Politechnika Śląska
Wydział Informatyki, Elektroniki i Informatyki

Programowanie Komputerów

GALOPP

autor	**Filip Kudła**
prowadzący	**dr hab. inż., prof. PŚ Roman Starosolski**
rok akademicki	**2023/2024**
kierunek	**informatyka**
rodzaj studiów	**SSI**
semestr	**4**
termin laboratorium	**wtorek, 14:15 - 15:45**
termin oddania sprawozdania	**28.06.24r.**

GALOPP
(Graficzny analizator liniowych obwodów prądu przemiennego)

Program jest aplikacją symulująca obwody prądu przemiennego. Użytkownik może tworzyć własne obwody elektryczne poprzez interaktywne rysowanie ich w aplikacji. Po zakończeniu rysowania obwodu, informacje o jego elementach są przekazywane do programu napisanego w C++, który przeprowadza symulację. Wyniki symulacji są następnie przekazywane z powrotem do aplikacji, gdzie użytkownik może je zobaczyć na ekranie. Logika rysowania jest rozdzielona od logiki analizowania obwodu.

Omówienie najważniejszych klas

1. Element_graficzny

Wirtualna klasa bazowa odpowiedzialna za wizualną reprezentację elementu na ekranie.

Po tej klasie dziedziczą wszystkie możliwe do narysowania elementy, tj. linia (służy do połączenia elementów na ekranie), źródło napięciowe, źródło prądowe, rezystor, cewka, kondensator.

2. ElementManager

Klasa zarządzająca elementem graficznym, posiada logikę umożliwiająca dodawanie elementów do pola roboczego, zarządza kolizjami, a także umożliwia zapisanie i odczyt wszystkich elementów do pliku.

3. Element

Wirtualna klasa bazowa reprezentująca element i jego wartości. Klasa służąca do rozwiązywania obwodów, reprezentant elementu graficznego od strony analitycznej.

Po tej klasie dziedziczą elementy, na których można coś obliczać (wszystkie oprócz linii).

4. Obwód

Klasa zamykająca elementy analityczne w wektor, na którym wywoływane są wszelkie funkcje potrzebne do przeanalizowania obwodu, to jest do obliczenia napięć, prądów itp. dla każdego elementu. Umożliwiają to algorytmy, takie jak metoda potencjałów węzłowych, a także metoda eliminacji Gaussa. Klasa ta również eksportuje wyliczone dane do pliku tekstowego, a także na ekran w celach zobrazowania wykonanej symulacji.

5. Window

Klasa przechowująca okno programu (dziedzicząca po SFML-owym oknie), dodatkowo służy m.in. do renderowania tła programu.

6. Button

Klasa reprezentuje przyciski możliwe do klikania w programie, a także ich wizualne przedstawienie (najechanie, kliknięcie, podświetlenie). Definiuje zespół funkcji przypisanych do przycisku oddelegowanych podczas startu programu.

7. Textbox

Klasa definiująca pola tekstowe (inputy użytkownika). Definiuje logikę stojąca za wprowadzaniem danych do pól, a także konwertuje wpisane wartości w polu do liczby.

Algorytmy

Służące głównie do analizy programu, najważniejsze 2 funkcje:

- coltri() - Funkcja tworzy macierz (układ równań) za pomocą metody Coltriego. Wprowadza do macierzy admitancje każdego potencjału używając metody potencjałów węzłowych. Napotkane gałęzie ze źródłem elektromotorycznym traktuje jako gałęzie z impedancja {1,0} Ω.

- gauss() - Funkcja oblicza macierz za pomocą metody eliminacji Gaussa-Jordana-Crouta. Funkcja sprowadza macierz do postaci trójkątnej (uzyskuje zero nad i pod przekątna macierzy). Zamienia wiersze (równanie) z innym wierszem, w którym występuje większy współczynnik w kolumnie. Metoda ta sprowadza macierz rozszerzona układu równań do postaci bazowej (macierzy jednostkowej). Z tej postaci można wprost odczytać potencjały w węzłach.

Biblioteki

- SFML (Simple and Fast Multimedia Library) – biblioteka zewnętrzna ułatwiająca tworzenie gier i aplikacji multimedialnych. Pozwala na renderowanie i zarządzanie grafiką w oknie.

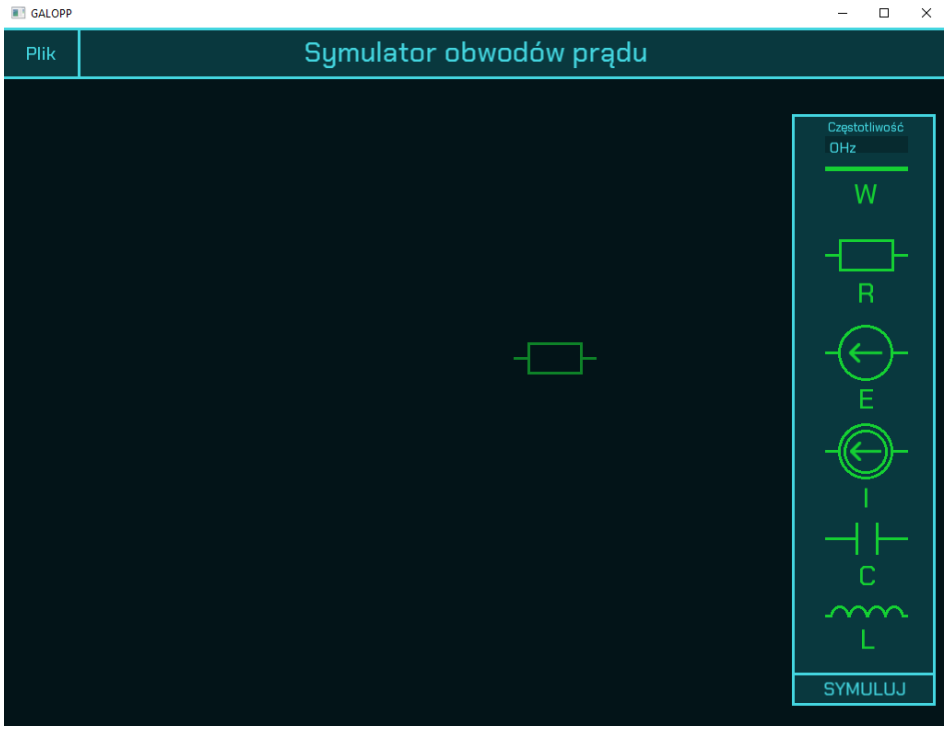
Specyfikacja zewnętrzna

Interfejs użytkownika:

1. Okno aplikacji: W oknie aplikacji użytkownik może rysować obwody elektryczne za pomocą myszy oraz klawiatury.
2. Elementy obwodu: W interfejsie dostępne są różne elementy obwodu, tj. źródła prądu i napięcia, rezystory, cewki i kondensatory. Użytkownik może wybierać elementy i umieszczać je na obszarze roboczym.
3. Interakcja z użytkownikiem: Użytkownik może dodawać elementy obwodu poprzez kliknięcie myszą w komponent, a następnie przeciągnięcie w obszar, aby narysować element. Może dodawać wartości elementom poprzez kliknięcie PPM na element.
4. Symulacja obwodu: Po zakończeniu rysowania obwodu użytkownik może uruchomić symulację, która przekazuje informacje o obwodzie do programu C++, gdzie odbywa się analiza i obliczenia.
5. Wyświetlanie wyników: Po zakończeniu symulacji wyniki są przekazywane z powrotem do aplikacji, gdzie użytkownik może zobaczyć wartości prądów, napięć itp. na ekranie.
6. Dodatkowe funkcje: Interfejs zawiera dodatkowe funkcje, takie jak zapisywanie i wczytywanie obwodów, zmiana parametrów symulacji, czy możliwość eksportu wyników.

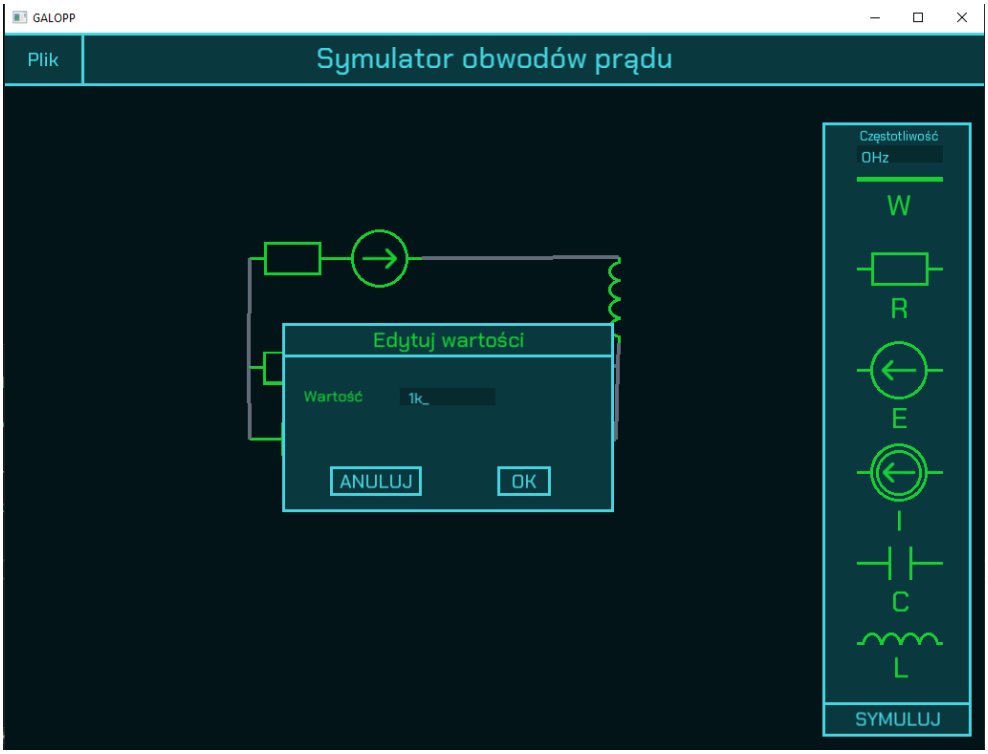
Standardowy przykład działania:

- Po uruchomieniu programu, mamy możliwość doboru elementów. Możemy wybrać je myszką, lub przez odpowiedni przycisk na klawiaturze (podpisany poniżej elementu w menu)



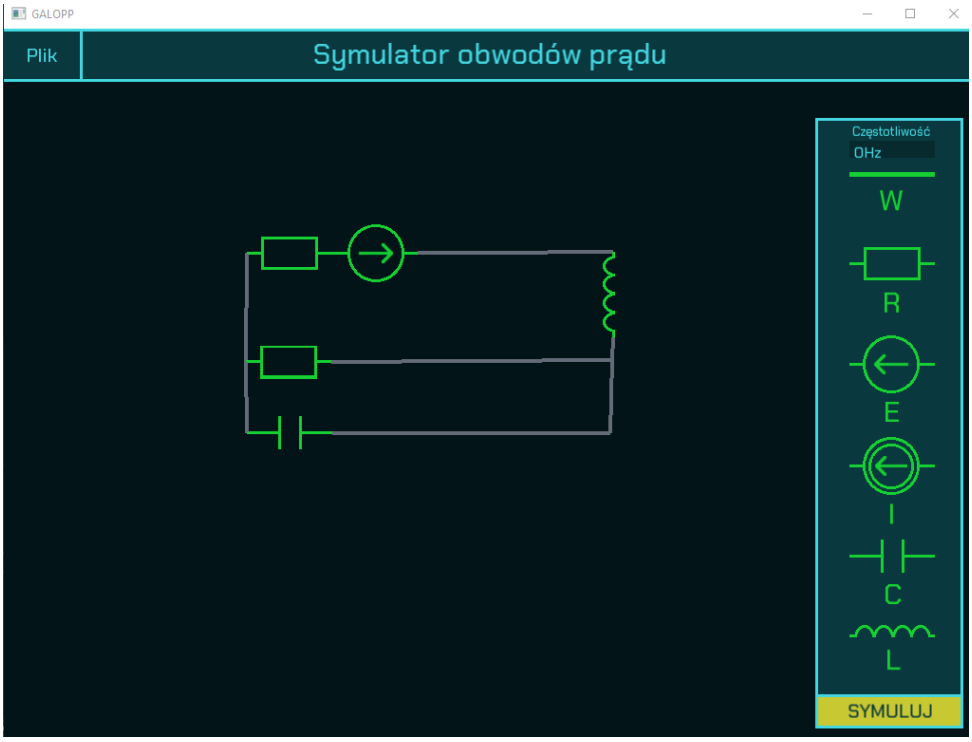
Łączymy elementy ze sobą na ekranie myszką. Możliwość umieszczenia elementu w danym miejscu sygnalizuje jego zielony kolor, w przeciwnym wypadku – czerwony. Anulowanie wyboru klawiszem ESC

- Po połączeniu elementów, możemy edytować ich wartości klikając na nich prawym przyciskiem myszy, wtem pokaże się okno edytowania:

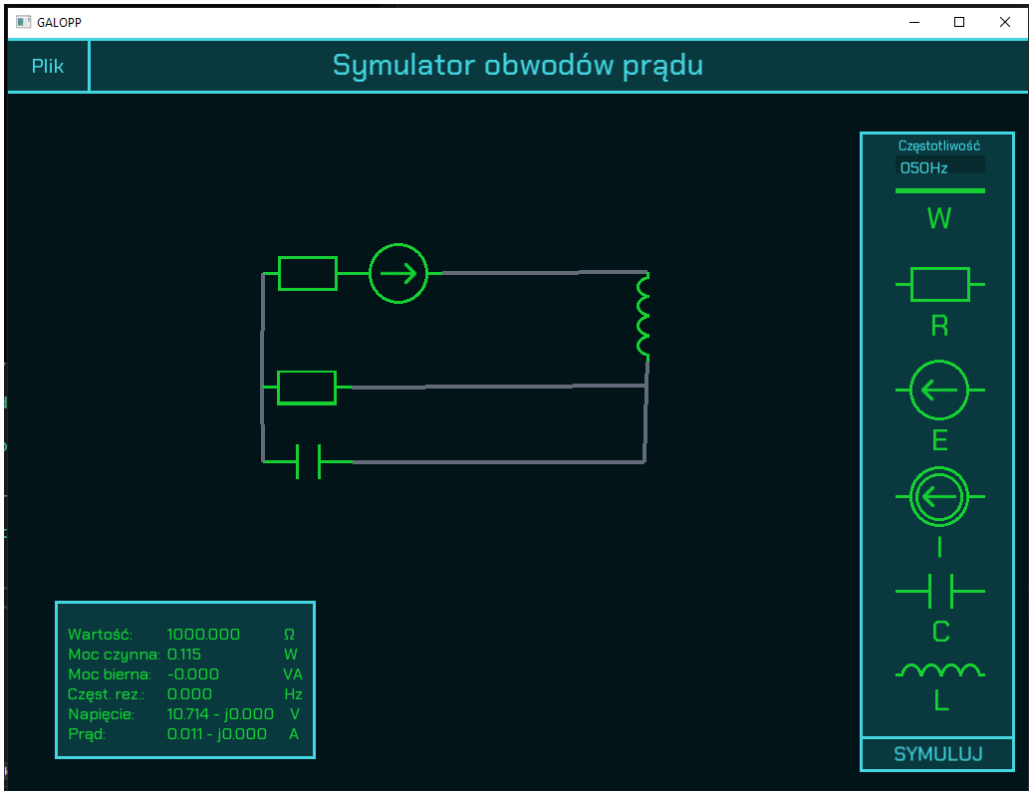


W tym miejscu możemy dodać wartość elementowi (dla źródeł możemy dodatkowo zdefiniować ich przesunięcie fazowe). Potwierdzamy kliknięciem przycisku lub klawiszem Enter.

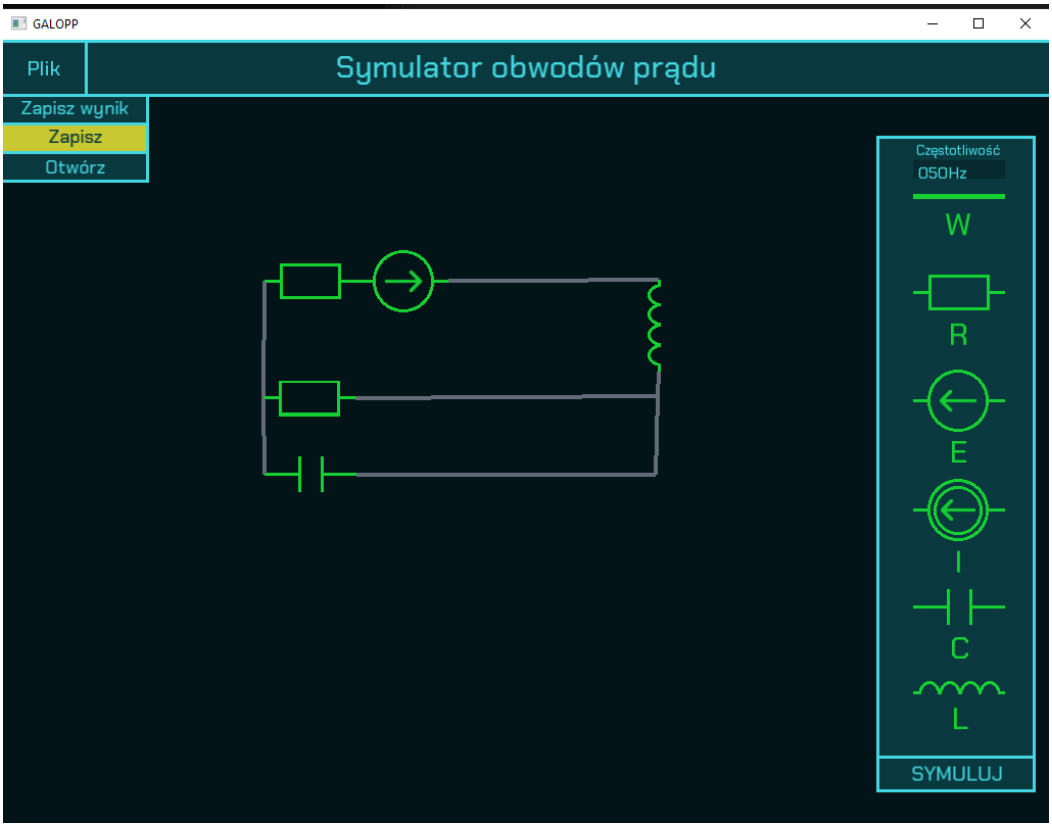
- Po narysowaniu obwodu i dodaniu wartości, jeżeli obwód jest poprawnie skonstruowany, możemy go zasymulować. Jeżeli dodaliśmy elementy AC, należy do poprawnego działania wpisać częstotliwość źródeł



- Po najechaniu na element możemy zaobserwować teraz jego charakterystyki. Wartości prądu i napięcia są domyślnie w formie zespolonej dla każdego elementu



- Dodatkowo możemy zapisać wyniki symulacji (obwodu zasymulowanego obecnie) do pliku tekstowego, zapisać obwód graficzny do pliku, a także odtworzyć go z istniejących plików



Specyfikacja wewnętrzna

Klasy:

- element_graficzny

Znaczenie obiektu: reprezentuje obiekt możliwy do narysowania na ekranie. Jest odpowiednikiem kształtu elementu, na którym można wywoływać różne metody z biblioteki SFML

Powiązania z innymi klasami: ElementManager zarządza tym obiektem

Istotne pola i metody: pola leftNode i rightNode, czyli węzły obiektu (współrzędne);

Metody serializacji i deserializacji danych eksportujące informacje o elemencie na ekranie do pliku tekstowego.

- Element_manager

Znaczenie obiektu: obwód graficzny (narysowany na polu roboczym obwód, możliwy do przeanalizowania).

Powiązania z innymi klasami: przechowuje obiekty elementów graficznych

Istotne pola i metody: id do zidentyfikowania obiektu, wektor obiektów klasy element_graficzny;

Metody dodawania i usuwania elementów do obwodu, sprawdzania kolizji (nakładania się na siebie elementów, sprawdzenia czy są wewnątrz pola roboczego), ustawiania węzłów, ustawiania id elementom, metody zapisu i odczytu z pliku (obwodu graficznego).

- element

Znaczenie obiektu: element możliwy do przeanalizowania

Powiązania z innymi klasami: jest częścią Obwodu (analitycznego)

Istotne pola i metody: pola id do zidentyfikowania elementu, typ, umiejscowienie (po węzłach);

Metody: wyznaczanie impedancji, admitancje, napięcia, prądu, mocy, częstotliwości rezonansowej każdego elementu.

- Obwód

Znaczenie obiektu: obwód analityczny

Powiązania z innymi klasami: przechowuje obiekty elementów, konwertuje wektor elementów graficznych do elementów analitycznych na bazując na ich położeniu i typie.

Istotne pola i metody: wektor obiektów klasy element, częstotliwość, mapa węzłów – nieuporządkowana mapa przechowująca jako klucz współrzędne węzła na ekranie, a jako wartość kolejny numer węzła. Dodatkowo przechowuje zbiór węzłów, macierz potencjałów, mapy potencjałów w węzłach;

Metody konwertowania obwodu graficznego do analitycznego bazując na wektorze graficznych, używa do tego mapy węzłów. Po przekonwertowaniu korzysta z metod tworzących kontener unikalnych węzłów z obwodu, metody dodającej rezystor o wartości -1 za gałęzią z siłą elektromotoryczną, następnie obliczana jest macierz potencjałów za pomocą metody potencjałów węzłowych, a później obliczana za pomocą metody eliminacji Gaussa. Po obliczeniu potencjałów w węzłach obwodu, oblicza napięcie, prąd i moc czynną i bierną dla każdego elementu. Te informacje są później eksportowane do głównej pętli programu, aby były możliwe do wyświetlenia po najechaniu na element.

Struktury danych

WEKTOR

- do przechowywania wskaźników na elementy graficzne
- do przechowywania wskaźników na elementy do analizy
- do tworzenia kontenerów węzłów

MACIERZ

- wektor wektorów, do przechowywania układów równań za pomocą metody potencjałów węzłowych

MAPA

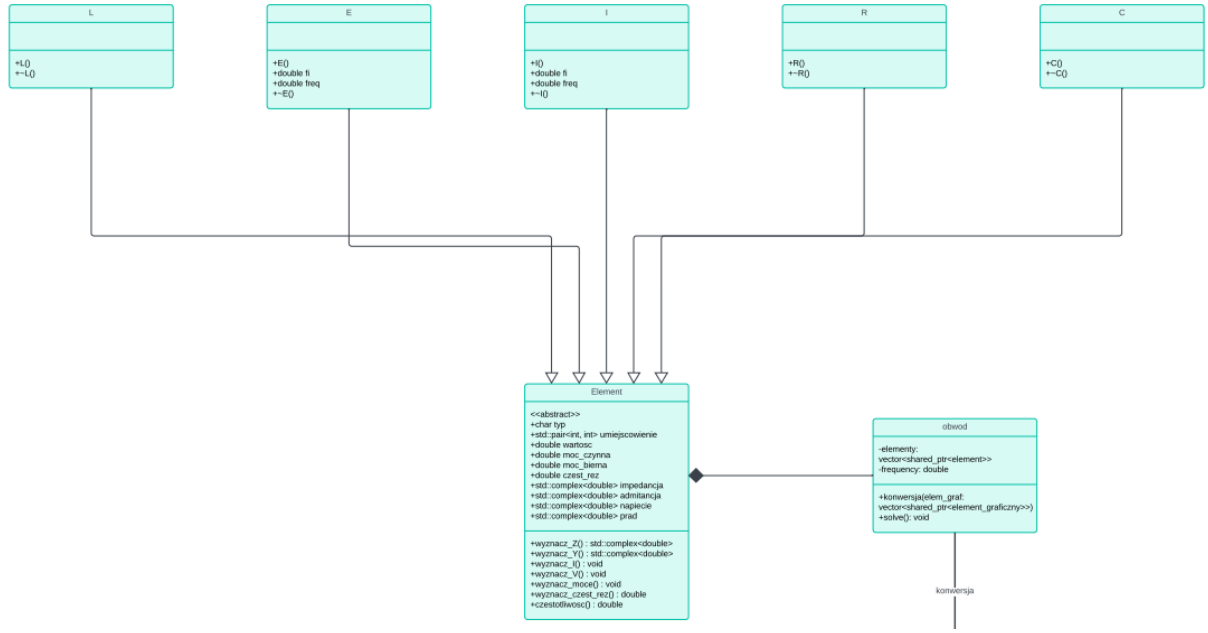
- do potencjału w konkretnym węźle (klucz jest węzłem, wartość zespolona potencjałem)

PARA

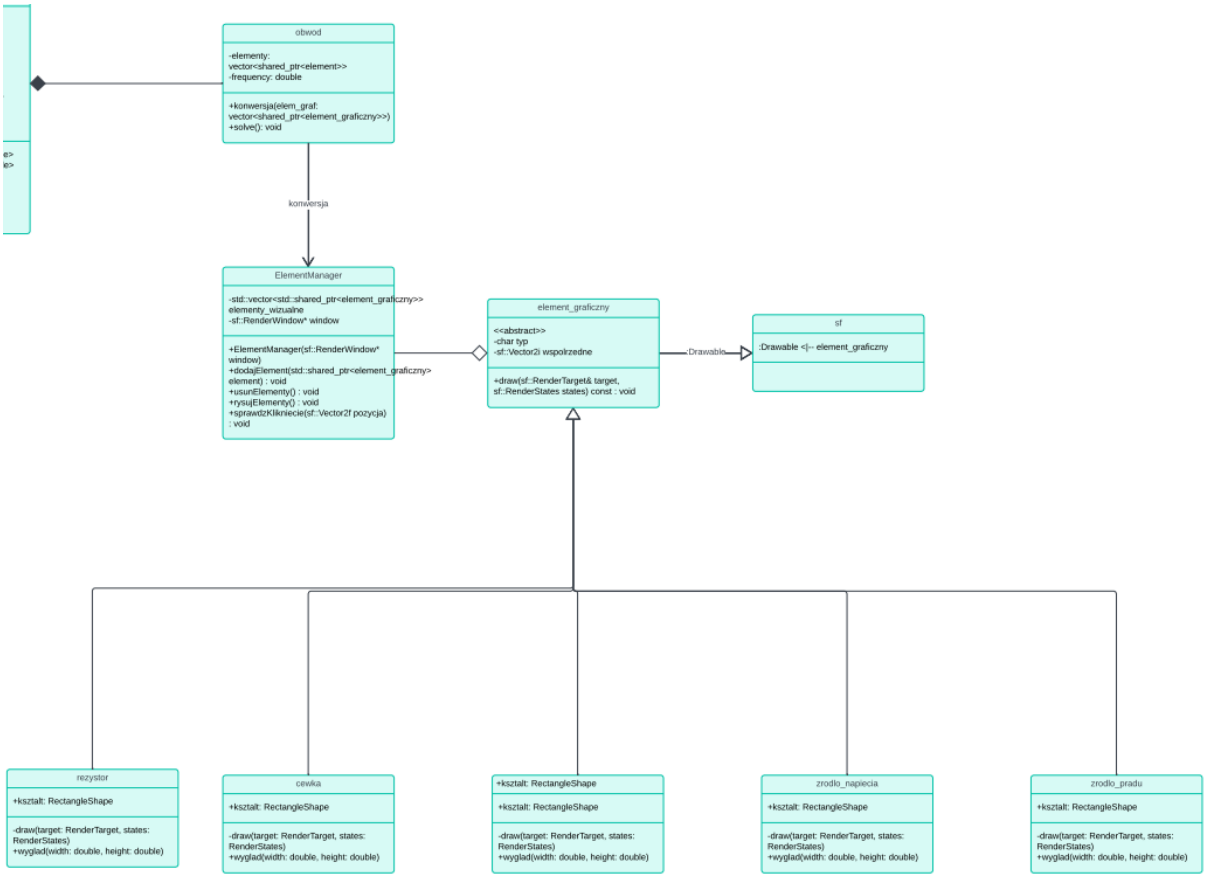
- dla węzłów (graficznie dla współrzędnych, analitycznie dla umiejscowienia w obwodzie)

Diagram hierarchii klas

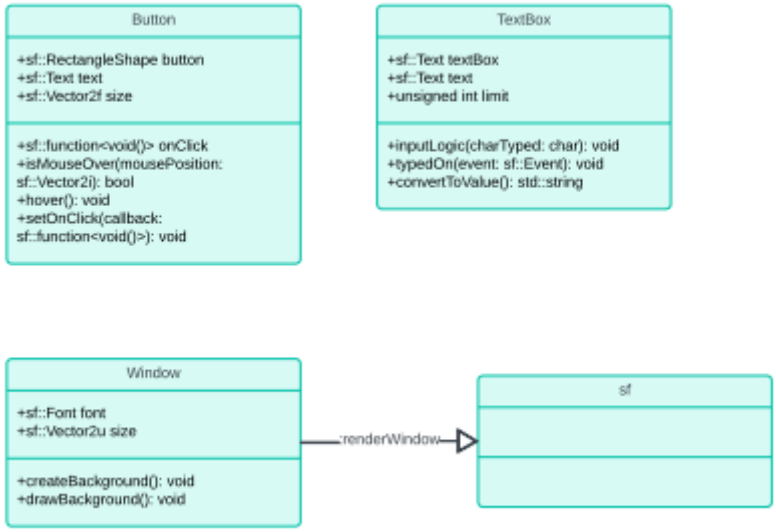
Klasa analityczna element i jej pochodne, powiązanie z klasą Obwód:



Klasa obwód konwertująca wektor elementów graficznych z ElementManager. Klasa element_graficzny i jej pochodne:



Dodatkowe klasy potrzebne do zwizualizowania okna programu i do wprowadzania wartości:



Wykorzystane techniki obiektowe

1. Dziedziczenie

Klasy pochodne od element_graficzny z niej dziedziczą, przykład:

```
class zrodlo_sem : public element_graficzny
{
private:
    void draw(sf::RenderTarget& target, sf::RenderStates states) const;
public:
    zrodlo_sem(const sf::Color& color);
    double fi;
    bool mouse_over(sf::Vector2f& pozycja) const;
    void serialize(std::ostringstream& oss) const;
    void deserialize(std::istringstream& iss);
    std::shared_ptr<element_graficzny> clone() const override;
};
```

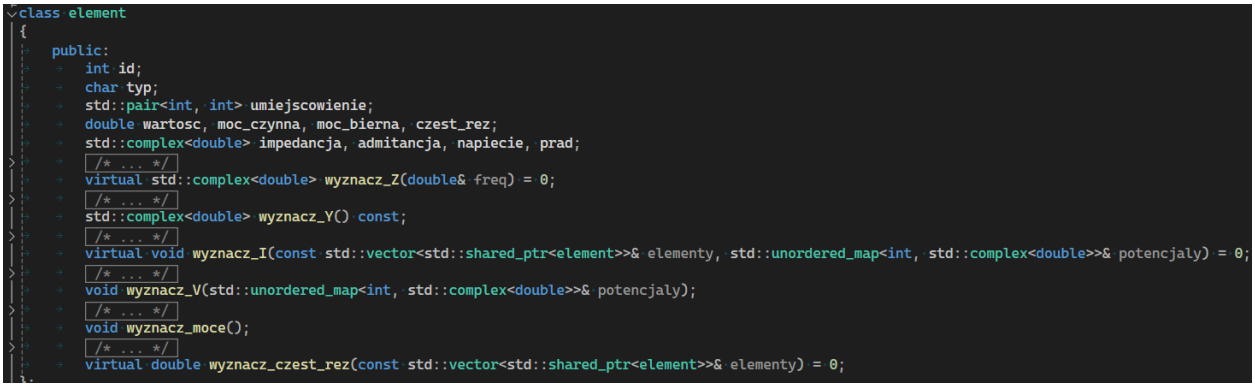
Dziedziczenie występuje również dla elementów analitycznych:

```
/**
@brief Klasa reprezentuje zrodla elektromotoryczne (SEM), jest klasa pochodna elementu.
Posiada dodatkowo zmienne double fi oznaczajaca przesuniecie fazowe zrodla
*/
class E : public element
{
    public:
        double fi;
        /* ... */
        E(int id, char typ, std::pair<int, int> miejsce, double wartosc, double fi);
        /* ... */
        std::complex<double> wyznacz_Z(double& freq);
        /* ... */
        void wyznacz_I(const std::vector<std::shared_ptr<element>>& elementy, std::unordered_map<int, std::complex<double>>& potencjaly);
        /* ... */
        double wyznacz_czest_rez(const std::vector<std::shared_ptr<element>>& elementy);
        /* ... */
        ~E();
};
```

Jest potrzebne, ponieważ każdy element posiada swoją wartość oraz można dla niego obliczyć m.in. prąd, napięcie, admitancje itd.

2. Polimorfizm

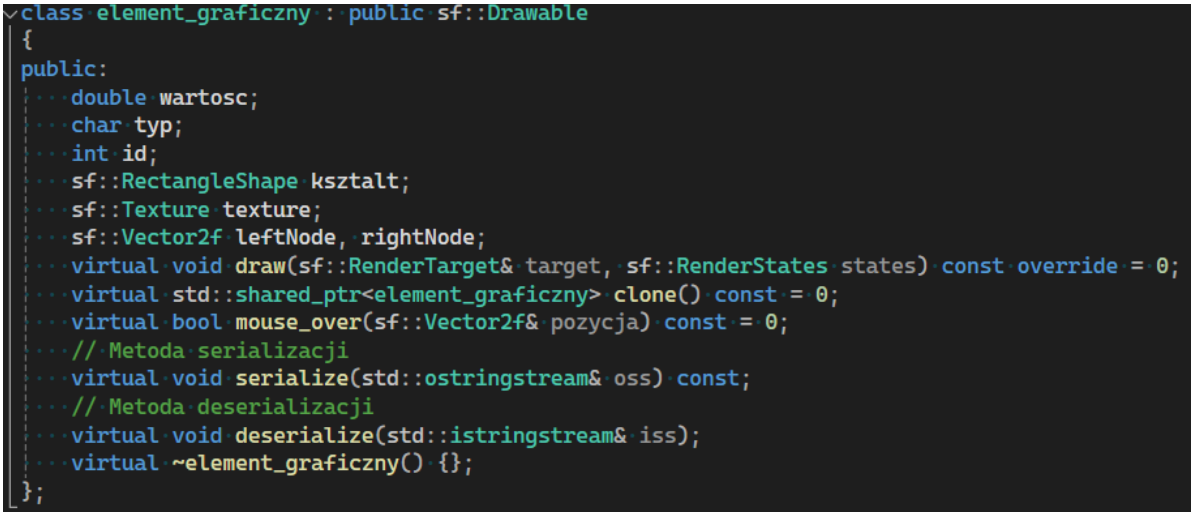
Wykorzystany dla klas elementu analitycznego:



```
class element
{
public:
    int id;
    char typ;
    std::pair<int, int> umiejscowienie;
    double wartosc, moc_czynna, moc_bierna, czest_rez;
    std::complex<double> impedancja, admitancja, napiecie, prad;
    /* ... */
    virtual std::complex<double> wyznacz_Z(double& freq) = 0;
    /* ... */
    std::complex<double> wyznacz_Y() const;
    /* ... */
    virtual void wyznacz_I(const std::vector<std::shared_ptr<element>>& elementy, std::unordered_map<int, std::complex<double>>& potencjaly) = 0;
    /* ... */
    void wyznacz_V(std::unordered_map<int, std::complex<double>>& potencjaly);
    /* ... */
    void wyznacz_moce();
    /* ... */
    virtual double wyznacz_czest_rez(const std::vector<std::shared_ptr<element>>& elementy) = 0;
};
```

Każdy element indywidualnie wyznacza swój prąd na gałęzi, stąd potrzeba polimorfizmu, a także impedancje i częstotliwości rezonansowe.

Polimorfizm dla elementu graficznego:

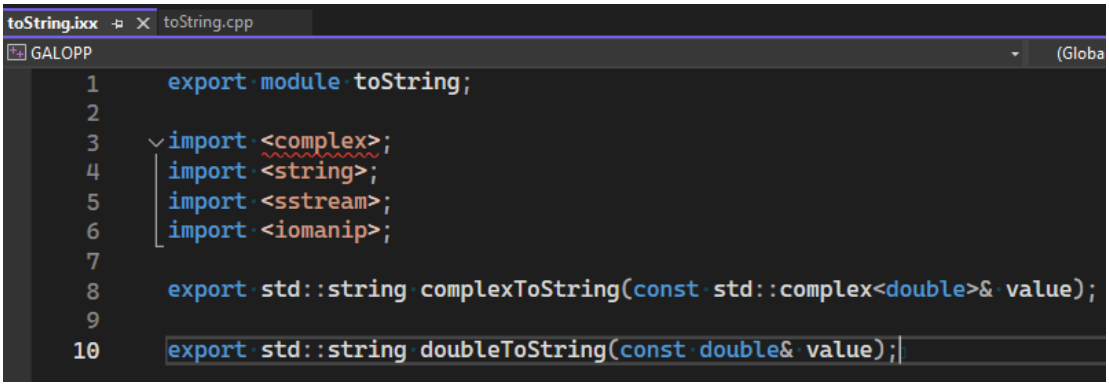


```
class element_graficzny : public sf::Drawable
{
public:
    double wartosc;
    char typ;
    int id;
    sf::RectangleShape ksztalt;
    sf::Texture texture;
    sf::Vector2f leftNode, rightNode;
    virtual void draw(sf::RenderTarget& target, sf::RenderStates states) const override = 0;
    virtual std::shared_ptr<element_graficzny> clone() const = 0;
    virtual bool mouse_over(sf::Vector2f& pozycja) const = 0;
    // Metoda serializacji
    virtual void serialize(std::ostringstream& oss) const;
    // Metoda deserializacji
    virtual void deserialize(std::istringstream& iss);
    virtual ~element_graficzny() {};
};
```

Klasy pochodne implementują m.in. sposób sprawdzania kolizji, metody serializację itd.

3. Moduły

Moduły zostały wykorzystane przy konwersji wyników programu, funkcje zmieniające liczby do łańcuchu znaków zostały zaimportowane w funkcji main.



toString.ixx | toString.cpp

GALOPP

```
1   export module toString;
2
3   import <complex>;
4   import <string>;
5   import <sstream>;
6   import <iomanip>;
7
8   export std::string complexToString(const std::complex<double>& value);
9
10  export std::string doubleToString(const double& value);
```

Implementacja w cpp:

toString.ixx toString.cpp

GALOPP (Globalny zasięg)

```cpp
module toString;

std::string complexToString(const std::complex<double>& value)
{
    std::ostringstream oss;
    oss << std::fixed << std::setprecision(3);
    oss << value.real() << (value.imag() < 0 ? " - j" : " + j") << std::abs(value.imag());
    return oss.str();
}

std::string doubleToString(const double& value)
{
    std::ostringstream oss;
    oss << std::fixed << std::setprecision(3);
    oss << value;
    return oss.str();
}
```

4. Regex

Służą do obsługi textboxów, można wpisywać wartości z przyrostkami ‘Mkmun’ – odpowiednio milion, kilo, mili, mikro, nano, w celu szybszego wypełniania pożądanych wartości elementom.

```cpp
// Funkcja do wpisywania podczas petli
void Textbox::typedOn(sf::Event input)
{
    if (isSelected)
    {
        int charTyped = input.text.unicode;
        std::regex digits("[0-9.0-9]");
        std::regex singleLetter("[Mkmun]");
        std::string charStr(1, static_cast<char>(charTyped));
        if (std::regex_match(charStr, digits) || charTyped == DELETE_KEY ||
            (std::regex_match(charStr, singleLetter) && !endsWithLetter()))
        {
            if (hasLimit)
            {
                if (getText().length() <= limit)
                    inputLogic(charTyped);
                // mozliwosc usuwania znakow
                else if (getText().length() > limit && charTyped == DELETE_KEY)
                    deleteLastChar();
            }
            else
                inputLogic(charTyped);
        }
    }
}
```

5. Filesystem

Do zapisywania wyników, zapisywania i odtwarzania obwodu graficznego z plików tekstowych. Stosowane podczas oddelegowania funkcji konkretnych przycisków

```
std::string export_file = "obwod.txt";
save.setOnClick([&menu_otwarte, &obwod, &analizowany_obwod, &export_file]() {
    // Ścieżka do folderu "Wyniki"
    std::filesystem::path output_folder("../Obwody");

    // Tworzenie folderu, jeśli nie istnieje
    if (!std::filesystem::exists(output_folder))
        std::filesystem::create_directory(output_folder);
    // Ustawienie pełnej ścieżki do pliku
    std::filesystem::path output_file = output_folder / export_file;

    // Zapisanie pliku
    obwod.setFrequency(analizowany_obwod.get_frequency());
    obwod.saveElementsToFile(output_file.string());

    menu_otwarte = false;
    });
```

6. Koncepty

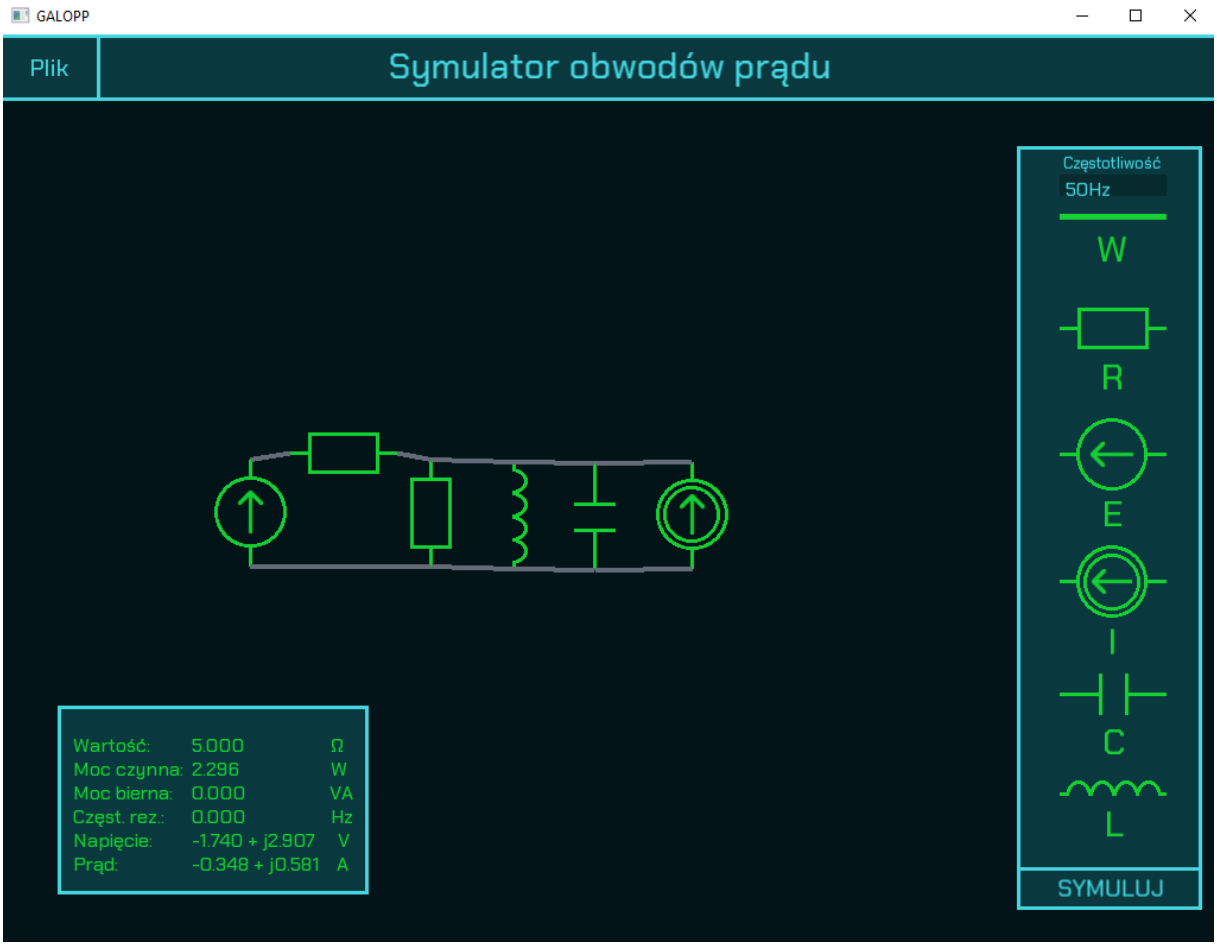
Potrzebne do haszowania klucza mapy przechowującej współrzędne węzłów na ekranie. Koncept filtruje możliwe do hashowania typy zmiennych, a struktura z odpowiednim hashem służy do zdefiniowania unikalnego klucza dla mapy węzłów (pola Obwodu analitycznego).

```
template <typename T>
concept Hashable = requires(T a) {
    { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
};

struct pair_hash {
    template <Hashable T1, Hashable T2>
    std::size_t operator()(const std::pair<T1, T2>& pair) const {
        auto hash1 = std::hash<T1>{}(pair.first);
        auto hash2 = std::hash<T2>{}(pair.second);
        return hash1 ^ (hash2 << 1);
    }
};
```

Testowanie

Program pod względem analizy został przetestowany na kilku obwodach. Znaleziony został błąd, który powodował wypisywanie wartości elementów na ekran o ‘niewiadomych’ własnościach. Poprzednio, przy wczytywaniu elementu z pliku tekstowego, dodawanie elementów do wektora odbywało się za pomocą funkcji push_back(), zamiast wywoływać uprzednio stworzoną metodę klasy ElementManager dodajElement(), która dodatkowo przed dodaniem ustawia unikalne id elementowi w obwodzie. To umożliwiło właściwy odczyt wartości po najechaniu na element.



Dodatkowo – program działa dobrze tylko dla oryginalnego okna (ewentualnie rozmiar możemy zmienić przy tworzeniu obiektu window na początku funkcji main). Jeżeli zmienimy rozmiar okna podczas działania, wygląd się „rozjeżdża” i niewłaściwie zarządza się elementami.

Obwód ze screenshota powyżej został prześledzony dla wyników tego samego obwodu (wartości zmienione!) ze strony:
[PE2024_ASOSU: Aplikacja interaktywna - Rozpływ prądów w obwodzie AC – obwód rozgałęziony | eSezam (pw.edu.pl)]

Uwagi:

Projektowanie obwodu, a później podgląd obliczonych wartości może być nieco skomplikowane ze względu na fakt, że program oblicza prądy i napięcia od lewego do prawego węzła. Stąd przydała by się wizualna informacja, który węzeł elementu jest lewy, a który prawy. Domyślnie elementy wybrane posiadają lewy węzeł na środku lewego boku, i odpowiednio prawy na środku prawego. Jednak przy projektowaniu większych obwodów, kiedy elementy mamy obrócone (nie ma różnicy np. dla rezystorów w wyglądzie obróconych o 0 lub 180 stopni), ciężko stwierdzić który węzeł jest który.

Aktualnie nie można uruchomić programu z pliku exe, jest to możliwe tylko przez Visual Studio. Rezultatem otworzenia z pliku exe będą niezaładowane czcionki i obrazy, a zapisywanie do pliku odbywa się w folderze nadrzędnym.

Wartości wyświetlane w programie zaokrąglane są do 3 miejsc po przecinku. Trzeba wziąć na to poprawkę przy wprowadzaniu wartości typu 1 nano. Zapisywanie wyników do pliku pozwala na dokładniejsze prześledzenie, a także podejrzenie bilansu mocy obwodu. Jeżeli skonstruowany właściwie, bilans powinien wyjść 0. Jeżeli element oddaje moc, wypisana jest ona z minusem. Jeżeli pobiera, odwrotnie.

Poza tym program działa dobrze, a obliczone wartości zgadzają się z oczekiwanymi. Programy można symulować zarówno dla obwodów prądu stałego, jak i przemiennego.

Wnioski

Realizowany projekt pomógł nieco lepiej zapoznać się z tematyką programowania obiektowego, a także pozwolił zapoznać się z biblioteką graficzną SFML. Dodatkowo zapoznałem się nieco głębiej z językiem C++ i jego możliwościami, wykorzystując biblioteki tj. regex, filesystem, threads (mimo braku końcowego wykorzystania w projekcie), concepts i wieloma innymi. Tworzenie zaawansowanej aplikacji i połączenie jej z uprzednio napisanym kodem w C++ było nieco wyzwaniem, jednak nauczyło mnie właściwej pracy nad ‘całkiem’ dużym projektem i odpowiednim zarządzaniem tego typu projektem.