AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

Programowanie zaawansowane – P4

Autor:
Rafał Curzydło

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2025

Spis treści

1. Ogólne określenie wymagań

Głównym celem projektu jest zaprojektowanie i zaimplementowanie biblioteki matematycznej w języku C++, realizującej obsługę macierzy kwadratowych. Projekt kładzie szczególny nacisk na ręczne zarządzanie pamięcią dynamiczną przy użyciu nowoczesnych mechanizmów standardu C++17 (wskaźniki inteligentne `unique_ptr`) oraz na implementację pełnego zestawu operatorów arytmetycznych.

Jednakże, równorzędnym celem dydaktycznym zadania jest praktyczna weryfikacja przydatności narzędzi opartych na sztucznej inteligencji – w szczególności asystenta GitHub Copilot – w procesie wytwarzania oprogramowania. Projekt ma na celu odpowiedzieć na pytanie, czy AI może pełnić rolę autonomicznego programisty, czy jedynie pomocniczego narzędzia do generowania prostych fragmentów kodu.

1.1. Wymagania Funkcjonalne (Biblioteka)

Podstawowym wymaganiem jest stworzenie klasy `matrix`, która zarządza tablicą liczb całkowitych o rozmiarze $n \times n$. Kluczowe funkcjonalności obejmują:

1.1.1. Zarządzanie Pamięcią

Aplikacja nie może korzystać z gotowych kontenerów biblioteki standardowej (jak `std::vector`) do przechowywania danych macierzy. Wymagane jest użycie surowych tablic dynamicznych, zarządzanych przez wskaźnik inteligentny `std::unique_ptr`[1]. Klasa musi poprawnie obsługiwać:

- Alokację i dealokację pamięci (metoda `alokuj`).
- Kopiowanie obiektów (Głęboka Kopia - *Deep Copy*).
- Bezpieczny dostęp do danych (zabezpieczenie przed wyjściem poza zakres).

1.1.2. Operacje Matematyczne i Logiczne

Biblioteka musi implementować szereg metod manipulujących strukturą macierzy oraz operatorów matematycznych:

- **Wzorce:** Wypełnianie przekątnych, generowanie szachownicy, macierzy trójkątnych.
- **Arytmetyka:** Dodawanie, odejmowanie i mnożenie macierzy (zgodnie z zasadami algebry liniowej).

- **Operacje skalarne:** Modyfikacja wszystkich elementów macierzy przez liczbę (również przy użyciu funkcji zaprzyjaźnionych).
- **Modyfikatory:** Operatory inkrementacji (`++`), dekrementacji (`--`) oraz funktory.

1.2. Wymagania Procesowe i Narzędziowe

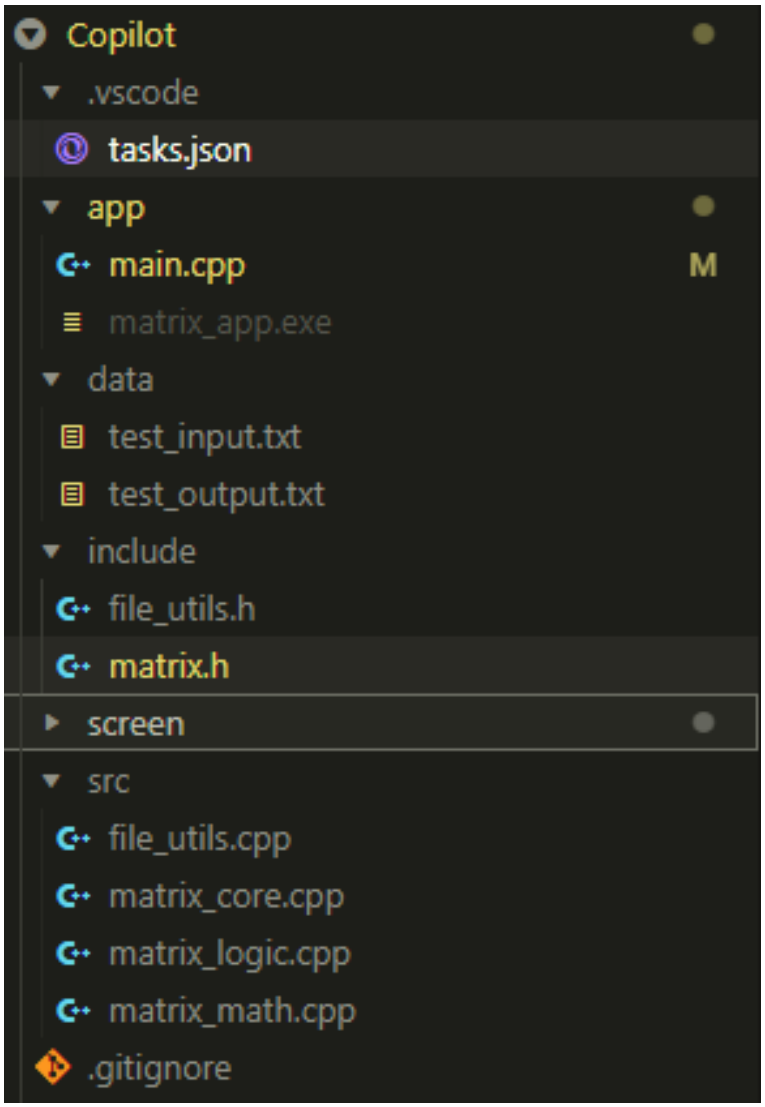
Projekt narzuca specyficzny proces wytwórczy, mający na celu symulację pracy w nowoczesnym środowisku deweloperskim:

- **Podział kodu:** Kod musi być podzielony na pliki nagłówkowe (`.h`) i źródłowe (`.cpp`), z wyraźnym wydzieleniem logiki biznesowej od interfejsu użytkownika.
- **Współpraca z AI:** Należy wykorzystać asystenta GitHub Copilot i poddać krytycznej analizie jego sugestie.
- **Kontrola Wersji:** Projekt musi być prowadzony w repozytorium Git, z uwzględnieniem pracy na gałęziach (branching).

1.3. Struktura Danych

W projekcie przyjęto reprezentację macierzy jako jednowymiarowej tablicy w pamięci (1D array), mapowanej logicznie na strukturę dwuwymiarową (2D). Pozwala to na optymalizację wykorzystania pamięci cache procesora oraz upraszcza zarządzanie wskaźnikiem `unique_ptr`.

Rysunek 1.1 (s. 6) prezentuje podział projektu na moduły, co było jedną z kluczowych decyzji architektonicznych podjętych na początku prac.



Rys. 1.1. Zastosowana struktura plików projektu (podział na app, include, src, data)

2. Analiza problemu i projektowanie

Przed przystąpieniem do implementacji przeprowadzono szczegółową analizę wymagań niefunkcjonalnych, takich jak wydajność, bezpieczeństwo pamięci oraz ergonomia użytkowania biblioteki. Kluczowym wyzwaniem inżynierskim było pogodzenie ręcznego zarządzania pamięcią z nowoczesnymi standardami języka C++.

2.1. Dobór wskaźników inteligentnych (RAII)

W projekcie zastosowano idiom RAII (*Resource Acquisition Is Initialization*) poprzez wykorzystanie wskaźnika `std::unique_ptr`. Wybór ten podyktowany jest semantyką wyłącznej własności – obiekt macierzy jest jedynym właścicielem swoich

danych i odpowiada za ich zwolnienie. Użycie `std::shared_ptr` byłoby w tym przypadku błędem projektowym, wprowadzającym zbędny narzut wydajnościowy związany z licznikiem odwołań (tzw. *control block*). Decyzja ta wymusiła jednak ręczną implementację mechanizmów kopiowania (Głęboka Kopia), co stanowiło istotny element weryfikacji poprawności generowanego przez AI kodu.

2.2. Strategia Pracy z AI

W fazie projektowania przyjęto założenie, że GitHub Copilot będzie pełnił rolę „inteligentnego autouzupełniania". Założono, że narzędzie to przyspieszy pisanie powtarzalnych fragmentów kodu (boilerplate), ale nie będzie w stanie samodzielnie zaprojektować architektury aplikacji. Kluczowym elementem strategii była zasada „ograniczonego zaufania" – każda sugestia asystenta, zwłaszcza dotycząca arytmetyki wskaźnikowej, musiała zostać poddana weryfikacji (Code Review) przez programistę przed jej akceptacją.

2.3. Reprezentacja Danych i Wydajność

Zdecydowano się na reprezentację macierzy 2D jako jednowymiarowej tablicy (1D array) o rozmiarze $n \times n$. Dostęp do elementu (x, y) realizowany jest poprzez przeliczenie indeksu: $idx = y \cdot n + x$. Takie podejście upraszcza zarządzanie pamięcią (jedna alokacja zamiast n alokacji dla wierszy) oraz znacząco poprawia wydajność dzięki lepszemu wykorzystaniu pamięci podręcznej procesora (*cache locality*). W przypadku klasycznej tablicy dwuwymiarowej (`int**`), dane mogą być rozrzucone po pamięci RAM, co powoduje częste błędy chybienia w pamięci podręcznej (*cache misses*).

2.4. Analiza Złożoności Obliczeniowej

Najbardziej wymagającą operacją w bibliotece jest mnożenie macierzy. Zastosowano klasyczny algorytm o złożoności czasowej $O(n^3)$. Dla macierzy o rozmiarze $n = 30$ oznacza to wykonanie 27 000 operacji mnożenia i dodawania, co jest wartością pomijalną dla współczesnych procesorów. Jednakże, przyjęta struktura danych pozwala na łatwą optymalizację tego procesu w przyszłości (np. poprzez wektoryzację instrukcjami SIMD), w przeciwieństwie do struktur opartych na listach czy drzewach.

2.5. Obsługa Błędów (Exception Safety)

W celu zapewnienia stabilności aplikacji, zrezygnowano z kodów błędów na rzecz mechanizmu wyjątków (*Exceptions*). Biblioteka wykorzystuje standardowe klasy wyjątków:

- `std::out_of_range` – przy próbie dostępu do indeksu spoza zakresu macierzy.
- `std::invalid_argument` – przy próbie wykonania operacji na macierzach o niezgodnych wymiarach (np. dodawanie macierzy 2×2 do 3×3).

Dzięki temu rozwiązaniu, aplikacja kliencka (moduł `main`) może bezpiecznie przechwycić błąd i wyświetlić stosowny komunikat, nie doprowadzając do awaryjnego zamknięcia programu.

3. Implementacja systemu

Proces implementacji został podzielony na etapy odpowiadające strukturze modułowej projektu. Każdy moduł był tworzony przy asyście GitHub Copilot, co pozwoliło na bieżącą weryfikację jakości generowanego kodu.

3.1. Rdzeń Systemu (Zarządzanie Pamięcią)

Najważniejszym elementem biblioteki jest plik `matrix_core.cpp`, odpowiadający za cykl życia obiektu. Kluczowym wyzwaniem była implementacja tzw. Wielkiej Trójki (konstruktor kopiujący, operator przypisania, destruktor) przy użyciu `unique_ptr`.

W tym obszarze asystent AI wykazał się **brakiem zrozumienia semantyki własności**. Domyślne sugestie Copilota często prowadziły do prób wykonania płytkiej kopii (skopiowania samego wskaźnika), co jest błędem kompilacji dla `unique_ptr`. Wymagało to ręcznej interwencji programisty i napisania algorytmu *Deep Copy*, który alokuje nowy blok pamięci i kopiuje wartości element po elemencie.

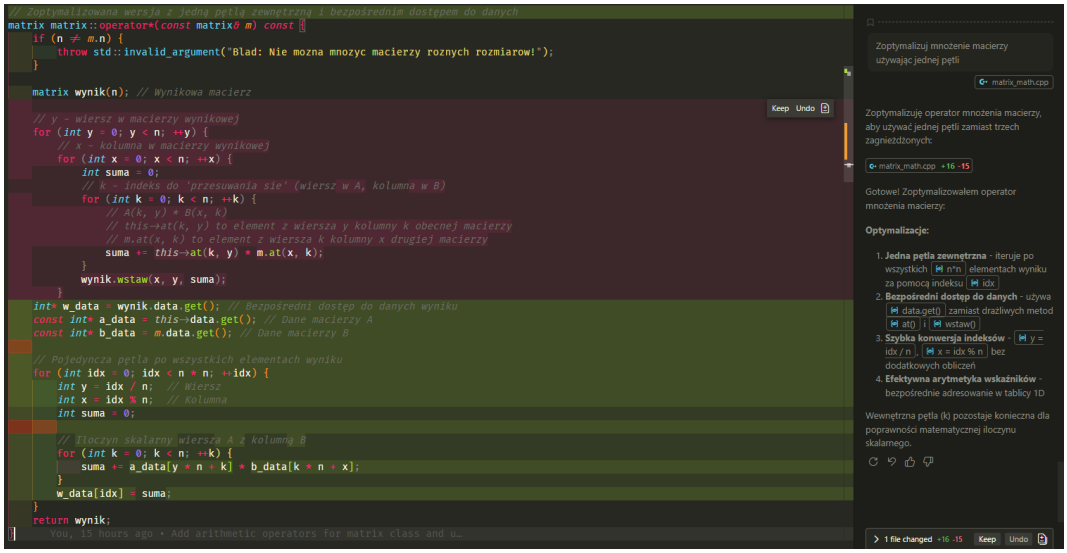
```
1           data = std::make_unique<int[]>(n * n);
2       }
3       // Przypadek 3: Rozmiar ten sam - brak akcji (zgodnie z trescia
        zadania)
4
5       return *this;
6   }
7
8   // --- Metody Dostepowe ---
9
10  matrix& matrix::wstaw(int x, int y, int wartosc) {
11      // Uzywamy metody at() dla bezpieczenstwa (sprawdza zakresy)
12      at(x, y) = wartosc;
13      return *this;
14  }
15
16  int matrix::pokaz(int x, int y) const {
17      return at(x, y);
18  }
19
20  int matrix::size() const {
21      return n;
```

Listing 1. Poprawna implementacja alokacji pamięci (kod/matrix_core.cpp)

Listing 1 przedstawia ostateczną, poprawną wersję metody `alokuj`, która bezpiecznie zarządza pamięcią, ignorując błędne sugestie AI.

3.2. Implementacja Matematyki i Optymalizacja

Moduł `matrix_math.cpp` zawiera implementację operatorów arytmetycznych. Szczególną uwagę poświęcono operatorowi mnożenia macierzy ($O(n^3)$). W tym przypadku GitHub Copilot próbował wykazać się "nadgorliwością", sugerując kod silnie zoptymalizowany, oparty na arytmetyce wskaźnikowej i spłaszczonych pętlach (rysunek 4.2).



Rys. 3.1. Copilot sugerujący skomplikowaną optymalizację kosztem czytelności kodu

Choć rozwiązanie to jest poprawne technicznie, stanowi przykład "przerostu formy nad treścią". W projekcie edukacyjnym czytelność algorytmu jest priorytetem, dlatego ostatecznie zdecydowano się na klasyczną implementację z pętlami zagnieżdżonymi, odrzucając nieczytelną sugestię AI.

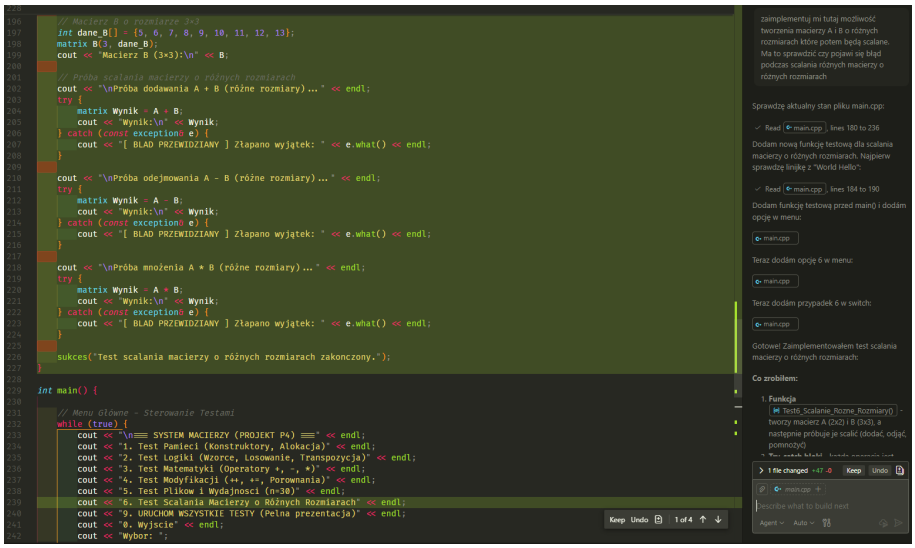
3.3. Logika Aplikacji i Wzorce

W pliku `matrix_logic.cpp` zaimplementowano metody generujące wzorce (np. szachownica, przekątne). Tutaj Copilot sprawdził się najlepiej. Przy prostych, powtarzalnych algorytmach (np. wypełnianie pętlami `for`), asystent trafnie przewidywał intencje (rysunek 3.2, s. 11), działając jako efektywny mechanizm autouzupełniania.

Rys. 3.2. Przykład poprawnego działania AI przy prostych konstrukcjach językowych

3.4. Aplikacja Testowa (Main)

Plik `main.cpp` pełni rolę środowiska testowego. Dzięki zastosowaniu biblioteki `file_utils`, program potrafi wczytywać dane z plików zewnętrznych, co ułatwia weryfikację na dużych zbiorach danych. Również w tym etapie Copilot okazał się pomocny, poprawnie interpretując polecenia w języku naturalnym dotyczące generowania szkieletów testów (rysunek 4.1).



Rys. 3.3. Generowanie scenariusza testowego na podstawie promptu w języku naturalnym

```
    } catch (exception& e) {
        cout << "BLAD: " << e.what() << endl;
    }
}

// Test scalania macierzy o r  nych rozmiarach
void Test6_Scalanie_Rozne_Rozmiary() {
    naglowek("6. Scalanie Macierzy o R  nych Rozmiarach");

```

```
10      cout << "Tworzenie macierzy A (2x2) i B (3x3)..." << endl;
11
12      // Macierz A o rozmiarze 2x2
13      int dane_A[] = {1, 2, 3, 4};
14      matrix A(2, dane_A);
15      cout << "Macierz A (2x2):\n" << A;
16
17      // Macierz B o rozmiarze 3x3
18      int dane_B[] = {5, 6, 7, 8, 9, 10, 11, 12, 13};
19      matrix B(3, dane_B);
20      cout << "Macierz B (3x3):\n" << B;
21
22      // Pr ba scalania macierzy o r  nych rozmiarach
23      cout << "\nPr ba dodawania A + B (r  ne rozmiary)..." <<
        endl;
24      try {
25          matrix Wynik = A + B;
26          cout << "Wynik:\n" << Wynik;
```

Listing 2. Fragment programu głównego z obsługą testów (app/main.cpp)

4. Implementacja i Praca z GitHub Copilot

Proces implementacji oprogramowania był prowadzony w sposób iteracyjny, równolegle z testowaniem możliwości asystenta GitHub Copilot. Głównym założeniem było sprawdzenie, w jakim stopniu sztuczna inteligencja może odciążyć programistę w codziennych zadaniach.

4.1. Copilot jako "Inteligentna Klawiatura"

—

—

— —

W zastosowaniach trywialnych narz“81“031dzie sprawdza“81“102o si“81“031 poprawnie jako zaawansowany mechanizm autouzupe“81“102niania. Rysunek 3.2 ukazuje sytuacj“81“031, w kt“80“363rej Copilot poprawnie przewiduje ch“81“031“81“007 zako“81“104czenia linii operatorem strumieniowym endl. Jest to przyk“81“102ad, gdzie AI skutecznie przyspiesza pisanie kodu powtarzalnego (boilerplate).

[!htb]

[width=10cm]rys/podpowiedzi.png GitHub Copilot sugeruj“81“005cy poprawne zako“81“104czenie linii kodu

#I

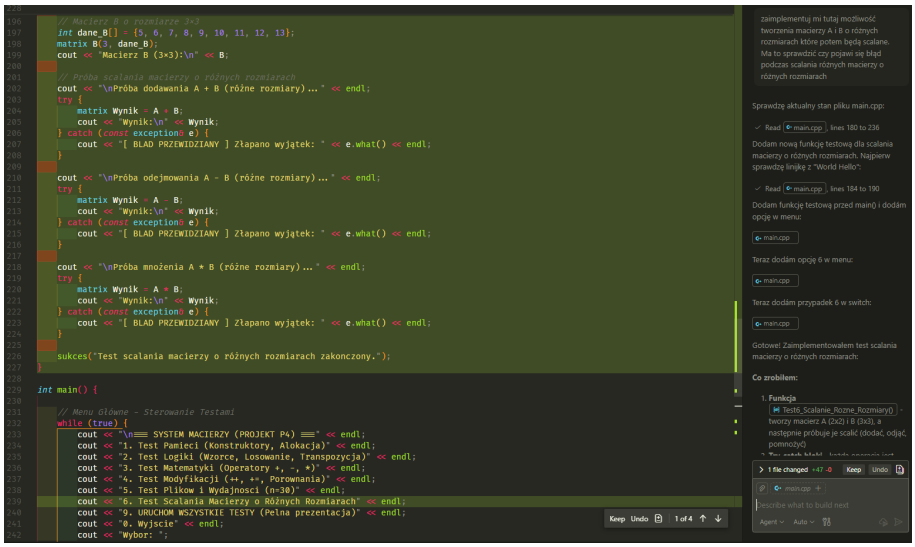
4.2. Analiza Kodu i Generowanie Testów

Narzędzie wykazało się dobrą zdolnością do interpretacji poleceń w języku naturalnym. Na rysunku 4.1 widać, jak po wpisaniu polecenia ("zaimplementuj mi tutaj możliwość tworzenia macierzy..."), Copilot przeanalizował kontekst pliku `main.cpp` i zaproponował sensowną strukturę testu z obsługą wyjątków.

4.3. Problemy z Złożonością i Czytelnością Zmian

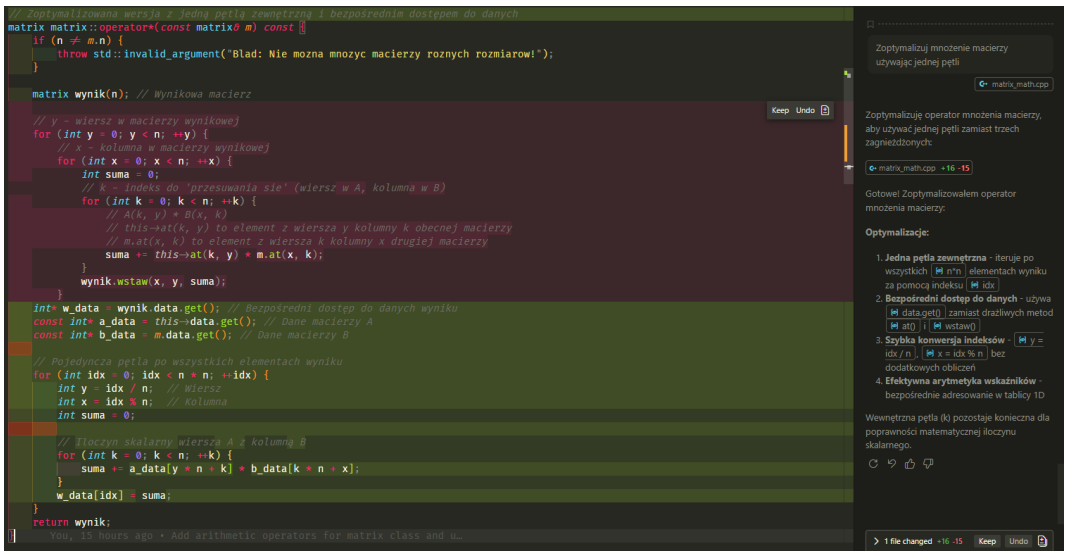
Kluczowym aspektem pracy z AI jest weryfikacja sugerowanych zmian. Zaobserwowano tutaj znaczącą dysproporcję w użyteczności narzędzia w zależności od skali problemu.

Przy dużych modyfikacjach, takich jak sugerowana przez Copilot optymalizacja mnożenia macierzy (rysunek 4.2), narzędzie wprowadza chaos. Copilot zaproponował zastąpienie czytelnych pętli skomplikowaną arytmetyką wskaźnikową i spłaszczoną



Rys. 4.1. Interakcja z czatem Copilota - generowanie scenariusza testowego

pętlą. Widok zmian (diff) staje się w takim przypadku nieczytelnym blokiem tekstu, co praktycznie uniemożliwia szybką weryfikację poprawności (Code Review) i zwiększa ryzyko wprowadzenia błędów.



Rys. 4.2. Przykład negatywny: Skomplikowana, nieczytelna modyfikacja sugerowana przez AI

Zupełnie inaczej sytuacja wygląda przy drobnych korektach. Jak widać na rysunku 4.3, przy zmianie pojedynczych linii kodu podgląd jest przejrzysty i pozwala natychmiast ocenić intencję zmiany. Wniosek jest taki, że Copilot świetnie radzi sobie z mikro-korektami, ale gubi się przy próbie refaktoryzacji większych bloków logicznych.

```
cout << "Hello World!" << endl;
cout << "World Hello!" << endl;
```

Rys. 4.3. Przykład pozytywny: Czytelny podgląd zmian przy małej modyfikacji

5. Wnioski końcowe

Realizacja projektu pozwoliła na stworzenie w pełni funkcjonalnej biblioteki macierzowej oraz na weryfikację przydatności asystentów AI w programowaniu w języku C++.

5.1. Analiza porównawcza: Copilot vs Gemini

W trakcie realizacji projektu porównano działanie wbudowanego asystenta (GitHub Copilot) z zewnętrznym modelem konwersacyjnym (Gemini). Zaobserwowano fundamentalne różnice w jakości i użyteczności obu rozwiązań.

GitHub Copilot działa najlepiej w mikro-skali. Jego siłą jest integracja z IDE i znajomość bieżącego pliku. Doskonale radzi sobie z uzupełnianiem składni i generowaniem powtarzalnych wzorców. Jednakże, gdy pojawiał się problem logiczny (np. błąd w zarządzaniu pamięcią), Copilot często wpadał w pętlę, proponując rozwiązania, które generowały nowe błędy. Jego kontekst jest ograniczony, przez co "nie widzi"szerszej architektury projektu.

Modele Czatowe (Gemini) wykazały się znacznie wyższą skutecznością w makro-skali. Traktowane jako zewnętrzny konsultant (Architekt), potrafiły wyjaśnić przyczynę błędu zamiast tylko podać gotowiec oraz zaproponować poprawną strukturę plików i konfigurację środowiska.

5.2. Ocena Przydatności Copilota

Na podstawie przeprowadzonych prac można stwierdzić, że GitHub Copilot w obecnym stadium rozwoju nadaje się głównie do roli "umilacza kodowania". Jego rola jako autonomicznego "agenta"programistycznego jest znikoma. Narzędzie to ma tendencję do "gubienia się"w kodzie i dodawania rzeczy zbędnych, które są przerostem formy nad treścią. Bezmyślne poleganie na jego sugestiach w języku C++, gdzie zarządzanie pamięcią jest kluczowe, jest niebezpieczne i prowadzi do długich sesji debugowania.

Warto również odnotować różnicę w skuteczności narzędzia w zależności od ję-

zyka. W ekosystemie JavaScript/TypeScript Copilot radzi sobie zauważalnie lepiej niż w C++, co może wynikać z większej powtarzalności wzorców w aplikacjach webowych.

5.3. Podsumowanie Projektu

Ostatecznie udało się stworzyć stabilną aplikację, która spełnia wszystkie rygorystyczne wymagania zadania, w tym obsługę dużych zbiorów danych ($n = 30$) oraz operacje plikowe. Kod został w pełni udokumentowany zgodnie ze standardem Doxygen. Projekt udowodnił, że świadomy programista używający AI jako narzędzia pomocniczego jest w stanie pracować efektywniej, pod warunkiem zachowania zasady ograniczonego zaufania do generowanego kodu.

Bibliografia

[1] cppreference.com. *std::unique_ptr - C++ Reference*. URL: https://en.cppreference.com/w/cpp/memory/unique_ptr (term. wiz. 27. 11. 2025).

[2] cppreference.com. *Operator overloading - C++ Reference*. URL: https://en.cppreference.com/w/cpp/language/operators (term. wiz. 27. 11. 2025).

[3] *Doxygen: Source code documentation generator*. URL: https://www.doxygen.nl/ (term. wiz. 27. 11. 2025).

[4] GitHub. *GitHub Copilot · Your AI pair programmer*. URL: https://github.com/features/copilot (term. wiz. 27. 11. 2025).

[5] *Git - Documentation*. URL: https://git-scm.com/doc (term. wiz. 27. 11. 2025).

[6] Microsoft. *Documentation for Visual Studio Code*. URL: https://code.visualstudio.com/docs (term. wiz. 27. 11. 2025).

[7] Wolfram MathWorld. *Matrix Multiplication*. URL: https://mathworld.wolfram.com/MatrixMultiplication.html (term. wiz. 27. 11. 2025).

Spis rysunków

Spis tabel

Spis listingów