AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

PROGRAMOWANIE URZĄDZEŃ MOBILNYCH

Data Point

Autor:
Tomasz Iwański
Adrian Kądziołka
Kacper Kosal

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2025

Spis treści

1. Ogólne określenie wymagań

Celem projektu jest stworzenie zaawansowanego programu w języku C++, który umożliwi wczytanie danych z pliku CSV, ich analizę oraz obsługę użytkownika za pomocą interaktywnego menu. Program musi spełniać wymagania dotyczące przetwarzania danych, ich przechowywania oraz wyświetlania wyników w oparciu o zapytania użytkownika. W projekcie należy wykorzystać wzorce projektowe, organizację kodu w osobnych plikach oraz testy jednostkowe.

1.1. Wczytywanie danych z pliku CSV

- Pomijanie pierwszej linii (nagłówka).
- Weryfikacja poprawności danych i odrzucanie błędnych rekordów.
- Obsługa błędnych linii (niepełnych, powtórzonych, pustych) i ich zapisywanie w logach.

1.1.1. Logowanie procesu wczytywania

- worzenie dwóch plików logów
  - log_data_godzina.txt: zapis wszystkich poprawnych i niepoprawnych rekordów.
  - log_error_data_godzina.txt: zapis tylko błędnych rekordów.
- Dynamiczne generowanie nazw plików z datą i godziną.

1.1.2. Struktura danych

- Reprezentacja pojedynczego rekordu jako obiektu w pamięci dynamicznej (na stercie).
- Organizacja danych w strukturze drzewa
  - Korzeń: rok.
  - Poziomy: miesiąc, dzień, ćwiartka doby (00:00–5:45, 6:00–11:45, 12:00–17:45, 18:00–23:45).

1.1.3. Analiza danych

- Obliczanie sum i średnich dla poszczególnych parametrów (autokonsumpcja, eksport, import, pobór, produkcja) w wybranych przedziałach czasowych.
- Porównywanie wartości parametrów pomiędzy dwoma przedziałami czasowymi.
- Wyszukiwanie rekordów na podstawie zadanych kryteriów (np. wartość z tolerancją i przedział czasowy).
- Wyświetlanie danych z określonego przedziału czasowego.

1.1.4. Interaktywne menu użytkownika

- Wczytywanie danych z pliku CSV.
- Zapisywanie danych do pliku binarnego.
- Odczyt danych z pliku binarnego (zawsze do pustego programu).
- Analiza i wyświetlanie wyników w oparciu o funkcje analityczne.

1.1.5. Obsługa wyjątków i braków danych

- Program musi działać bez przestojów nawet w przypadku napotkania błędnych danych.
- Dane nieciągłe czasowo muszą być poprawnie obsługiwane.

1.1.6. Iterator jako wzorzec projektowy

- Użycie iteratora do poruszania się po węzłach drzewa danych.

1.2. Wymagania techniczne

1.2.1. Kod w języku C++

- Każda klasa w osobnym pliku.
- Kod zgodny z zasadami obiektowości.

1.2.2. Testy jednostkowe

- Stworzenie testów przy użyciu frameworka GoogleTest do weryfikacji poprawności metod.

1.2.3. Dokumentacja projektu

- Przygotowanie dokumentacji technicznej w formacie LaTeX i DoxyGen
- Publikacja projektu na GitHub z kompletną historią zmian i plikami projektu

1.2.4. Wymagania dotyczące logów

- Pliki logów muszą być czytelne i zawierać szczegółowe informacje o przebiegu procesu.

2. Określenie wymagań szczegółowych

2.1. Cel aplikacji

Aplikacja ma na celu analizę danych energetycznych zapisanych w pliku CSV. Pozwala użytkownikowi na wczytanie pliku, przetwarzanie danych oraz wykonywanie obliczeń takich jak suma, średnia oraz porównania dla zadanego zakresu czasu. Program obsługuje błędne dane wejściowe, generując odpowiednie logi i umożliwiając dalsze przetwarzanie tylko poprawnych danych.

2.2. Zakres aplikacji

Aplikacja obsługuje następujące funkcjonalności:

- Wczytywanie pliku CSV z danymi energetycznymi.
- Analiza i walidacja danych, odrzucanie błędnych rekordów.
- Tworzenie logów zawierających informacje o poprawnych i błędnych rekordach.
- Przetwarzanie danych do struktury drzewa zawierającego hierarchię: rok, miesiąc, dzień, ćwiartka (6 godzin).
- Wykonywanie operacji takich jak suma, średnia oraz porównanie danych w zdefiniowanych przedziałach czasowych.
- Eksport danych do pliku binarnego oraz ich ponowne wczytywanie.
- Obsługa niepożądanych sytuacji (np. brak pliku, błędny format danych).

2.3. Dane wejściowe

Dane wejściowe to plik CSV zawierający następujące pola:

- Data i godzina pomiaru.
- Autokonsumpcja (energia zużyta bezpośrednio przez odbiorniki).
- Eksport (energia wysłana do sieci energetycznej).
- Import (energia pobrana z sieci energetycznej).
- Pobór (energia zużyta przez odbiorniki).

- Produkcja (energia wyprodukowana przez falownik).

Pierwsza linia pliku zawiera nagłówki i nie jest uwzględniana w analizie danych.

2.4. Opis elementów interfejsu i zdarzeń

2.4.0.1. Interfejs aplikacji: Aplikacja zawiera następujące elementy interfejsu użytkownika:

- Menu główne z opcjami:
  - **Wczytaj plik CSV** – umożliwia wybór i wczytanie pliku.
  - **Zapisz dane do pliku binarnego** – zapisuje wczytane dane do pliku binarnego.
  - **Wczytaj dane z pliku binarnego** – odczytuje dane zapisane w pliku binarnym.
  - **Analiza danych** – umożliwia wykonanie operacji takich jak suma, średnia, porównanie.
  - **Wyjście** – zamyka aplikację.
- Okna dialogowe do wprowadzania przedziałów czasowych oraz wyboru operacji analitycznych.
- Pola tekstowe do wprowadzenia dodatkowych parametrów (np. wartości wyszukiwania z tolerancją).

2.4.0.2. Zdarzenia aplikacji:

- Kliknięcie przycisku **Wczytaj plik CSV**: otwiera okno dialogowe umożliwiające wybór pliku. Po wczytaniu pliku generowane są logi i wyświetlane podsumowanie poprawnych oraz błędnych rekordów.
- Kliknięcie przycisku **Zapisz dane do pliku binarnego**: zapisuje dane w formacie binarnym.
- Kliknięcie przycisku **Wczytaj dane z pliku binarnego**: odczytuje dane i zastępuje bieżące dane w aplikacji.
- Kliknięcie przycisku **Analiza danych**: otwiera okno dialogowe do wyboru operacji (np. suma, średnia) oraz określania zakresu czasu.
- Obsługa zdarzeń niepożądanych (np. błędny format pliku): wyświetla odpowiedni komunikat o błędzie.

2.5. Możliwości dalszego rozwoju

Aplikacja może zostać rozszerzona o następujące funkcjonalności:

- Wizualizacja danych w formie wykresów.
- Integracja z bazą danych w celu przechowywania wyników analizy.
- Automatyczne generowanie raportów w formacie PDF.
- Obsługa dodatkowych formatów danych (np. JSON, XML).
- Zdalny dostęp do danych za pośrednictwem API.

2.6. Zachowanie aplikacji w sytuacjach niepożądanych

Aplikacja reaguje na nieprzewidziane sytuacje w następujący sposób:

- Błędne linie w pliku CSV są ignorowane, a informacje o nich zapisywane w logach.
- W przypadku braku ciągłości czasowej dane są sortowane według czasu.
- Jeśli wczytany plik nie istnieje lub ma niepoprawny format, wyświetlany jest komunikat o błędzie.
- Operacje analizy na pustym zbiorze danych są blokowane z odpowiednim komunikatem dla użytkownika.
- Logi zawierają szczegółowe informacje o postępie wczytywania i błędach, co umożliwia ich dalszą analizę.

3. Projektowanie

W ramach etapu projektowania przygotowano środowisko pracy, zaprojektowano strukturę projektu oraz określono dokładne wymagania funkcjonalne aplikacji. Poniżej przedstawiono szczegóły związane z każdym z etapów.

3.1. Przygotowanie narzędzi

Do realizacji projektu wykorzystano następujące narzędzia:

- **Visual Studio** – środowisko programistyczne do pisania i debugowania kodu w języku C++.[1]
- **GitHub** – platforma do zarządzania kodem źródłowym oraz współpracy w zespole.[2]
- **GoogleTest** – framework do tworzenia testów jednostkowych.[3]
- **Doxygen** – narzędzie do generowania dokumentacji technicznej kodu.[4]
- **Overleaf** – narzędzie do wspólnego pisania dokumentacji w LaTeX.[5]

3.2. Struktura projektu

Projekt podzielono na klasy, z których każda odpowiada za określoną funkcjonalność. Poniżej przedstawiono szczegóły dotyczące zaprojektowanych klas.

Klasa `DataPoint`

- Reprezentuje pojedynczy punkt danych, odpowiadający jednej linii z pliku CSV.
- **Pola:**
  - `std::string dateTime` – data i godzina pomiaru.
  - `double autokonsumpcja, eksport, import, pobor, produkcja` – wartości pomiarowe.
- **Metody:**
  - Konstruktor i destruktor.
  - Metoda walidująca dane z linii CSV.
  - Metoda parsująca linię CSV na obiekt.

Klasa `Quarter`

- Reprezentuje dane dla ćwiartki dnia (np. 00:00-05:45).
- **Pola:** lista obiektów `DataPoint`.
- **Metody:**
  - Dodawanie punktów danych.
  - Obliczanie sum i średnich dla każdego typu danych.

Klasa `Day`

- Reprezentuje dane dzienne, podzielone na ćwiartki.
- **Pola:** cztery obiekty klasy `Quarter`.
- **Metody:**
  - Zarządzanie ćwiartkami.
  - Agregacja danych dziennych.

Klasa `Tree`

- Hierarchiczna struktura przechowująca dane w układzie: rok → miesiąc → dzień → ćwiartka.
- **Metody:**
  - Dodawanie danych do odpowiednich węzłów drzewa.
  - Iteracja po węzłach za pomocą wzorca projektowego *iterator*.
  - Wyszukiwanie i porównywanie danych w przedziałach czasowych.

3.3. Opis działania aplikacji

Aplikacja ma za zadanie wczytać plik CSV, przeprowadzić analizę danych oraz umożliwić użytkownikowi wykonywanie różnych operacji.

Założenia:

- Plik CSV zawiera dane pomiarowe, z których każda linia reprezentuje jeden punkt danych.
- Pierwsza linia w pliku jest linią informacyjną i jest pomijana podczas analizy.
- Dane mogą zawierać błędne linie, które należy odrzucić i zapisać w plikach logów.
- Czas w pliku zmienia się co 15 minut, a dane muszą być posortowane chronologicznie.

Algorytmy:

- **Sortowanie:** Dane w ćwiartkach są sortowane według czasu (godzina i minuta) przy użyciu algorytmu `std::sort`.
- **Iterator:** Wzorzec projektowy pozwalający na iterację po węzłach drzewa.
- **Wyszukiwanie binarne:** Umożliwia szybkie wyszukiwanie danych w przedziałach czasowych.

3.4. Wczytywanie pliku CSV

- Linie zawierające błędy są zapisywane w plikach `log_data_godzina.txt` i `log_error_data_godzina.txt`.
- Każda poprawna linia tworzy obiekt `DataPoint` i jest dodawana do drzewa danych.
- Podsumowanie wczytanych danych (liczba poprawnych i błędnych rekordów) jest wypisywane na ekranie.

3.5. Funkcje analizy danych

Aplikacja umożliwia wykonanie następujących operacji:

- Obliczanie sumy i średniej dla dowolnych przedziałów czasowych.
- Porównywanie wartości (np. autokonsumpcji) między dwoma przedziałami.
- Wyszukiwanie rekordów spełniających określony warunek z tolerancją.
- Eksport i import danych do/z plików binarnych.

4. Implementacja

W listingu 1 przedstawiono implementację funkcji głównej programu, której zadaniem jest odczyt danych z pliku CSV, parsowanie linii tekstowych na odpowiednie wartości, a następnie dodawanie ich do struktury drzewa za pomocą klasy `Tree`. Program pomija pierwszą linię pliku (nagłówki), a każda kolejna linia jest interpretowana jako nowy obiekt `DataPoint`, który następnie zostaje dynamicznie zaalokowany na stercie i wstawiony do drzewa.

```
int main() {
    std::ifstream file("data.csv");
    std::string line;
    Tree dataTree;
    std::getline(file, line);

    while (std::getline(file, line)) {
        std::istringstream ss(line);
        std::string dateTime, autoConsumption, exportPower,
   importPower, consumption, production;

        std::getline(ss, dateTime, ',');
        std::getline(ss, autoConsumption, ',');
        std::getline(ss, exportPower, ',');
        std::getline(ss, importPower, ',');
        std::getline(ss, consumption, ',');
        std::getline(ss, production, ',');

        DataPoint* point = new DataPoint(dateTime, std::stod(
   autoConsumption), std::stod(exportPower),
            std::stod(importPower), std::stod(consumption), std::
   stod(production));
        dataTree.addDataPoint(point);
    }

    file.close();
    return 0;
}
```

Listing 1. Wczytywanie danych z pliku CSV i budowa drzewa danych

Na listningu 2 przedstawiono konstruktor klasy `DataPoint` oraz metodę `parseDateTime`. Konstruktor przyjmuje łańcuch znaków reprezentujący datę i godzinę, który jest następnie konwertowany na typ `std::time_t` za pomocą metody `parseDateTime`. Pozostałe argumenty, takie jak `autoConsumption`, `exportPower`, `importPower`, `consumption`

i `production`, są używane do inicjalizacji odpowiednich zmiennych członkowskich obiektu.

```
1
2 DataPoint::DataPoint(const std::string& dateTimeStr, double
    autoConsumption, double exportPower,
3     double importPower, double consumption, double production)
4     : autoConsumption(autoConsumption), exportPower(exportPower),
5     importPower(importPower), consumption(consumption), production(
    production) {
6     dateTime = parseDateTime(dateTimeStr);
7 }
8
9 std::time_t DataPoint::parseDateTime(const std::string& dateTimeStr
    ) {
10     std::tm tm = {};
11     std::istringstream ss(dateTimeStr);
12     ss >> std::get_time(&tm, "%Y-%m-%d %H:%M:%S");
13     return std::mktime(&tm);
14 }
```

Listing 2. Konstruktor klasy `DataPoint` oraz metoda `parseDateTime`

Na listningu 3 przedstawiono definicję klasy `DataPoint`, która reprezentuje punkt danych z wartościami związanymi z konsumpcją energii oraz produkcją. Klasa zawiera konstruktor do inicjalizacji tych wartości, oraz metodę statyczną `parseDateTime` do konwersji łańcucha znaków na typ `std::time_t`.

```
1 class DataPoint {
2 public:
3     std::time_t dateTime;
4     double autoConsumption;
5     double exportPower;
6     double importPower;
7     double consumption;
8     double production;
9
10     DataPoint(const std::string& dateTimeStr, double
    autoConsumption, double exportPower,
11         double importPower, double consumption, double production);
12
13     static std::time_t parseDateTime(const std::string& dateTimeStr
    );
14 };
```

Listing 3. Definicja klasy `DataPoint`

Na listningu 4 przedstawiono metody klasy `Day`, które umożliwiają dodawanie punktów danych do odpowiednich kwartałów dnia na podstawie godziny. Metoda `addDataPoint` oblicza, do którego kwartału dnia należy dany punkt danych, a następnie dodaje go do odpowiedniego kwartału. Funkcja `getQuarterIndex` oblicza indeks kwartału na podstawie godziny z `std::time_t`.

```
#include "Day.h"

void Day::addDataPoint(DataPoint* point) {
    int quarterIndex = getQuarterIndex(point->dateTime);
    quarters[quarterIndex].addDataPoint(point);
}

int Day::getQuarterIndex(const std::time_t& dateTime) {
    std::tm* tm = std::localtime(&dateTime);
    int hour = tm->tm_hour;

    if (hour < 6) return 0;
    if (hour < 12) return 1;
    if (hour < 18) return 2;
    return 3;
}
```

Listing 4. Metody klasy `Day`: `addDataPoint` i `getQuarterIndex`

Na listningu 5 przedstawiono definicję klasy `Day`, która reprezentuje dzień i zawiera tablicę `quarters` z czterema obiektami typu `Quarter`. Klasa ta udostępnia metody do dodawania punktów danych oraz obliczania indeksu kwartału na podstawie daty.

```
class Day {
public:
    Quarter quarters[4];

    void addDataPoint(DataPoint* point);
    int getQuarterIndex(const std::time_t& dateTime);
};
```

Listing 5. Definicja klasy `Day`

Na listningu 6 przedstawiono metodę `addDataPoint` klasy `Tree`, która dodaje punkt danych do odpowiedniego roku. Metoda ta pobiera rok z daty i dodaje punkt danych do odpowiedniego obiektu w mapie `years`, której kluczem jest rok.

```
#include "Tree.h"

void Tree::addDataPoint(DataPoint* point) {
    std::tm* tm = std::localtime(&point->dateTime);
```

```
    int year = tm->tm_year + 1900;
    years[year].addDataPoint(point);
}
```

Listing 6. Metoda `addDataPoint` klasy `Tree`

Na listningu 7 przedstawiono definicję klasy `Tree`, która reprezentuje strukturę przechowującą dane zorganizowane według lat. Klasa zawiera mapę `years`, która mapuje rok (typ `int`) na obiekt klasy `Year`. Metoda `addDataPoint` dodaje punkt danych do odpowiedniego roku.

```
class Tree {
public:
    std::map<int, Year> years;

    void addDataPoint(DataPoint* point);
};
```

Listing 7. Definicja klasy `Tree`

Na listningu 8 przedstawiono metodę `addDataPoint` klasy `Year`, która dodaje punkt danych do odpowiedniego miesiąca. Metoda ta pobiera miesiąc z daty i dodaje punkt danych do odpowiedniego obiektu w mapie `months`, której kluczem jest miesiąc.

```
#include "Year.h"

void Year::addDataPoint(DataPoint* point) {
    std::tm* tm = std::localtime(&point->dateTime);
    int month = tm->tm_mon + 1;
    months[month].addDataPoint(point);
}
```

Listing 8. Metoda `addDataPoint` klasy `Year`

Na listningu 9 przedstawiono definicję klasy `Year`, która przechowuje dane zorganizowane według miesięcy. Klasa zawiera mapę `months`, która mapuje numer miesiąca (typ `int`) na obiekt klasy `Month`. Metoda `addDataPoint` dodaje punkt danych do odpowiedniego miesiąca.

```
class Year {
public:
    std::map<int, Month> months;

    void addDataPoint(DataPoint* point);
};
```

Listing 9. Definicja klasy `Year`

Na listningu 10 przedstawiono metody klasy `Quarter`: `addDataPoint` oraz `sortData`. Metoda `addDataPoint` dodaje punkt danych do wektora `dataPoints`, natomiast `sortData` sortuje te punkty według daty.

```
#include "Quarter.h"

void Quarter::addDataPoint(DataPoint* point) {
    dataPoints.push_back(point);
}

void Quarter::sortData() {
    std::sort(dataPoints.begin(), dataPoints.end(), [](DataPoint* a
    , DataPoint* b) {
        return a->dateTime < b->dateTime;
        });
}
```

Listing 10. Metody klasy `Quarter`: `addDataPoint` i `sortData`

Na listningu 11 przedstawiono definicję klasy `Month`, która przechowuje dane zorganizowane według dni. Klasa zawiera mapę `days`, która mapuje numer dnia (typ `int`) na obiekt klasy `Day`. Metoda `addDataPoint` dodaje punkt danych do odpowiedniego dnia.

```
class Month {
public:
    std::map<int, Day> days;

    void addDataPoint(DataPoint* point);
};
```

Listing 11. Definicja klasy `Month`

Na listningu 12 przedstawiono definicję klasy `DateTime`, która reprezentuje datę i godzinę. Klasa przechowuje dzień, miesiąc, rok, godzinę oraz minutę. Zawiera metody do pobierania poszczególnych elementów daty i godziny, a także metodę `ToString`, która zwraca datę i godzinę w sformatowanej postaci.

```
class DateTime {
public:
    int _day, _month, _year, _hour, _minute;

    DateTime(int day, int month, int year, int hour, int minute);
    int GetDay() const;
    int GetMonth() const;
    int GetYear() const;
    int GetHour() const;
```

```
10      int GetMinute() const;
11      std::string ToString() const;
12  };
```

Listing 12. Definicja klasy `DateTime`

Na listningu 13 przedstawiono definicję klasy `Data`, która przechowuje informacje o zużyciu i produkcji energii w danym czasie. Klasa zawiera wskaźnik do obiektu `Time`, który reprezentuje czas pomiaru, oraz zmienne przechowujące wartości autokonsumpcji, eksportu, importu, zużycia i produkcji energii (wszystkie w jednostkach [kWh]). Metody klasy umożliwiają dostęp do tych danych, a destruktor dba o zwolnienie pamięci zaalokowanej dla obiektu `Time`.

```
1  class Data {
2  public:
3      Time* _time;
4      double _autoConsumption, _export, _import, _consumption,
       _generation;
5
6      Data(Time* time, double autoConsumption, double exportW, double
       importW, double consumption, double generation);
7      ~Data();
8
9      Time& GetTime() const;
10     double GetAutoConsumption() const;
11     double GetExport() const;
12     double GetImport() const;
13     double GetConsumption() const;
14     double GetGeneration() const;
15 };
```

Listing 13. Definicja klasy `Data`

5. Wnioski

Realizacja projektu polegającego na analizie danych energetycznych z pliku CSV i ich przetwarzaniu przy użyciu struktur danych w C++ dostarczyła wielu cennych wniosków i pozwoliła na pogłębienie wiedzy w zakresie zaawansowanego programowania obiektowego oraz analizy danych. W poniższym rozdziale przedstawiono najważniejsze wnioski wynikające z implementacji tego zadania.

Efektywna obsługa danych z pliku CSV

Proces wczytywania danych z pliku CSV z uwzględnieniem błędów w strukturze pliku pozwolił zrozumieć, jak ważne jest zapewnienie odpowiedniej walidacji i obsługi błędów w aplikacjach pracujących z danymi wejściowymi. Mechanizmy filtrowania niepoprawnych rekordów i generowania logów umożliwiły stworzenie niezawodnej aplikacji odpornej na niespodziewane przypadki, takie jak puste linie, powtarzające się wpisy czy niekompletne dane.

Struktura danych oparta na drzewie

Zastosowanie drzewa hierarchicznego, w którym korzeniem jest rok, a kolejne poziomy odpowiadają miesiącom, dniom i ćwiartkom doby, pozwoliło na efektywne grupowanie i organizację danych. Rozwiązanie to zapewniło szybki dostęp do informacji w określonych przedziałach czasowych oraz umożliwiło realizację różnorodnych operacji analitycznych, takich jak obliczanie sum i średnich dla wybranych okresów.

Iterator jako wzorzec projektowy

Zastosowanie wzorca projektowego iterator w implementacji drzewa ułatwiło poruszanie się po strukturze danych oraz realizację funkcji takich jak przeszukiwanie węzłów i wykonywanie operacji analitycznych w łatwy i czytelny sposób. Dzięki temu kod pozostał modularny, a jego rozwijanie i testowanie było znacznie prostsze.

Obsługa niepełnych danych

Uwzględnienie braku ciągłości czasowej w danych wymagało dodatkowych mechanizmów, takich jak interpolacja lub ignorowanie brakujących wpisów podczas analizy. Był to istotny aspekt, który nauczył nas, jak radzić sobie z rzeczywistymi danymi, które często nie spełniają idealnych założeń.

Zastosowanie testów jednostkowych

Implementacja testów jednostkowych z wykorzystaniem biblioteki GoogleTest pozwoliła na szybkie wykrywanie błędów w kodzie oraz upewnienie się, że wszystkie metody działają poprawnie zgodnie z oczekiwaniami. Testy okazały się szczególnie przydatne przy weryfikacji poprawności operacji na drzewie i obsłudze danych wejściowych.

Generowanie logów i podsumowanie

Automatyczne generowanie plików logów poprawnych i niepoprawnych rekordów pozwoliło na bieżące monitorowanie stanu aplikacji oraz ułatwiło analizę problemów występujących podczas wczytywania danych. Podsumowanie procesu wczytywania w postaci liczby poprawnych i niepoprawnych rekordów dostarczyło użytkownikowi informacji zwrotnej o jakości danych wejściowych.

Podsumowanie całości projektu

Projekt stanowił wyzwanie, które pozwoliło na praktyczne wykorzystanie wiedzy z zakresu programowania obiektowego, zarządzania danymi i wzorców projektowych. Stworzenie programu o wysokiej niezawodności i funkcjonalności umożliwiło rozwój umiejętności projektowania aplikacji przeznaczonych do analizy danych rzeczywistych. Wykorzystanie nowoczesnych narzędzi, takich jak GoogleTest i GitHub, dodatkowo usprawniło proces tworzenia i zarządzania projektem

Bibliografia

[1] *Visual Studio*. URL: https://pl.wikipedia.org/wiki/Microsoft_Visual_Studio.

[2] *GitHub*. URL: https://en.wikipedia.org/wiki/GitHub.

[3] *GoogleTest*. URL: https://en.wikipedia.org/wiki/Google_Test.

[4] *Doxygen*. URL: https://pl.wikipedia.org/wiki/Doxygen.

[5] *Overleaf*. URL: https://en.wikipedia.org/wiki/Overleaf.

Spis rysunków

Spis tabel

Spis listingów