x

## 4.8. Pliki *database.cpp* i *database.h*

Moduł *Database* odpowiada za inicjalizację bazy SQLite oraz wykonywanie operacji zapisu i odczytu danych.

### Inicjalizacja bazy danych

Metoda *initialize()* tworzy połączenie do lokalnego pliku SQLite *moja\_baza.db* i w razie potrzeby – tworzy tabelę *zapisy*:

db.setDatabaseName("moja\_baza.db");

query.exec(R"(

CREATE TABLE IF NOT EXISTS zapisy (

id INTEGER PRIMARY KEY AUTOINCREMENT,

nazwa TEXT NOT NULL,

opis TEXT,

val1 TEXT,

val2 TEXT,

valW TEXT,

valA TEXT,

valB TEXT,

valU TEXT

)

)");

### Zapis danych

Dane z formularza oraz z pól globalnych są przekazywane do funkcji *insertRecord*:

query.prepare("INSERT INTO zapisy (...) VALUES (...)");

query.bindValue(":val1", val1);

...

query.exec();

Zwracana jest informacja o powodzeniu operacji (true/false), dzięki czemu *main.cpp* może pokazać odpowiedni komunikat użytkownikowi.

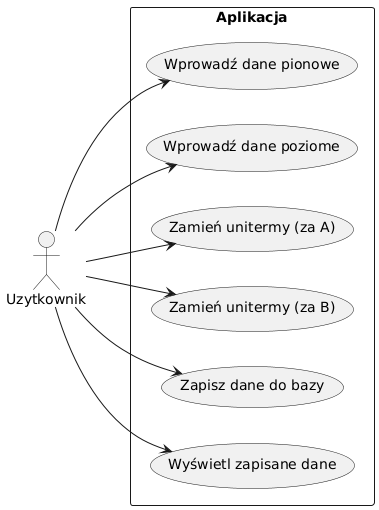
### Przegląd danych – *DatabaseViewer*

Klasa *DatabaseViewer* to proste okno dialogowe (*QDialog*) z tabelą (*QTableWidget*), w której wyświetlane są wszystkie rekordy z bazy.

* Metoda *loadData()* wykonuje zapytanie *SELECT \* FROM zapisy* i wypełnia tabelę.
* Kolumny są etykietowane i dopasowane do pól zdefiniowanych w tabeli.

# 5. Diagramy systemu

## 5.1 Diagramy systemu – diagram przypadków użycia



Powyższy diagram stanowi **diagram przypadków użycia (Use Case Diagram)** przedstawiający podstawowe interakcje użytkownika z aplikacją.

Głównym aktorem jest **Użytkownik**, który może korzystać z funkcjonalności dostępnych w ramach systemu oznaczonego jako **Aplikacja**. Poszczególne przypadki użycia zostały zobrazowane jako elipsy i opisują dostępne działania:

1. **Wprowadź dane pionowe** – użytkownik może podać dane wejściowe do struktury unitermu w postaci pionowej.
2. **Wprowadź dane poziome** – umożliwia wprowadzenie danych do unitermu poziomego.
3. **Zamień unitermy (za A)** – pozwala na przekształcenie unitermów poprzez podstawienie pionowej formy w miejsce wartości A.
4. **Zamień unitermy (za B)** – alternatywna opcja zamiany, z podstawieniem w miejsce B.
5. **Zapisz dane do bazy** – użytkownik może zarchiwizować aktualny stan danych w lokalnej bazie.
6. **Wyświetl zapisane dane** – umożliwia przeglądanie wcześniej zapisanych rekordów.

## 5.2. Diagram klas

Diagram klas przedstawia **strukturę statyczną aplikacji** poprzez odwzorowanie klas, ich atrybutów, metod oraz zależności między nimi. Model ten ilustruje organizację systemu z perspektywy programistycznej, zgodnie z paradygmatem obiektowym (OOP), i ułatwia analizę ról poszczególnych komponentów w systemie.

Diagram obejmuje cztery główne grupy klas:

### 5.2.1. Klasa sterująca (kontroler główny)

* ***MainWindow***
  + Główna klasa okna aplikacji Qt (*QWidget*), odpowiedzialna za:
    - tworzenie i układanie elementów GUI,
    - obsługę interakcji użytkownika,
    - zarządzanie widżetami graficznymi i przepływem danych między nimi.
  + Klasa zawiera wskaźniki do przycisków (*QPushButton*), układów (*QVBoxLayout*) oraz instancji widżetów (*VerticalWidget*, *HorizontalWidget*, *ASwapWidget*, *BSwapWidget*).
  + Obsługuje zdarzenia przycisków i łączy komponenty logiczne z graficznymi.

### 5.2.2. Komponenty wizualne (widżety graficzne)

Te klasy odpowiadają za graficzne przedstawienie danych (unitermów):

* ***VerticalWidget***
* ***HorizontalWidget***
* ***ASwapWidget***
* ***BSwapWidget***

Każda z tych klas:

* dziedziczy po *QWidget*,
* implementuje metodę *paintEvent(QPaintEvent \*event)* do własnoręcznego rysowania zawartości (linii, tekstu, strzałek, separatorów),
* zawiera metodę *setLabels(...)*, która przyjmuje wartości wejściowe i aktualizuje zawartość graficzną,
* przechowuje dane etykiet w strukturach typu *QString* lub *QStringList*.

Widżety *ASwapWidget* i *BSwapWidget* pełnią dodatkową funkcję — prezentują **wynik operacji zamiany unitermu** za A lub B, zgodnie z wyborem użytkownika.

### 5.2.3. Klasy obsługi zapisu i bazy danych

* ***save***  
  Klasa dialogowa (*QDialog*), pozwalająca użytkownikowi wprowadzić nazwę i opis dla transformacji przed zapisaniem ich do bazy danych.
  + Atrybuty: *QLineEdit \*nameEdit*, *QTextEdit \*descriptionEdit*
  + Metody: *getName()*, *getDescription()* – umożliwiają dostęp do wprowadzonych danych.
* ***Database***  
  Klasa zawierająca statyczne metody do obsługi bazy danych SQLite:
  + *initialize()* – tworzy i otwiera bazę (jeśli nie istnieje),
  + *insertRecord(...)* – zapisuje dane transformacji,
  + Współpracuje z tabelą *zapisy*.
* ***DatabaseViewer***  
  Komponent okienkowy (*QDialog*) umożliwiający przegląd zapisanych rekordów z bazy w formie tabeli (*QTableWidget*), ładowanych przez metodę *loadData()*.

### 5.2.4. Klasa globalnych danych

* ***globals.h* / *globals.cpp***  
  Deklaruje i przechowuje tymczasowe dane wejściowe w formie zmiennych globalnych:
  + *val1*, *val2*, *valW*, *valA*, *valB*, *valU*  
    Dzięki temu wartości wprowadzone przez użytkownika mogą być udostępniane między komponentami bez bezpośredniego przekazywania przez konstruktor czy metodę.

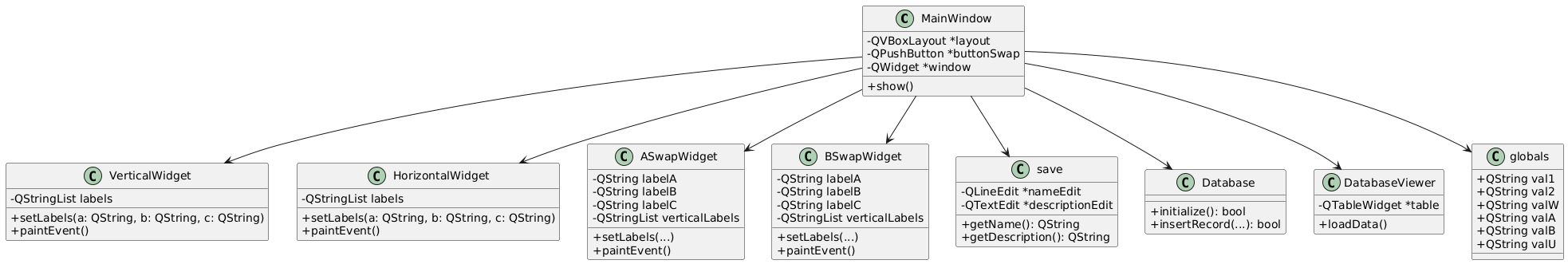
#### Relacje między klasami

* *MainWindow* **agreguje** wszystkie komponenty graficzne i logikę zapisu – posiada wskaźniki do instancji widżetów i klas dialogowych.
* *MainWindow* **komunikuje się** z klasą *Database*, wywołując jej metody statyczne.
* *ASwapWidget* i *BSwapWidget* **dziedziczą** interfejs i funkcjonalność *QWidget*, ale mają własne wersje *paintEvent()*.
* Wszystkie widżety działają niezależnie, co ułatwia ich izolowane testowanie i ponowne użycie.

#### Zastosowanie wzorców projektowych

Diagram klas ujawnia zastosowanie następujących wzorców projektowych:

* **MVC (Model-View-Controller) – w wersji uproszczonej:**
  + *MainWindow* – Controller
  + Widżety – View
  + *globals*, *Database* – Model (dane i logika zapisu)
* **Singleton (pośrednio)** – klasa *Database* używana jest jako statyczna instancja dostępu do bazy.
* **Dialog Object** – *save* i *DatabaseViewer* to osobne, niezależne komponenty dialogowe z odpowiedzialnością za zbieranie lub wyświetlanie danych.



## 5.3. Diagram aktywności

Diagram aktywności prezentuje **przepływ sterowania** w aplikacji z punktu widzenia użytkownika i systemu, uwzględniając kolejność operacji, warunki decyzyjne oraz interakcje między warstwami. Reprezentuje on pełny cykl życia jednej sesji użytkownika — od momentu uruchomienia programu do zapisu i przeglądu danych.

Diagram podzielony jest logicznie na trzy główne fazy:

1. **Wprowadzanie danych**
2. **Operacja zamiany unitermów**
3. **Zapis i przegląd danych**

Każdy etap odpowiada konkretnemu przypadkowi użycia i obejmuje zarówno interakcje użytkownika z interfejsem, jak i reakcje aplikacji.

### 5.3.1. Etap 1: Uruchomienie i przygotowanie aplikacji

* **Start** – aplikacja zostaje uruchomiona przez użytkownika.
* **Inicjalizacja bazy danych** (*initialize()*) – następuje połączenie z lokalną bazą SQLite, ewentualnie jej utworzenie, jeśli jeszcze nie istnieje.
* **Wyświetlenie interfejsu użytkownika** – na ekranie pojawia się główne okno z ramką graficzną, przyciskami oraz możliwością interakcji.

### 5.3.2. Etap 2: Wprowadzanie danych

* Użytkownik wybiera opcję „**Wprowadź dane poziomo**” lub „**Wprowadź dane pionowo**”.
* Aplikacja otwiera kolejne **okna dialogowe** (*QInputDialog*) do wprowadzenia wartości:
  + dla pionowego: *val1*, *val2*, *valW*
  + dla poziomego: *valA*, *valB*, *valU*
* Dane są zapisywane w **globalnych zmiennych**, które służą jako tymczasowe buforowanie wartości dla dalszych operacji.
* Następuje **aktualizacja widżetów graficznych** (*VerticalWidget* / *HorizontalWidget*) – wartości są wizualizowane w ramce aplikacji.

### 5.3.3. Etap 3: Operacja zamiany

* Użytkownik klika przycisk **„Zamień”**, co powoduje:
  + Wyświetlenie **dialogu wyboru** wariantu: „Zamień za A” lub „Zamień za B”.
  + Na podstawie wyboru system określa, czy należy użyć *ASwapWidget* czy *BSwapWidget*.
* Następnie następuje warunek logiczny:
  + Jeśli **wszystkie wymagane pola** są wypełnione:
    - Dane zostają przekazane do widżetu zamiany (*setLabels(...)*),
    - Wyświetlany jest **zaktualizowany widok z przekształconym unitermem**.
  + Jeśli brakuje danych:
    - Aplikacja wyświetla **komunikat o błędzie** (za pomocą *QMessageBox*).

### 5.3.4. Etap 4: Zapis i przegląd danych

* Po udanej zamianie użytkownik klika przycisk **„Zapisz”**:
  + Aplikacja wyświetla **formularz dialogowy** (*save*) do podania:
    - *nazwa* – etykieta rekordu,
    - *opis* – dowolny opis operacji.
  + Wartości te, wraz z danymi logicznymi (*val1*, *val2*, *valW*, *valA*, *valB*, *valU*), są przekazywane do warstwy danych.
  + *Database::insertRecord(...)* zapisuje dane w tabeli *zapisy*.
* Po zapisie użytkownik może kliknąć **„Zobacz dane”**:
  + Uruchamiany jest komponent *DatabaseViewer*,
  + Wykonywane jest zapytanie *SELECT \* FROM zapisy*,
  + Wyniki są prezentowane w **widoku tabelarycznym**, umożliwiając przegląd zapisanych przypadków użycia aplikacji.

#### Mechanizmy kontrolne

Diagram zawiera również:

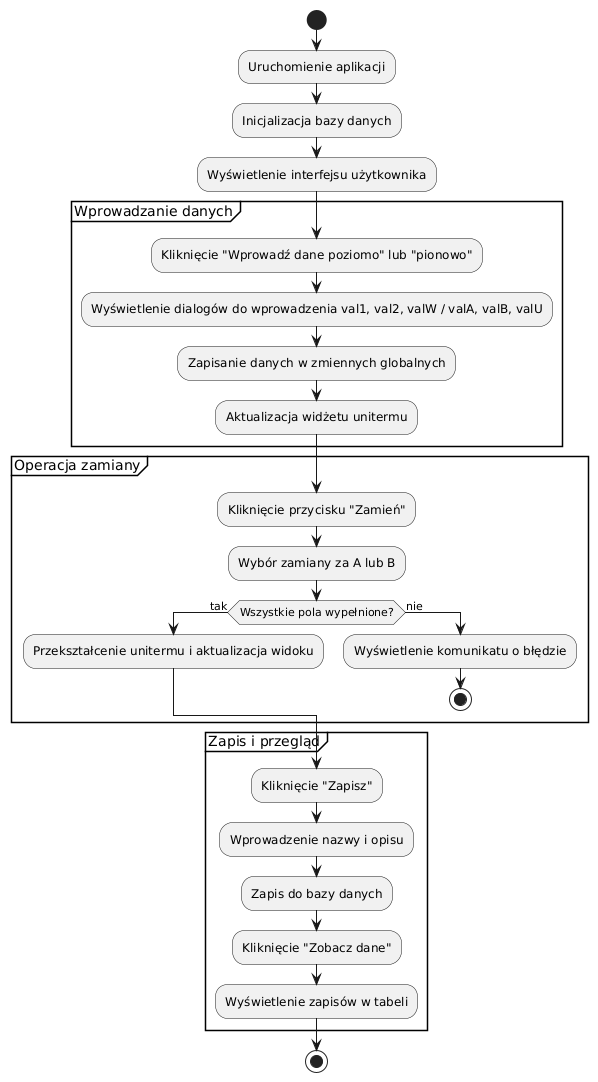
* **warunek logiczny** sprawdzający kompletność danych (przed zamianą),
* **ścieżkę alternatywną** w przypadku błędów wejścia,
* końcowy **punkt zakończenia aktywności** po wykonaniu operacji zapisu i/lub przeglądu danych.

#### Podsumowanie

Diagram aktywności doskonale oddaje **sekwencyjny, zdarzeniowy charakter działania aplikacji** i umożliwia przejrzyste śledzenie decyzji użytkownika oraz reakcji systemu.

Zastosowany model workflow:

* uwzględnia logikę wejścia, przetwarzania i wyjścia,
* uwzględnia wyjątki (brak danych),
* jest zgodny z zasadą **Single Responsibility Principle** – każda część odpowiada za jedno konkretne zadanie.



## 5.4. Diagram sekwencji

### 5.4.1. **Uruchomienie aplikacji**

* Użytkownik uruchamia program.
* *MainWindow* wywołuje *Database::initialize()*, aby nawiązać połączenie z bazą danych SQLite.
* W zależności od wyniku inicjalizacji (*true/false*), aplikacja kontynuuje lub zgłasza błąd.

### 5.4.2. **Wprowadzanie danych**

* Po kliknięciu przycisku „Wprowadź dane”, *MainWindow* otwiera sekwencyjne okna dialogowe (*QInputDialog*) do wprowadzenia wartości:  
  *val1*, *val2*, *valW* (dla pionowego układu) lub *valA*, *valB*, *valU* (dla poziomego).
* Wprowadzone wartości są przekazywane do komponentów graficznych (*VerticalWidget*, *HorizontalWidget*) poprzez metodę *setLabels(...)*, a ich wygląd jest aktualizowany w *paintEvent()*.

### 5.4.3. **Zamiana unitermów**

* Użytkownik klika przycisk „Zamień”, co powoduje wyświetlenie okna dialogowego (*QMessageBox*) z pytaniem o wariant zamiany (za A lub za B).
* Po dokonaniu wyboru, system sprawdza, czy wszystkie wymagane pola zostały poprawnie wypełnione.
* Jeśli tak – dane są przekazywane do odpowiedniego widżetu (*ASwapWidget* lub *BSwapWidget*) za pomocą *setLabels(...)*, a układ graficzny zostaje zaktualizowany.
* Jeśli nie – wyświetlany jest komunikat o błędzie.

### 5.4.4. **Zapis danych do bazy**

* Kliknięcie przycisku „Zapisz” uruchamia dialog *save*, w którym użytkownik podaje **nazwę** i **opis** rekordu.
* *MainWindow* przekazuje dane do klasy *Database*, wywołując metodę *insertRecord(...)*, która zapisuje dane (val1–valU) do bazy SQLite.
* Po udanym zapisie wyświetlany jest komunikat o sukcesie (*QMessageBox*).

## 5.4.5. Przegląd zapisanych danych

* Po kliknięciu „Zobacz dane” wywoływana jest instancja *DatabaseViewer*.
* Klasa ta wykonuje zapytanie SQL *SELECT \* FROM zapisy*, a otrzymane dane są ładowane do widoku tabelarycznego (*QTableWidget*).
* Użytkownik może w tym miejscu przeglądać wszystkie wcześniejsze operacje.

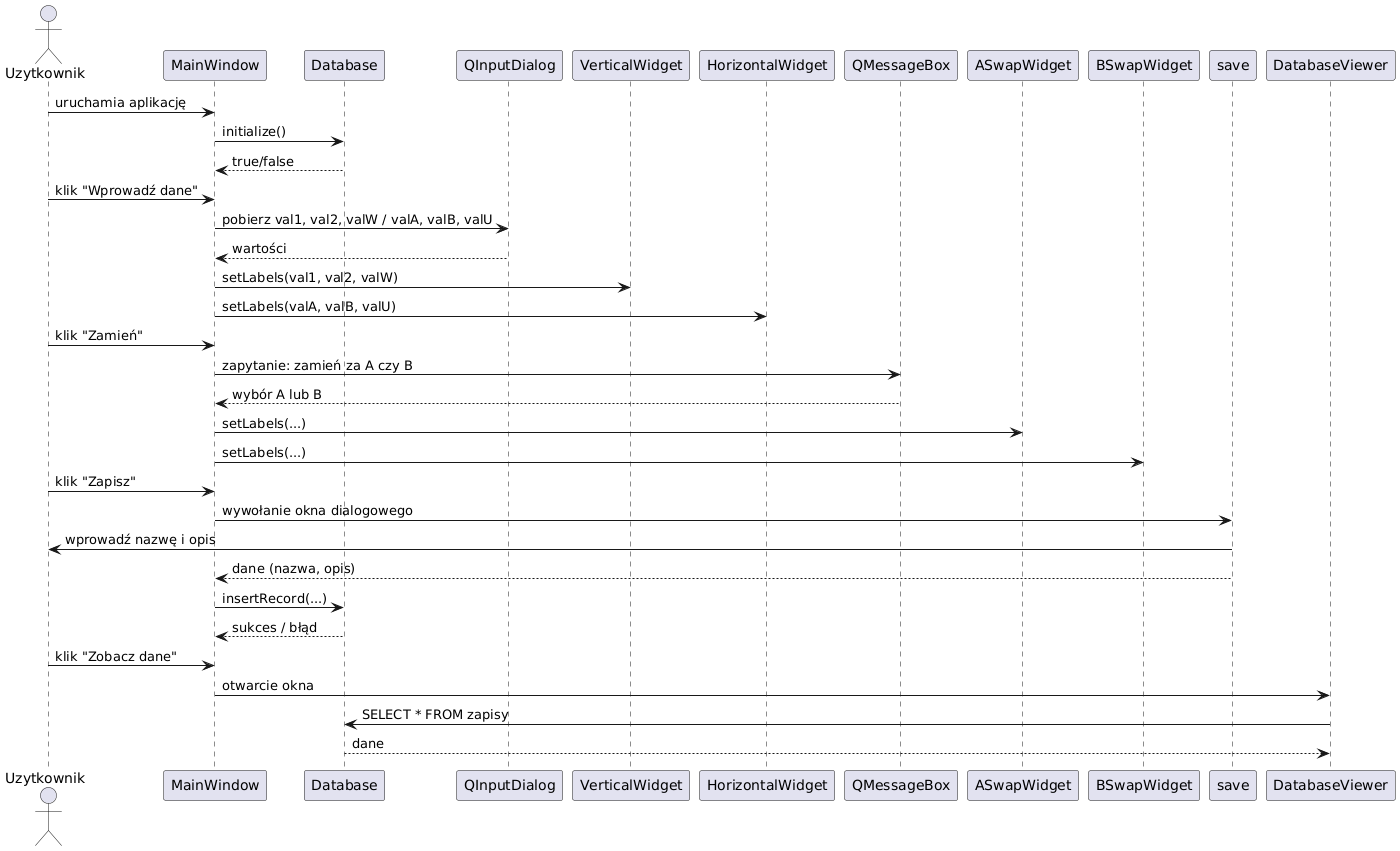
#### Kluczowe komponenty na diagramie

* **MainWindow** – centralna klasa kontrolera, zarządza interakcjami i koordynuje przepływ informacji między komponentami.
* **Database** – logika dostępu do danych (insert/select).
* **QInputDialog** – mechanizm wprowadzania danych.
* **VerticalWidget / HorizontalWidget / ASwapWidget / BSwapWidget** – komponenty odpowiedzialne za rysowanie unitermów.
* **QMessageBox** – komunikaty informacyjne i okna dialogowe.
* **save / DatabaseViewer** – formularz zapisu oraz komponent przeglądania bazy danych.

#### Podsumowanie

Diagram sekwencji pokazuje, że aplikacja oparta jest na **zdarzeniowym modelu sterowania (event-driven model)**, a wszystkie działania użytkownika prowadzą do wywołania konkretnych metod w odpowiednich komponentach.  
Zachowana została **jasna separacja odpowiedzialności** między komponentami GUI, logiką operacji oraz warstwą danych, co zapewnia wysoką spójność i niski stopień sprzężenia w systemie.

.



## 5.5. Diagram warstw systemu

Diagram warstw przedstawia **architekturę logiczną aplikacji** z wyraźnym podziałem na niezależne poziomy funkcjonalne. Taka struktura opiera się na zasadach **architektury warstwowej (layered architecture)** i zapewnia modularność, łatwość testowania oraz ułatwia rozbudowę systemu. Każda warstwa pełni odrębną rolę i komunikuje się wyłącznie z warstwami przyległymi.

### 5.5.1. Warstwa prezentacji (UI)

Warstwa interfejsu użytkownika odpowiedzialna jest za komunikację z użytkownikiem oraz wizualną reprezentację danych.

Składa się z następujących komponentów:

* ***ASwapWidget*, *BSwapWidget*** – komponenty odpowiedzialne za prezentację wyników przekształcenia unitermów w zależności od wybranego wariantu (za A lub za B),
* ***VerticalWidget*, *HorizontalWidget*** – odpowiadają za rysowanie unitermów w układzie pionowym lub poziomym,
* ***MainWindow*** – główna klasa okna aplikacji, pełniąca rolę integratora widżetów oraz pośrednika między UI a logiką aplikacji,
* ***DatabaseViewer*, *save*** – komponenty umożliwiające odpowiednio przeglądanie i zapisywanie danych w bazie.

UI nie zawiera logiki biznesowej, lecz przekazuje dane do warstwy aplikacji i od niej odbiera przetworzone wyniki.

### 5.5.2. Warstwa logiki aplikacji

To centralna warstwa odpowiedzialna za realizację zasad działania programu i przetwarzanie danych.

Główne funkcje:

* **Logika zamiany unitermów** – odpowiada za transformację struktury algebraicznej (np. podstawienie pionowej formy w miejsce jednego z operandów unitermu poziomego),
* **Zarządzanie danymi wejściowymi** – przechowuje tymczasowe wartości, pobrane z formularzy, oraz inicjuje aktualizacje widżetów.

Warstwa ta nie posiada wiedzy o implementacji interfejsu użytkownika ani o bazie danych – pozostaje niezależna i skupiona na przetwarzaniu.

### 5.5.3. Warstwa dostępu do danych

Izoluje logikę biznesową od technologii przechowywania danych. Odpowiada za:

* inicjalizację bazy (*initialize()*),
* zapis rekordów (*insertRecord(...)*),
* wykonywanie zapytań (*SELECT \* FROM zapisy*).

Zawarta w niej klasa *Database* stanowi **interfejs komunikacyjny z warstwą fizyczną** (bazą SQLite), zapewniając spójność i bezpieczeństwo danych.

### 5.5.4. Warstwa bazy danych

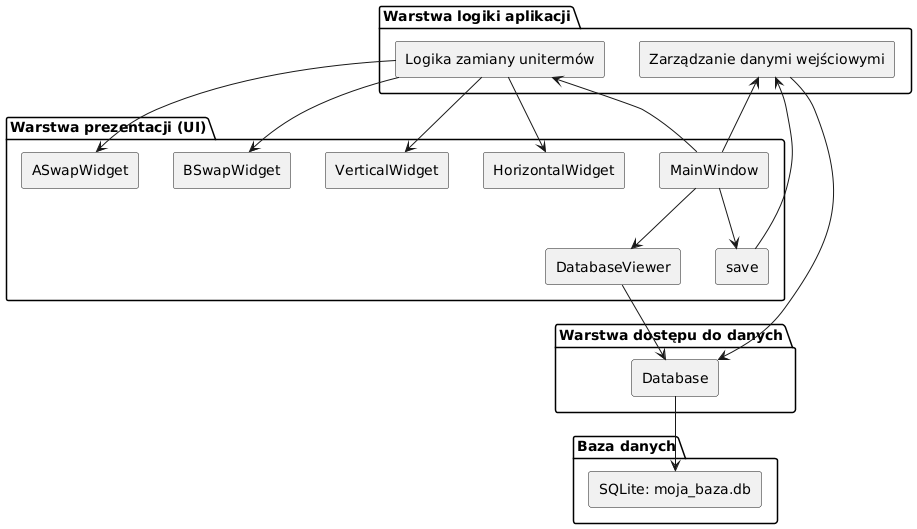
Ostatnia warstwa to **fizyczna baza danych SQLite**, reprezentowana przez plik *moja\_baza.db*. Służy do trwałego przechowywania informacji o operacjach wykonywanych w aplikacji (nazwy, opisy, wartości unitermów).

Warstwa ta jest niewidoczna dla użytkownika końcowego, ale w pełni wykorzystywana przez warstwę dostępu do danych.

#### Podsumowanie

Dzięki architekturze warstwowej aplikacja:

* spełnia zasadę **separacji odpowiedzialności (Separation of Concerns)**,
* umożliwia **łatwą konserwację i testowanie jednostkowe** każdej warstwy z osobna,
* sprzyja **rozszerzalności**, np. poprzez możliwość zastąpienia bazy SQLite innym silnikiem bez ingerencji w UI czy logikę aplikacji.



## 5.6. Diagram komponentów

Diagram komponentów prezentuje strukturę logiczną oraz zależności pomiędzy głównymi modułami aplikacji, zorganizowanymi według ich funkcji w systemie.

### **5.6.1. MainWindow Controller**

Moduł sterujący logiką głównego okna aplikacji.  
Zawiera dwa główne podkomponenty:

* ***MainWindowCtrl*** – centralny komponent odpowiedzialny za inicjalizację widżetów, obsługę interfejsu oraz koordynację przepływu danych,
* ***EventHandler*** – odpowiada za reakcje na zdarzenia użytkownika, takie jak kliknięcia przycisków i wybory opcji.

Zawiera również zestaw elementów interfejsu użytkownika (UI), takich jak *QPushButton*, *QInputDialog*, *QVBoxLayout*.

### **5.6.2. Widgets**

Pakiet zawierający komponenty wizualne, odpowiadające za prezentację danych w różnych formach:

* *VerticalWidget* i *HorizontalWidget* – odpowiadają za wyświetlanie unitermów odpowiednio w układzie pionowym i poziomym,
* *ASwapWidget* oraz *BSwapWidget* – reprezentują wynik operacji przekształcenia unitermu w zależności od wybranego wariantu zamiany.

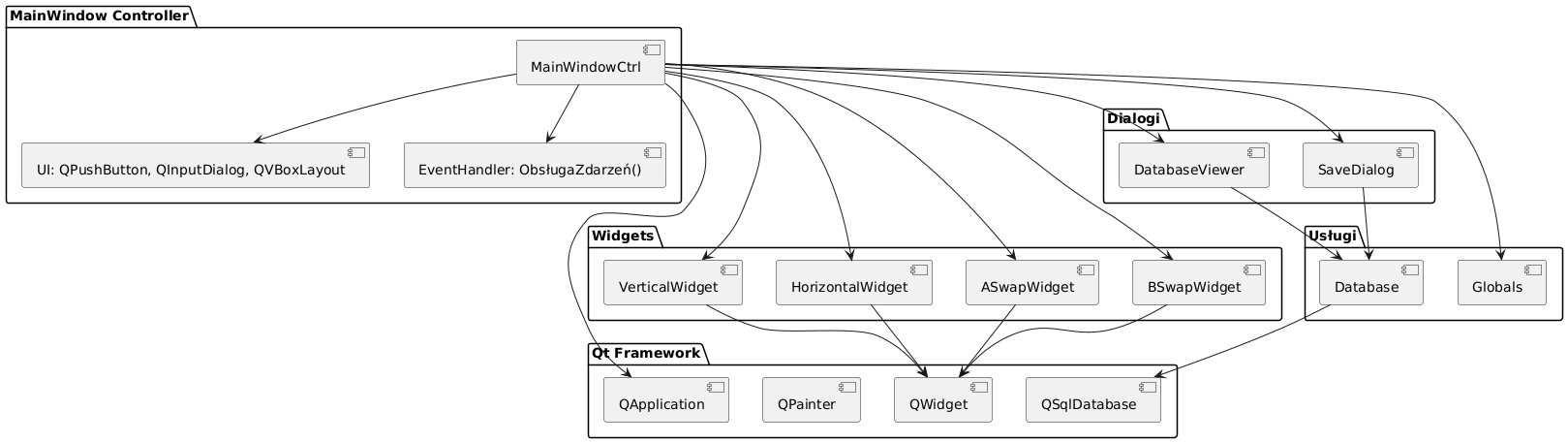
### **5.6.3. Dialog**

* *DatabaseViewer* – niezależny komponent dialogowy, służący do prezentacji danych zapisanych w bazie w postaci tabelarycznej.

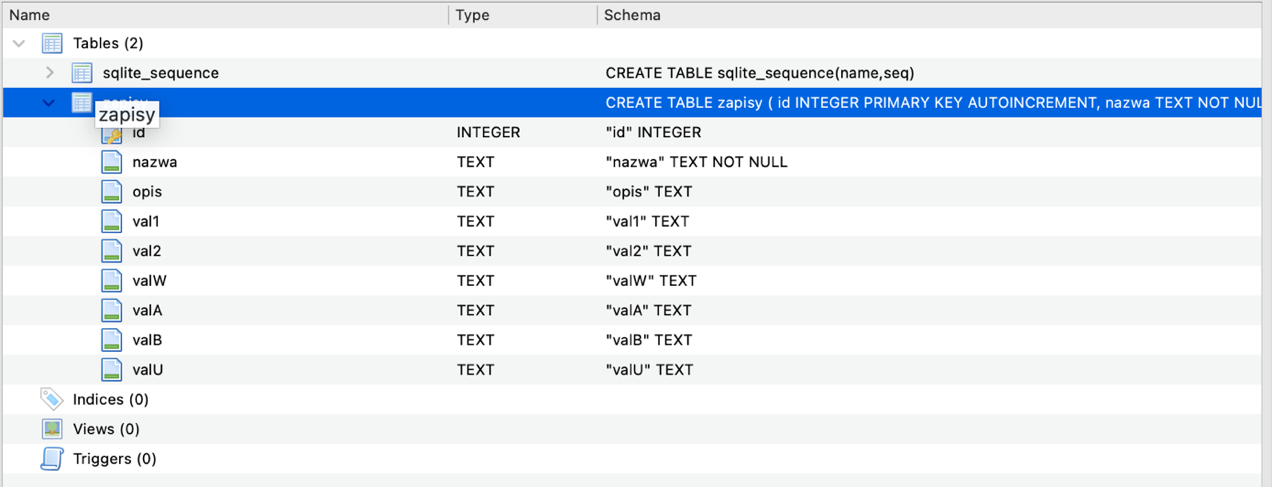
### **5.6.4. Qt Framework**

Zbiór komponentów biblioteki Qt, z których korzysta aplikacja:

* *QApplication* – główny obiekt zarządzający cyklem życia aplikacji,
* *QPainter* – odpowiada za renderowanie grafiki wewnątrz widżetów,
* *QWidget* – klasa bazowa dla wszystkich komponentów graficznych,
* *QSqlDatabase* – mechanizm komunikacji z bazą danych SQLite.

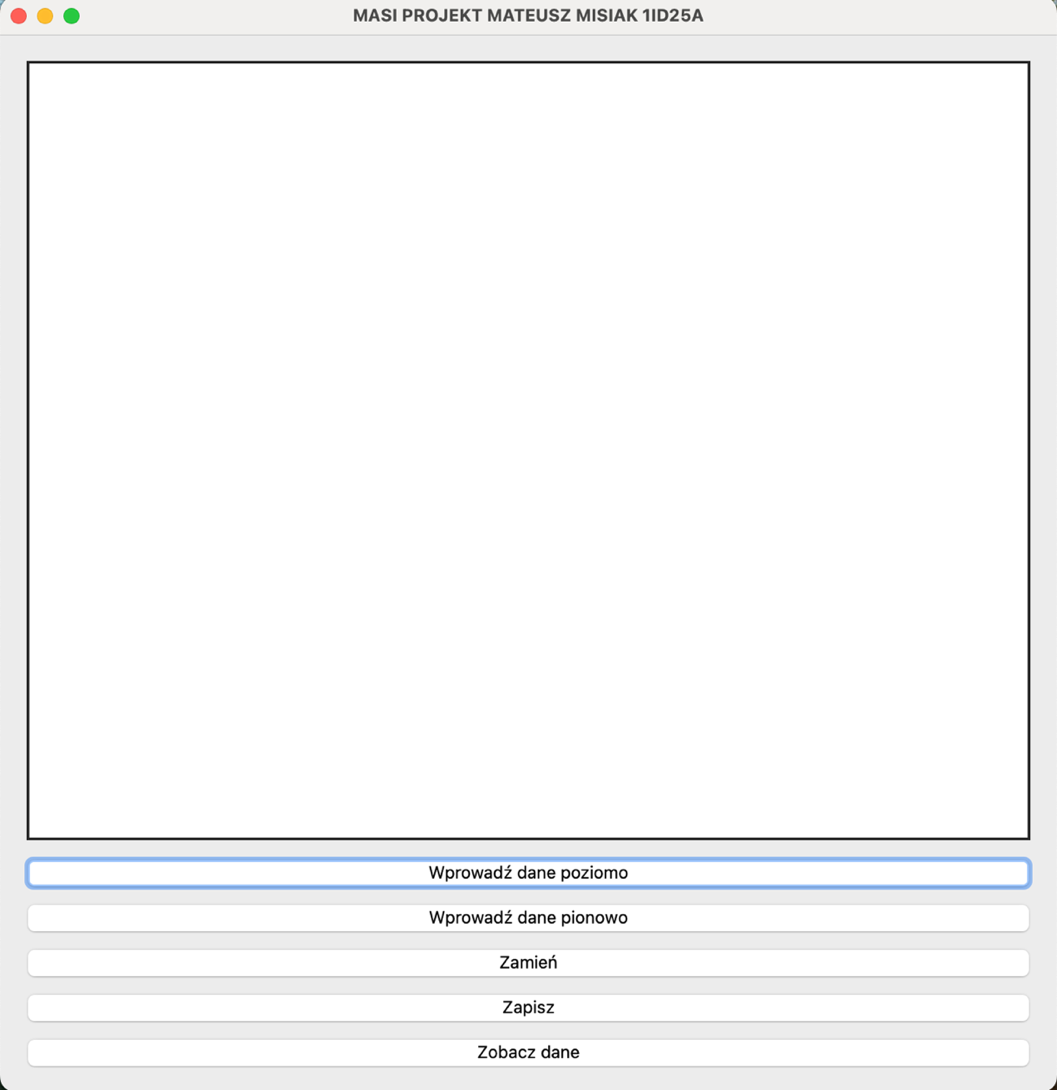


# 6. Struktura bazy danych



# 7. Zrzuty ekranu z działania aplikacji

## 7.1. Ekran główny aplikacji

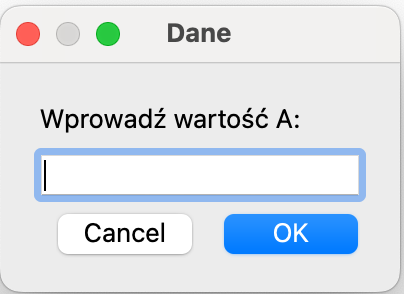


Na powyższym zrzucie ekranu przedstawiono główne okno aplikacji zaprojektowanej w ramach projektu MASI. Okno nosi tytuł **„MASI PROJEKT MATEUSZ MISIAK 1ID25A”**, co jednoznacznie identyfikuje autora oraz temat realizacji. Centralnym elementem interfejsu użytkownika jest duża ramka (obszar roboczy) służąca jako płaszczyzna do graficznego przedstawiania unitermów w postaci poziomej lub pionowej.

Pod ramką znajduje się zestaw pięciu przycisków, które umożliwiają użytkownikowi pełną obsługę programu:

1. **„Wprowadź dane poziomo”** – uruchamia dialog umożliwiający wprowadzenie danych wejściowych dla unitermu poziomego.
2. **„Wprowadź dane pionowo”** – otwiera formularz wprowadzania danych dla unitermu w układzie pionowym.
3. **„Zamień”** – pozwala na przekształcenie unitermu poziomego na pionowy lub odwrotnie. Przycisk wyświetla interaktywne okno z wyborem wariantu transformacji (za A lub za B).
4. **„Zapisz”** – umożliwia zapisanie aktualnych danych oraz opisu operacji do lokalnej bazy danych SQLite.
5. **„Zobacz dane”** – pozwala na przeglądanie wcześniej zapisanych rekordów w dedykowanym oknie przeglądarki bazy danych.

## 7.2. Okno dialogowe – wprowadzanie danych



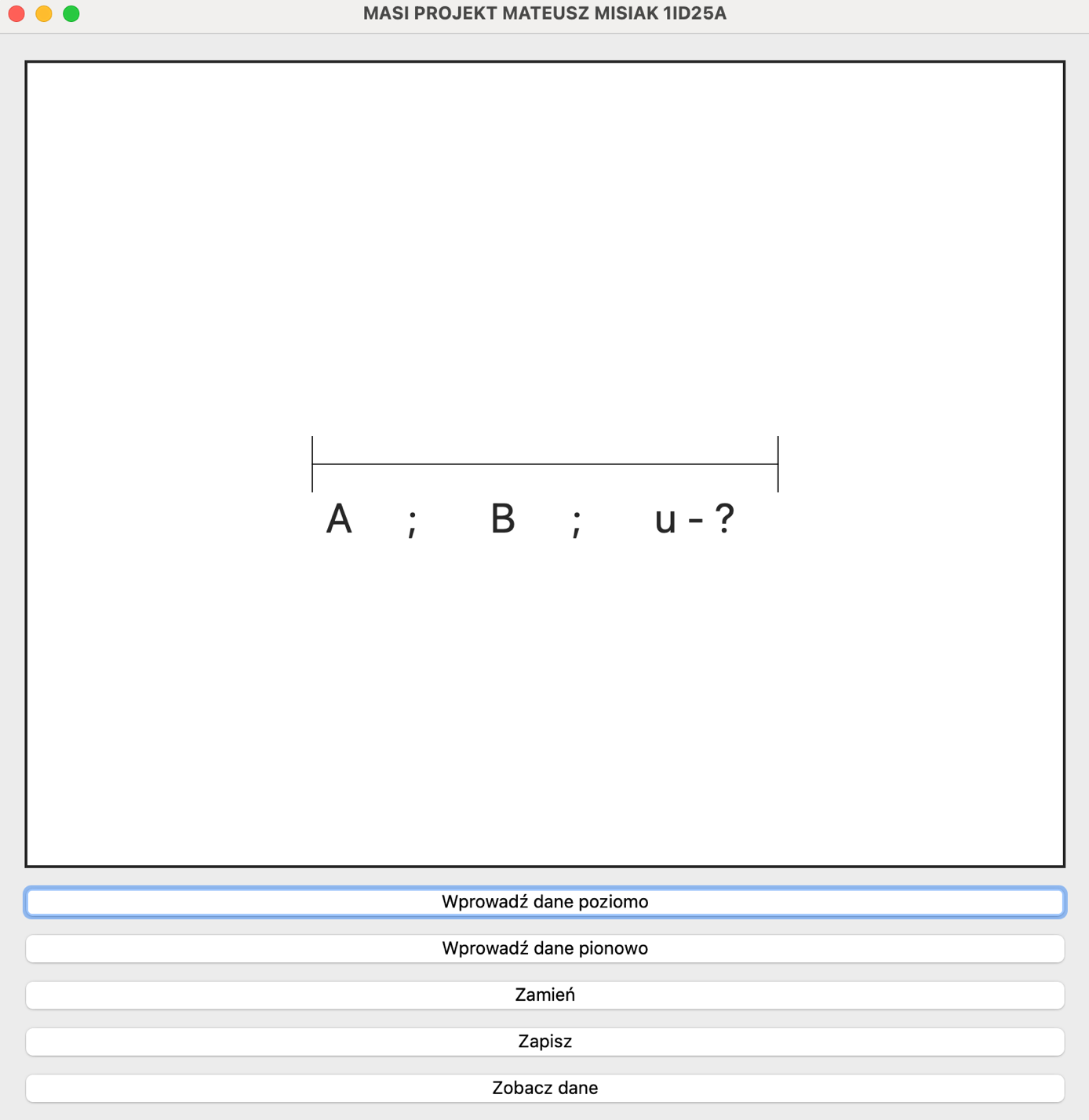
Na powyższym zrzucie przedstawiono **okno dialogowe wejściowe**, które pojawia się po kliknięciu przycisku **„Wprowadź dane poziomo”**. Jest to interfejs umożliwiający użytkownikowi ręczne wprowadzenie jednej z wymaganych wartości — w tym przypadku **wartości A**.

Okno nosi tytuł **„Dane”** i zawiera:

* etykietę z komunikatem: **„Wprowadź wartość A:”**,
* pole tekstowe do wpisania danej,
* dwa przyciski funkcyjne:
  + **„Cancel”** – umożliwiający anulowanie operacji,
  + **„OK”** – potwierdzający wprowadzenie danych i przekazujący wartość do aplikacji.

Okno to jest częścią dynamicznego mechanizmu interakcji z użytkownikiem, który pozwala wprowadzać wartości do graficznego modelu unitermów. Analogiczne okna pojawiają się także dla wartości B oraz parametru warunkowego *u* lub *w*, w zależności od wybranej ścieżki wejściowej (poziomej lub pionowej).

## 7.3. Graficzna prezentacja unitermu poziomego



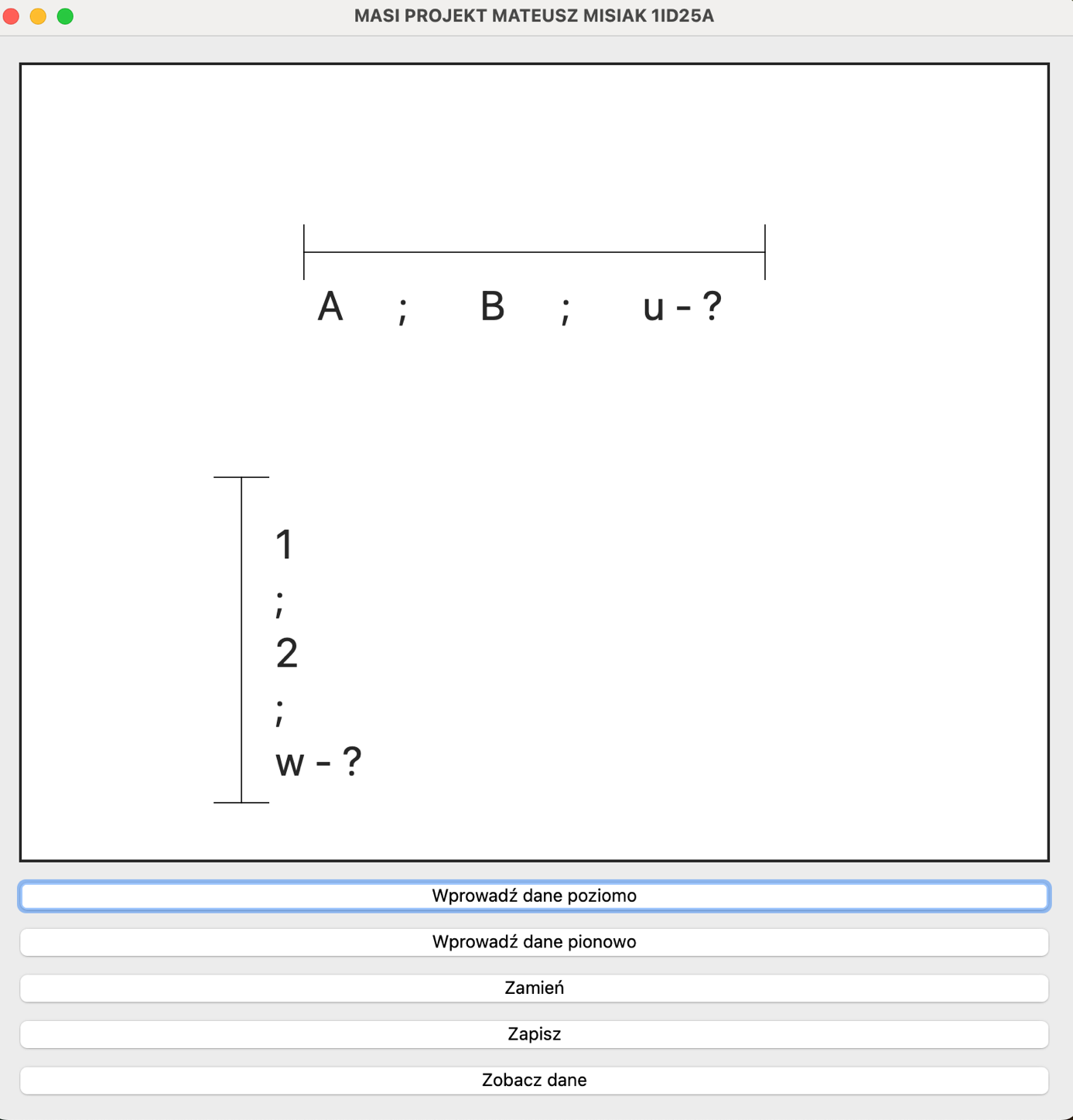
Na powyższym zrzucie ekranu widoczna jest **graficzna reprezentacja unitermu poziomego**, która pojawia się w obszarze roboczym aplikacji po wprowadzeniu danych za pomocą przycisku **„Wprowadź dane poziomo”**.

W centralnej części widoczna jest pozioma linia oznaczająca strukturę operacji eliminowania, uzupełniona trzema tekstowymi etykietami:

* **A** – pierwsza wartość (pierwszy operand),
* **B** – druga wartość (drugi operand),
* **u - ?** – miejsce warunku lub wyniku, którego wartość użytkownik może później obliczyć lub wskazać w kolejnych krokach.

Separator ***;*** oddziela poszczególne elementy logiczne, co jest zgodne z zapisem algebraicznym operacji eliminowania w algebrze algorytmów.

## 7.4. Równoczesna prezentacja unitermów poziomego i pionowego



Na zaprezentowanym zrzucie ekranu widoczna jest sytuacja, w której użytkownik wprowadził dane zarówno w formacie **poziomym**, jak i **pionowym**. Aplikacja umożliwia jednoczesne graficzne przedstawienie obu wersji eliminowania unitermów w głównym obszarze roboczym.

**W górnej części**:

* Wyświetlony jest uniterm poziomy w formie:  
  *A ; B ; u - ?*  
  Reprezentuje klasyczną sekwencję operacji, w której *A* i *B* są operandami, a *u* stanowi warunek bądź wynik operacji eliminowania.

**W dolnej części**:

* Przedstawiony został odpowiadający mu uniterm pionowy:

1

;

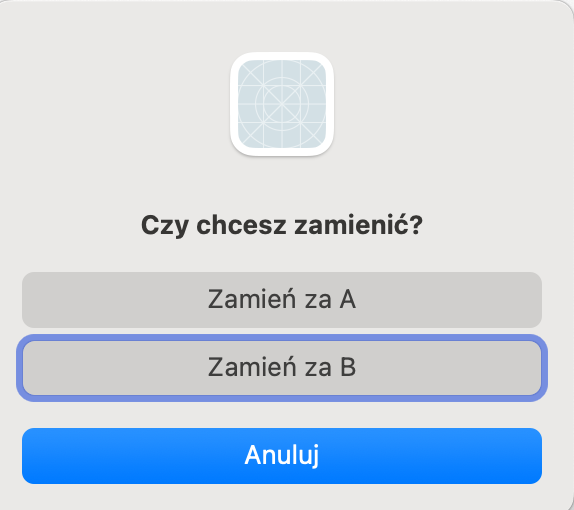
2

;

w - ?

W tej formie dane układane są kolumnowo, co symbolizuje operację pionowego eliminowania w rozszerzonej algebrze algorytmów.

## 7.5. Okno dialogowe – wybór sposobu zamiany unitermu

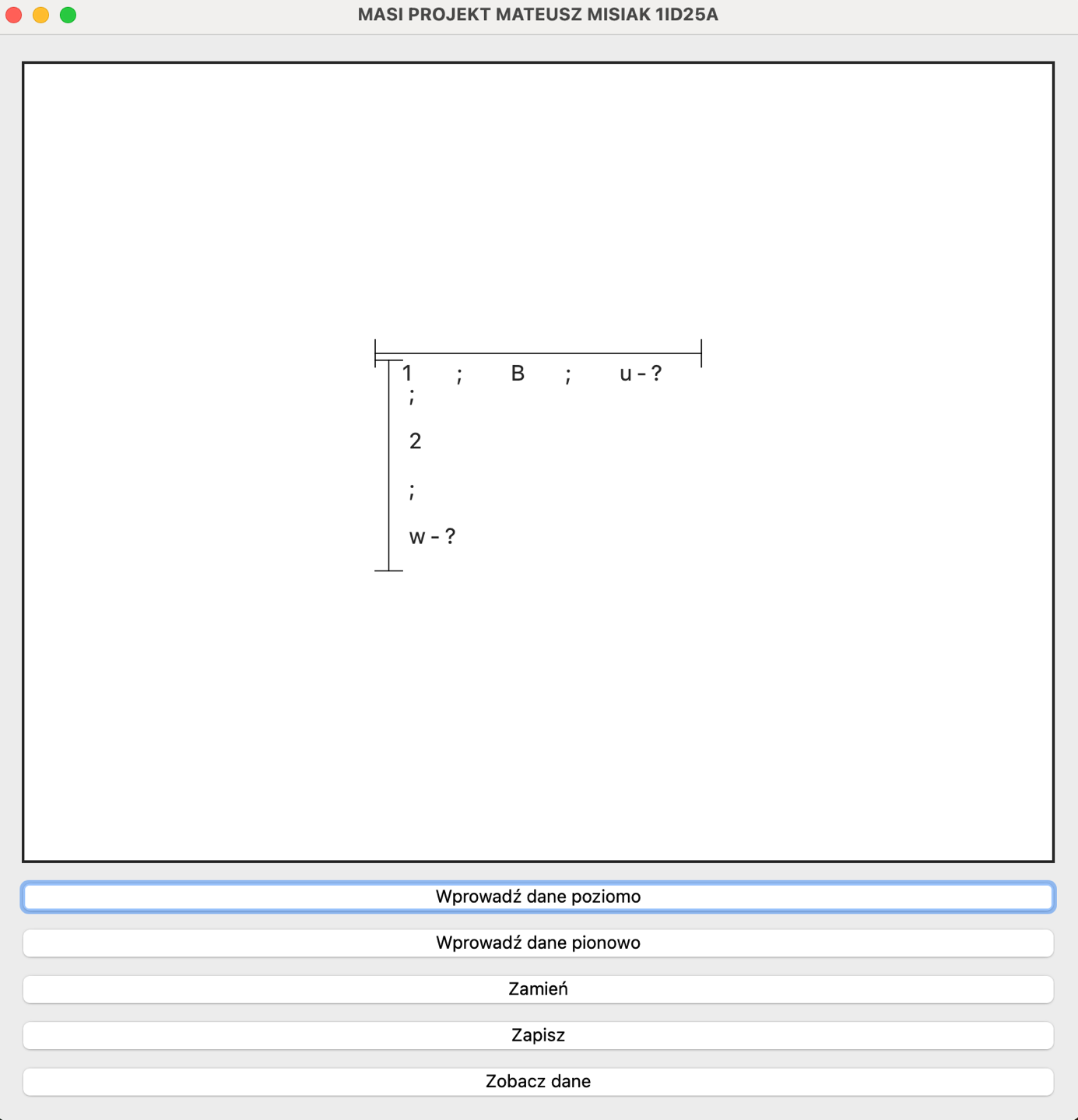


Na powyższym zrzucie przedstawiono **okno dialogowe potwierdzenia operacji zamiany**, które pojawia się po kliknięciu przycisku **„Zamień”** w głównym oknie aplikacji. Jest to kluczowy element interfejsu użytkownika umożliwiający wybór sposobu przekształcenia unitermu z formy poziomej na pionową.

Okno zawiera:

* pytanie: **„Czy chcesz zamienić?”** – informujące użytkownika o planowanej operacji,
* dwa przyciski akcji:
  + **„Zamień za A”** – powoduje utworzenie pionowego unitermu z podstawieniem w miejsce wartości *A*,
  + **„Zamień za B”** – powoduje analogiczne przekształcenie względem wartości *B*,
* przycisk **„Anuluj”**, który pozwala użytkownikowi zrezygnować z operacji bez wprowadzania zmian.

## 7.6. Efekt zamiany unitermu – przypadek „Zamień za A”



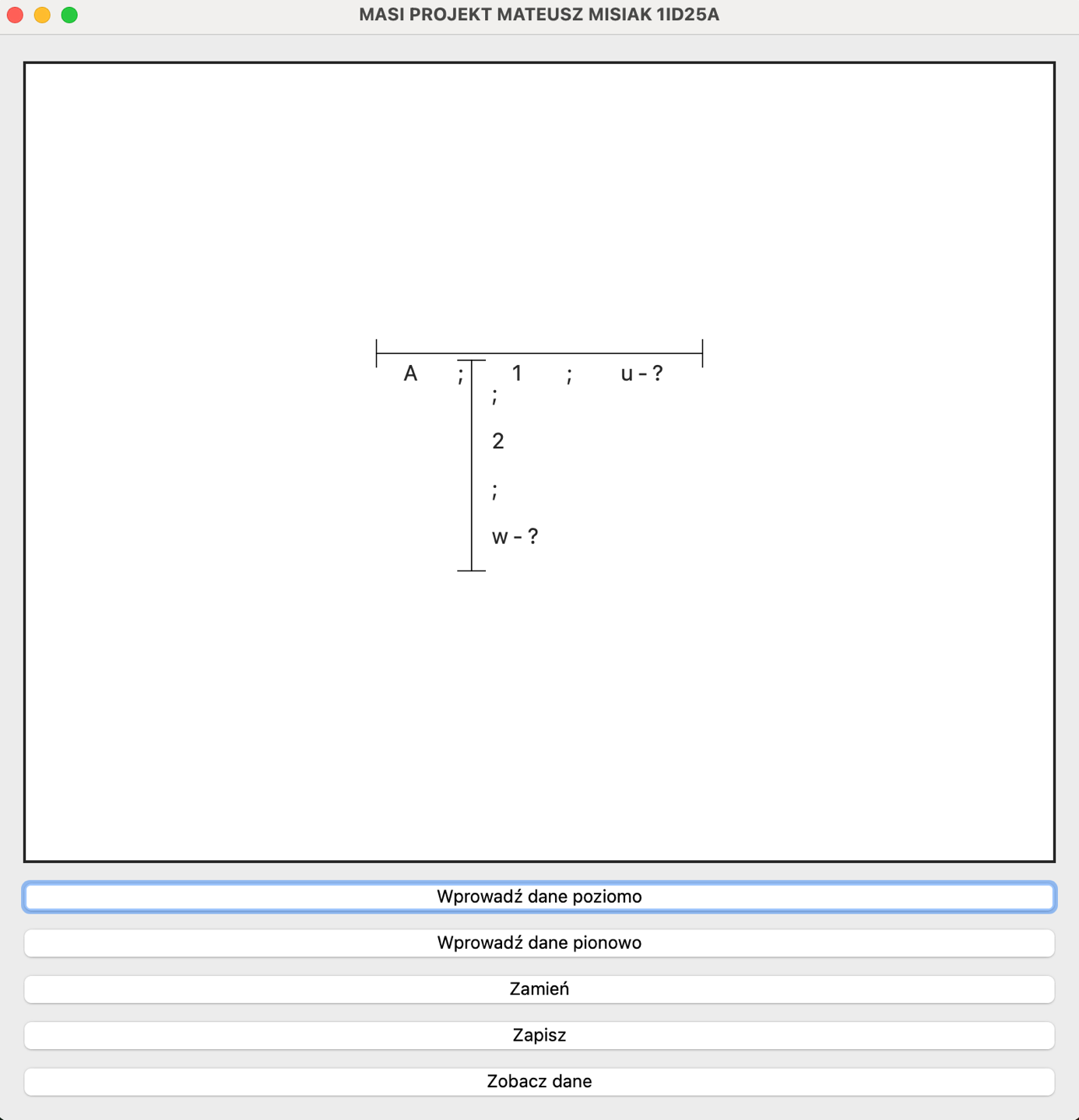
Na powyższym zrzucie ekranu przedstawiony jest wynik działania aplikacji po dokonaniu transformacji unitermu za pomocą opcji **„Zamień za A”**, wybranej w oknie dialogowym.

Efektem operacji jest:

* **nowa graficzna reprezentacja**, w której część **pionowa** została wstawiona w miejsce wartości *A* w unitermie poziomym,

lub — jak pokazano na zrzucie — wizualnie rozdzielony pionowy uniterm znajduje się w miejscu pozycji *A*, przy zachowaniu pozostałych elementów poziomego zapisu (*B* oraz *u - ?*).

## 7.7. Efekt zamiany unitermu – przypadek „Zamień za B”

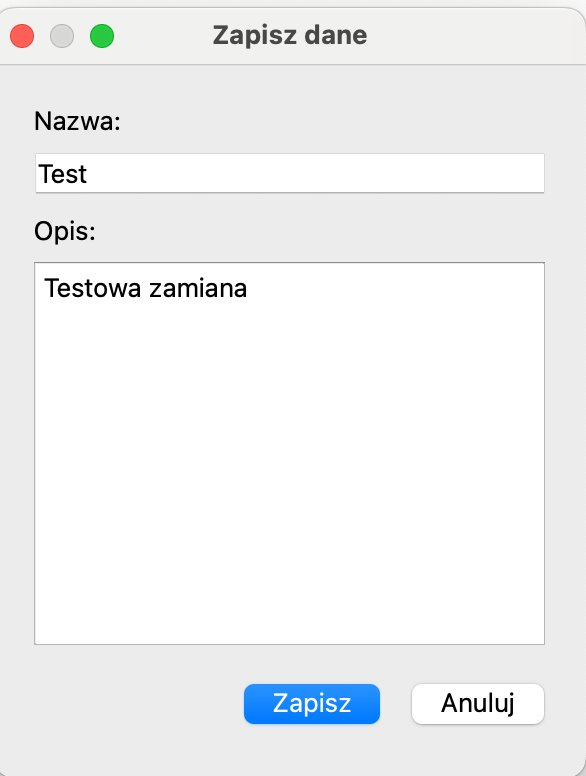


Powyższy zrzut ekranu przedstawia rezultat działania aplikacji po wykonaniu transformacji z użyciem opcji **„Zamień za B”**.

W tym wariancie:

* **uniterm pionowy** (*1 ; 2 ; w - ?*) został osadzony w miejscu wartości *B* w strukturze poziomej,
* pozostałe elementy (*A* oraz *u - ?*) zachowały swoje pozycje,
* wynikowy układ przedstawia zagnieżdżoną strukturę logiczną, w której **pionowy blok zastępuje operand *B***.

## 7.8. Okno zapisu danych – formularz metadanych



Na przedstawionym zrzucie ekranu widoczne jest **okno dialogowe zapisu danych**, które pojawia się po kliknięciu przycisku **„Zapisz”** w głównym oknie aplikacji. Jest to formularz służący do archiwizacji aktualnego stanu operacji unitermów w lokalnej bazie danych SQLite.

Okno nosi tytuł **„Zapisz dane”** i zawiera dwa pola wejściowe:

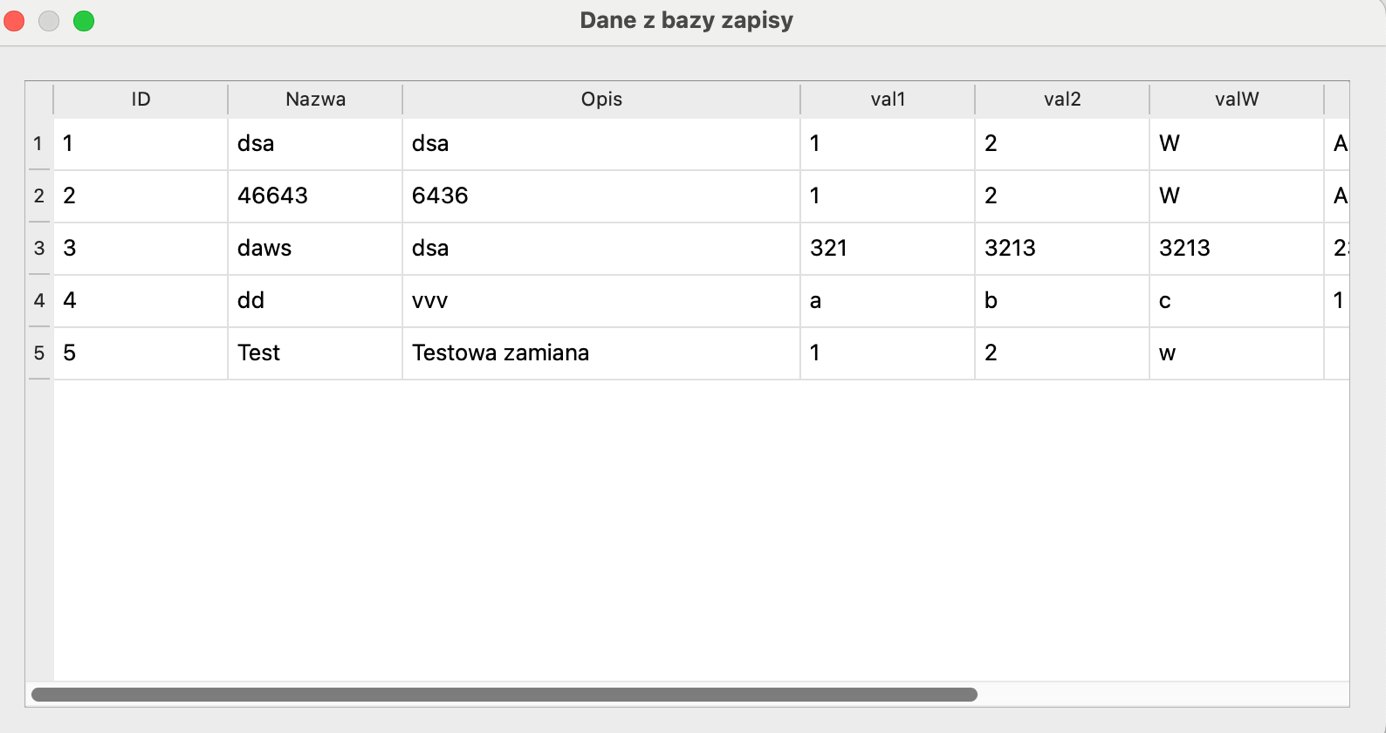
* **„Nazwa:”** – pole tekstowe, w którym użytkownik podaje unikalną nazwę rekordu (np. identyfikator transformacji lub scenariusza testowego),
* **„Opis:”** – pole tekstowe wielowierszowe (textarea), pozwalające użytkownikowi dodać komentarz opisujący kontekst lub cel danej transformacji (np. „Testowa zamiana”).

Na dole znajdują się dwa przyciski funkcyjne:

* **„Zapisz”** – zatwierdza formularz i zapisuje dane do bazy,
* **„Anuluj”** – zamyka okno bez wykonania operacji.

Po kliknięciu przycisku „Zapisz”, dane z obu reprezentacji (poziomej i pionowej) oraz wartości pól *val1, val2, valW, valA, valB, valU* zostają wprowadzone do tabeli *zapisy* w pliku *moja\_baza.db*.

## 7.9. Podgląd zapisanych danych – widok bazy



Na powyższym zrzucie ekranu zaprezentowano **okno przeglądu danych z bazy**, które pojawia się po kliknięciu przycisku **„Zobacz dane”**. Aplikacja umożliwia użytkownikowi wgląd do wszystkich uprzednio zapisanych rekordów w lokalnej bazie SQLite (*moja\_baza.db*).

Tabela zawiera kolumny odpowiadające strukturze danych zapisanych w bazie:

* **ID** – unikalny identyfikator rekordu,
* **Nazwa** – nadana przez użytkownika etykieta scenariusza,
* **Opis** – tekstowy komentarz opisujący kontekst transformacji,
* **val1, val2, valW, valA, valB, valU** – wartości jednostek unitermów zapisane podczas działania aplikacji.

W wierszu nr 5 widoczny jest rekord utworzony podczas testowego zapisu („Test”, „Testowa zamiana”), co potwierdza poprawność działania zarówno mechanizmu zapisu, jak i odczytu danych z bazy.

Dzięki temu widokowi użytkownik może:

* porównywać różne transformacje,
* analizować dane historyczne,
* lub w przyszłości — rozszerzyć aplikację o możliwość edycji/usuwania rekordów.

To finalny komponent systemu, zamykający pełny cykl: **wprowadzenie danych → transformacja → zapis → przegląd**.

# 8. Wnioski i podsumowanie

Projekt dotyczył opracowania aplikacji komputerowej, która umożliwia transformację unitermu z poziomej operacji eliminowania na pionową. Program został stworzony w języku C++ z wykorzystaniem biblioteki Qt 6. Zastosowano lokalną bazę danych SQLite do przechowywania informacji o wykonanych transformacjach, w tym nazw, opisów oraz wartości wejściowych.

Aplikacja umożliwia wprowadzenie danych w dwóch układach: poziomym (A ; B ; u - ?) oraz pionowym (1 ; 2 ; w - ?). Użytkownik może wybrać sposób przekształcenia – przez podstawienie za operand A lub operand B. Interfejs został zrealizowany za pomocą widżetów graficznych, które rysują poziome i pionowe formy unitermów oraz wynik transformacji.

Struktura projektu obejmuje kilka niezależnych komponentów: widżety do rysowania (*HorizontalWidget, VerticalWidget, ASwapWidget, BSwapWidget*), moduł do zapisu danych (*save*) oraz komponent obsługujący bazę danych (*database*). Zmienne użytkownika są przechowywane w osobnym module *globals*, co pozwala na dostęp do wartości w wielu częściach programu.

Zapis danych realizowany jest przez formularz, który przyjmuje nazwę i opis. Po zatwierdzeniu, dane są przekazywane do bazy SQLite i zapisywane jako nowy rekord. Użytkownik może przeglądać zapisane rekordy w osobnym oknie z widokiem tabeli.

Transformacja unitermów została zrealizowana poprzez graficzne przedstawienie zamiany – pionowy uniterm jest osadzany w miejsce A lub B w unitermie poziomym, w zależności od wyboru użytkownika. Rysowanie komponentów opiera się na klasie QPainter i odbywa się w metodzie *paintEvent*.

Pod względem technicznym, projekt nie wymaga dostępu do internetu ani zewnętrznych usług. Aplikacja działa lokalnie i wszystkie dane są zapisywane w pliku bazy danych o nazwie *moja\_baza.db.*

Projekt został wykonany zgodnie z założeniami tematu. Umożliwia wprowadzenie danych, przekształcenie unitermów oraz zapis i przegląd wyników transformacji.