Nazwa przedmiotu: ...
Kierunek: …
Semestr: …

System obsługi kontaktów

P**rowadzący projekt:** …

Członkowie sekcji projektowej:	
Imię i nazwisko:	*Zakres obowiązków:*
Tomasz Wojtasek	Opracowanie struktury opartej na wzorcu MVC, zastosowanie odpowiedniej hermetyzacji, wprowadzenie obsługi wyjątków oraz zabezpieczenie wprowadzanych danych, wprowadzenie algorytmów do operacji na kontaktach, wymyślenie i opracowanie sposobu na zapis i odczyt danych do pliku binarnego, wprowadzenie biblioteki “pdcurses” do stworzenia klasy AppView odpowiedzialnej za interfejs użytkownika, zastosowanie lambd do stworzenia menu.
Paweł Kurek	Opracowanie struktury opartej na wzorcu MVC, zastosowanie odpowiedniej hermetyzacji, wprowadzenie obsługi wyjątków oraz zabezpieczenie wprowadzanych danych, wprowadzenie algorytmów do operacji na kontaktach, wymyślenie i opracowanie sposobu na zapis i odczyt danych do pliku binarnego, opracowanie algorytmu sortowania dla różnych kryteriów.

Gliwice, 2023

1 Cel i temat projektu

- **Temat projektu**: Prosty i wygodny w obsłudze system obsługi kontaktów.

- **Cel projektu:** Opracowanie systemu zarządzania kontaktami z możliwością zapisania imienia m nazwiska, numeru telefonu, adresu e-mail, itp. Zapewnienie możliwości dodawania, usuwania i przeszukiwania, sortowania oraz edytowania rekordów według zadanego klucza oraz posiadania zabezpieczenia przed wprowadzeniem błędnych lub duplikujących się danych.

2 Opis projektu

2.1 Założenia projektu

Głównym założeniem aplikacji była implementacja wzorca **MVC**, na którym opiera się cała architektura aplikacji, wprowadzenie przejrzystego interfejsu użytkownika oraz wdrożenie odpowiednich algorytmów aby projekt był zgodny z wymaganiami które otrzymała grupa projektowa.

Zastosowany **wzorzec MVC** zmienił logikę aplikacji oraz ułatwił pozbierać wszystko w całość. Odpowiednie rozgraniczenie między warstwą logiczną aplikacji, interfejsem użytkownika oraz modelami typu Contact czy ContactBook ułatwiły zarządzanie kodem oraz dalsze ulepszanie go. W momencie kiedy uczestnik zespołu wpadł na pomysł na implementację dodatkowej metody, to od razu było wiadomo w którym miejscu ma się ona znaleźć. Zastosowania wspomnianego wzorca projektowego można znaleźć na stronie [MVC].

Zgodnie z założeniem została użyta biblioteka **algorithms** z której (zgodnie z zamysłem "Po co na nowo wynajdywać koło") użyliśmy gotowej metody sortującej, która jednocześnie potrafi sortować według wielu kryteriów, dzięki czemu mogliśmy użyć jednej metody w wielu przypadkach, co znacznie zoptymalizowało algorytm sortowania.

Priorytetowym założeniem było opracowanie wygodnego interfejsu użytkownika, dzięki któremu obsługa programu przez użytkownika będzie bardzo prosta. Do stworzenia interfejsu użytkownika wykorzystano bibliotekę **pdcurses**, która umożliwia zaawansowane operacje na konsoli. Ważnym priorytetem było aby zamknąć użycie tej biblioteki tylko w klasie **AppView** – zgodnie z wzorcem MVC, który mówi, że jakakolwiek interakcja z użytkownikiem powinna się zawierać w klasach typu modelu **view**. Takich klas może być wiele, jednak postanowiono zamknąć się do jednej takiej klasy z uwagi na niski stopień zaawansowania aplikacji. Dużym źródłem inspiracji jeśli chodzi o interfejs okazał się program demo z biblioteki pdcurses [PDC2], który jest przykładem programu interfejsu tekstowego.

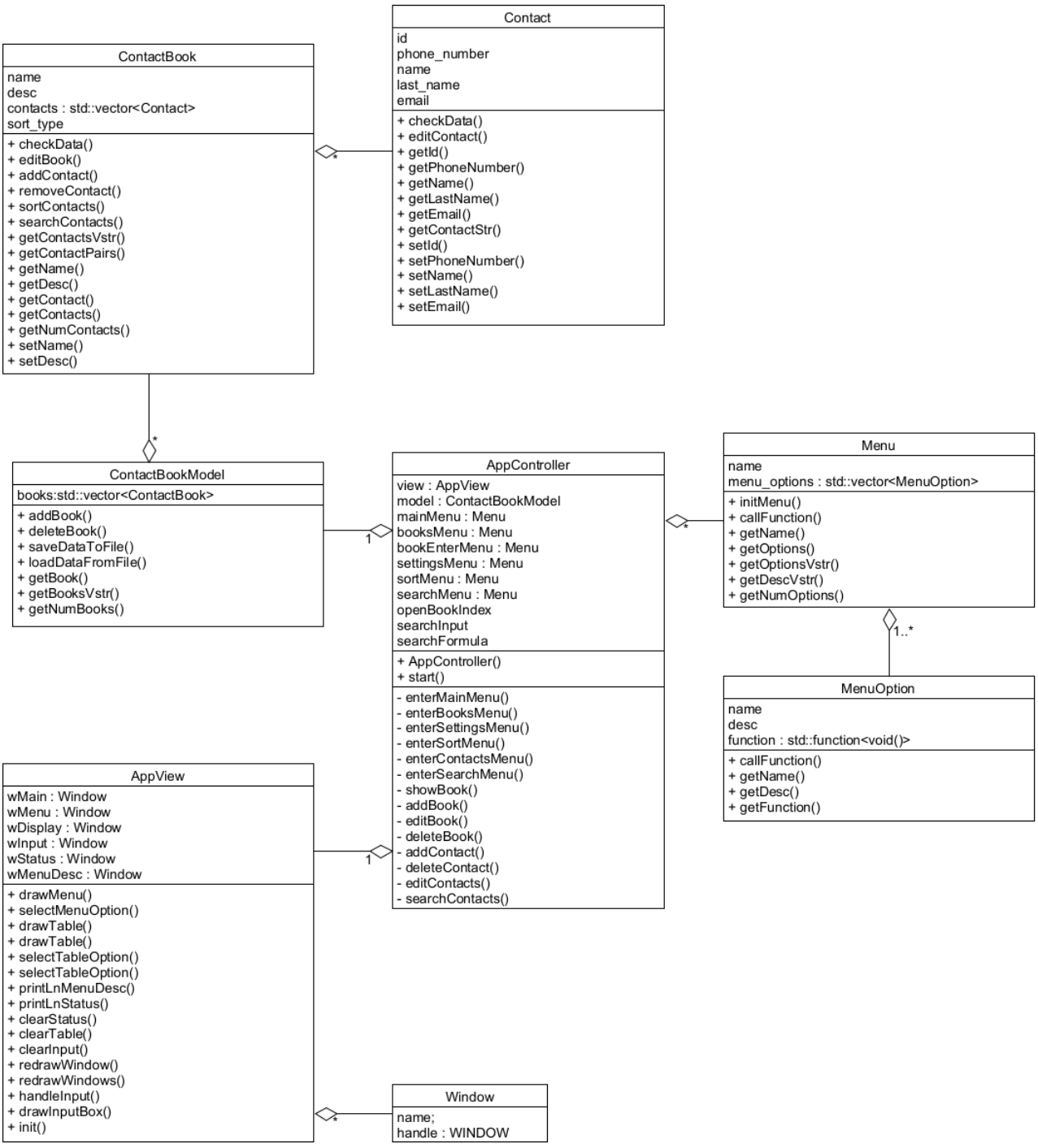
Ważnym aspektem dla zespołu było odpowiednie zorganizowanie kodu dotyczącego tworzenia menu aby maksymalnie go skrócić bez spadku na wydajności. Założeniem było pozbycie się wielu operacji **switch(case)**, co ostatecznie się udało.

Jeśli chodzi o logikę tworzenia menu – zastosowano klasy **Menu** oraz **MenuOption** przy czym klasa **AppController** jest odpowiedzialna za stworzenie obiektów tych klas oraz odpowiednie zainicjalizowanie zmiennych w owych obiektach. Następnie korzystając z własności klasy opartych o menu – można je odpowiednio wyświetlać w części **view** aplikacji.

Postanowiono że każdej opcji w menu będzie przyporządkowana jakaś funkcja, które menu będzie wykonywać, co udało się zrealizować za pomocą instrukcji **lambdy.** W ten sposób każdy obiekt menu

ma swoją funkcję z klasy **AppController** i funkcję programu wykonują się w inny sposób niż standardowo – jednak estetyczniej. W ten sposób pozbyto się instrukcji switch(case), które w nadmiarze są słabo czytelne.

2.2 Architektura programu – diagram klas



2.2.1 Główna klasa aplikacji

Główna klasa aplikacji **AppController** zawiera w sobie pojedyncze obiekty **ContactBookModel** oraz **AppView** aby mieć bezpośredni dostęp do logicznej części aplikacji oraz do wizualnej części aplikacji, zgodnie z wzorcem MVC. Składa się z wielu obiektów klasy **Menu** aby była możliwa przejrzysta łączność między poszczególnymi menu a klasą **AppView**.

2.2.2 Logika tworzenia menu oraz nawigacji po menu

W każdym obiekcie klasy **Menu** jest kontener obiektów **MenuOption**, aby każdemu menu dopasować odpowiednio jego opcję. Każdy obiekt MenuOption musi posiadać **funkcję**, którą będzie wykonywał. W tym celu w konstruktorze AppController inicjalizujemy obiekty Menu kontenerami obiektów MenuOption, które przyjmują w parametrze funkcję stricte z klasy AppController. To znaczy, że w klasie AppController tworzymy funkcję, które będą działać po przejściu w dane Menu. AppController stanowi podstawę nawigacji – poprzez wykonywanie następnych funkcji z przypisanych menu, które są częścią AppController.

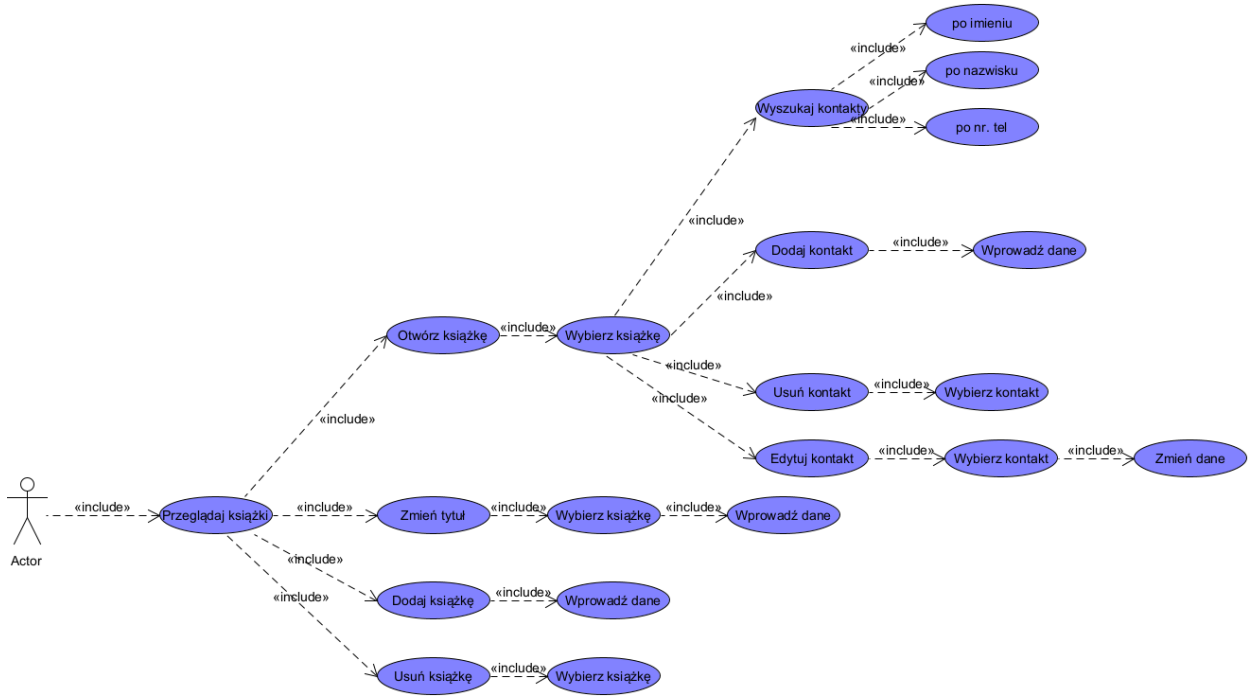
2.2.3 Wizualna część aplikacji – klasa AppView

Na klase **AppView** reprezentującą operację na interfejsie użytkownika składa się struktura **Window**, która reprezentuje okno w aplikacji na bazie okna z biblioteki pdcurses (ma jego właściwości). Klasa AppView jest klasą służącą do tworzenia wizualnego interfejsu użytkownika. Tzn. mamy w niej możliwość rysowania Menu, rysowania tabel z danymi, przechwytywania klawiszy z klawiatury, itp. jednak co ważne - cała logika nawigacji po menu dalej odbywa się w klasie AppController. Klasa AppView jak większość klas w aplikacji jest niezależna od innych klas – przykładem są tutaj metody rysujące np. tabelę – wykorzystują one jedynie dane tekstowe, tzn. trzeba przekazać do nich w parametrze dane o np. kontaktach w postaci tekstowej (czyli w funkcji przypisanemu do konkretnego menu w klasie AppController, trzeba najpierw wywołać metodę, która zwróci dane o kontaktach z klasy ContactBook, a następnie wywołać metodę rysującą tabelę na podstawie danych tekstowych).

2.2.4 Modele aplikacji

Klasa **ContactBookModel** jest głównym modelem aplikacji, zgodnie z wzorcem MVC. Służy do zarządzania obiektami **ContactBook** mając w sobie jednocześnie kontener takich obiektów. Każdy obiekt **ContactBook** ma z kolei swój kontener obiektów klasy **Contact.** Klasa ContactBookModel została wydzielona z klasy AppController w celu poprawy czytelności. W klasach tych występuje odpowiednia hermetyzacja. Dzięki takiemu rozwiązaniu w momencie stworzenia jednego obiektu modelu w AppController, jesteśmy w stanie mieć kontrole nad wszystkimi modelami danych w aplikacji.

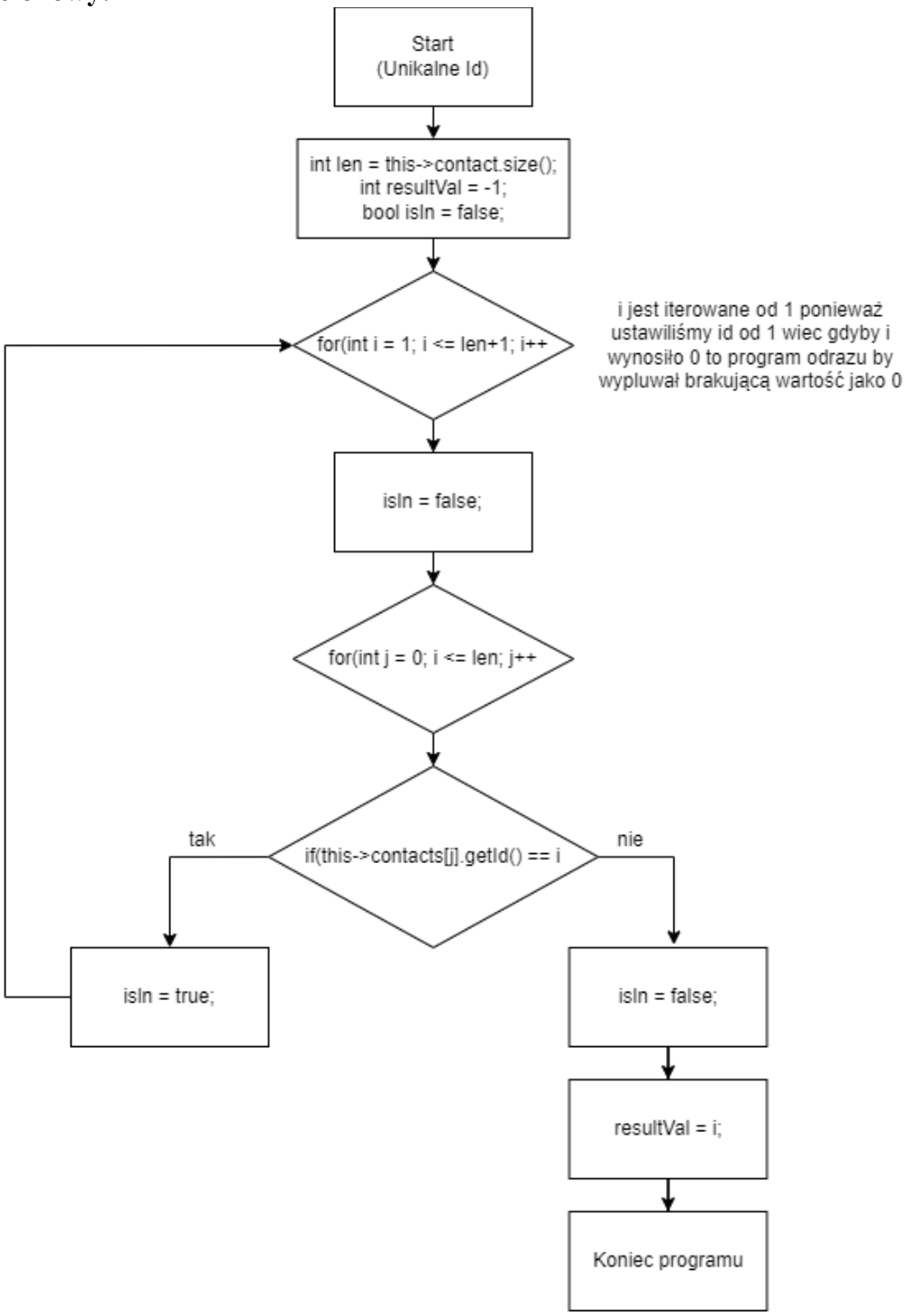
2.3 *Diagram use-case*



2.4 Przykład algorytmu – algorytm znajdowania unikatowego id.

W projekcie został stworzony algorytm "unikatowego id" dzięki któremu podczas tworzenia kontaktu program będzie wiedział jakie id mu przypisać (kiedy użytkownik np. usunął wcześniejszy kontakt). Algorytm jest szczególnie przydatny w momencie, gdy stare kontakty są usuwane, a nowe są dodawane. Jest to jeden z ważniejszych algorytmów – bez niego nie byłaby możliwa poprawna komunikacja między klasą AppView a klasą AppController i modelami.

Schemat blokowy:



Zasada działania algorytmu:

1. Inicjowane są zmienne: **int len** – liczba elementów w contacts **int resultVal** – zmienna przechowująca wynik unikalnej zmiennej, **Bool isln** - wskazujący czy dany indeks został znaleziony, początkowo ustawiony na **false**.

2. Pętla for która iteruje od **i = 1** do **len+1**, i reprezentuje potencjalne wartości id, które będą sprawdzane. Przed każdą iteracją pętli zewnętrznej, zmienna **isIn** jest resetowana i ustawiana na **false**, co oznacza, że na początku każdej iteracji nie znaleziono jeszcze danego indeksu **i** w kolekcji.

3. Wewnętrzna pętla for iteruje od **j = 0** do **len**, przechodząc przez każdy element kolekcji contacts i sprawdza czy **getId()** elementu jest równy aktualnej wartości **i**.

4. Jeśli wewnętrzna pętla znajdzie element o id równym **i**, ustawia zmienna **isln** na **true** i przerywa pętlę, ponieważ **i** nie jest wtedy unikalne w kolekcji.

5. Jeśli pętla wewnętrzna nie znajdzie elementu o id = **i**, ustawia zmienna **isln** na **false**, a **i** zostaje unikalnym elementem kolekcji.

6. W przypadku znalezienia unikalnej wartości zmienna **resultVal** zostaje ustawiona na wartość **i** a pętla jest przerywana.

7. Dzięki temu zmienna **resultVal** posiada unikalny identyfikator, który może zostać przypisany do nowego elementu kolekcji contacts.

2.5 Przykład algorytmu – algorytm wyszukiwania kontaktów

Algorytm wyszukiwania konkretnych kontaktów został opracowany z wykorzystaniem metod obiektu **std::string.** W metodzie wyszukiwania kontaktów kluczowe jest to, aby zwrócić dane o kontaktach jako obiekt :

> **std::vector<std::pair<int, std::string>>** , gdzie zmienną **int** jest id kontaktu, a **std::string** jest ciągiem znaków składających się z wszystkich danych osobowych wyszukanego kontaktu (korzystając z metody **Contact::getContactStr**()).

Tak zwrócona wartość może zostać przekazana w parametrze do metod z klasy AppView. Dzięki temu gdy użytkownik w metodzie klasy AppView nawiguje po tabeli z danymi, to metoda ta zwróci odpowiedni index – będący analizowany po logicznej stronie aplikacji w AppController.

```
std::vector<std::pair<int, std::string>> ContactBook::searchContacts(std::string input, std::string formula)
{
    bool exactMatch = false;
    if (formula != "byName" && formula != "byLastName" && formula != "byPhoneNumber") {
        throw std::invalid_argument("Podano zla formule wyszukania!");
    }

    std::vector <std::pair<int, std::string>> contactsFound;
    std::string to_find;

    for (auto& contact : this->contacts)
    {
        if (formula == "byName")
            to_find = contact.getName();

        else if (formula == "byLastName")
            to_find = contact.getLastName();

        else if (formula == "byPhoneNumber")
            to_find = contact.getPhoneNumber();

        if (to_find.find(input) != std::string::npos) {
            contactsFound.push_back(std::make_pair(contact.getId(), contact.getContactStr()));
        }
    }

    return contactsFound;
```

Metoda **searchContacts** przyjmuje w parametrze zmienną **std::string input** – czyli fraza, którą wpisują użytkownik i po której algorytm ma wyszukiwać oraz zmienną **std::string formula** – która określa formułę, po której algorytm ma szukać danych.

Iterujemy po wszystkich kontaktach. Przed operacją wyszukiwania sprawdzana jest zmienna z formułą, w zależności od niej zmienna **to_find** jest ustawiana na odpowiednią zmienną kontaktu.

Następnie odbywa się wyszukiwanie przy użyciu metody klasy std::string – **find**, która sprawdza czy w frazie, która wpisał użytkownik występuje określona wcześniej zmienna kontaktu. Ostatecznie w przypadku odnalezienia, wynik zostaje dodany do kontenera **contactsFound**, który po zakończonym procesie zostaje zwrócony.

3 Obsługa błędów i testowanie programu

W programie zorganizowano obsługę błędów za pomocą mechanizmu zgłaszania i obsługi wyjątków. W klasach **ContactBook**, **ContactBookModel** oraz **Contact** zastosowano instrukcje **throw** do zgłaszania wyjątków w przypadku napotkania nieprawidłowych danych lub błędów. W następnych nagłówkach zostaną przedstawione przykładowe metody korzystające z obsługi błędów.

3.1 Metoda checkData

W klasie ContactBook znajduje się metoda **checkData**, która jest wykorzystywana w metodach tworzących lub edytujących pojedynczy kontakt oraz w konstruktorze parametrycznym. Metoda ta sprawdza poprawność danych podczas tworzenia lub edycji kontaktu. Jeśli któreś z pól nie spełnia określonych warunków (np. nieprawidłowa długość imienia), metoda zgłasza wyjątek typu **std::length_error** z odpowiednim komunikatem.

```
void ContactBook::checkData(std::string name, std::string desc)
{
    if (name.length() > 12 || name.length() == 0)
        throw std::length_error("Nieprawidlowa dlugosc imienia");

    if (desc.length() > 50 || desc.length() == 0)
        throw std::length_error("Nieprawidlowa dlugosc nazwiska");
}
```

3.2 Metoda deleteBook

Analogicznie, w klasie **ContactBookModel** metoda **deleteBook** sprawdza, czy podany indeks książki jest prawidłowy. Jeśli indeks jest nieprawidłowy, metoda zwraca wartość logiczną **false**, co może służyć jako sygnał o błędzie.

```
bool ContactBookModel::deleteBook(int index)
{
    if (index >= 0 && index < books.size()) {
        this->books.erase(this->books.begin() + index);
        return true;
    }
    else {
        return false;
    }
}
```

W klasie **Contact** metoda **checkData** sprawdza poprawność danych podczas tworzenia lub edycji kontaktu. Jeśli którekolwiek z pól nie spełnia warunków (np. nieprawidłowa długość numeru telefonu), metoda zgłasza wyjątek **std::length_error** z odpowiednim komunikatem.

```
void Contact::checkData(int phone_number, std::string name, std::string last_name, std::string email)
{
    if(std::to_string(phone_number).length() != 9)
       throw std::length_error("Nieprawidlowa ilosc znakow w numerze telefonu");

    if(name.length()>12 || name.length() == 0)
        throw std::length_error("Nieprawidlowa dlugosc imienia");

    if(last_name.length()>16 || last_name.length() == 0)
        throw std::length_error("Nieprawidlowa dlugosc nazwiska");

    if(email.length()>30 || email.length() == 0)
        throw std::length_error("Nieprawidlowa dlugosc adresu email");
}
```

3.3 Przykład przechwytywania błędów

Poniżej znajduję się przykład przechwytywania błędów. W bloku **try** staramy się wykonać instrukcję, która może wygenerować wyjątek, w tym wypadku wywołujemy metodę **addContact**. Do wyświetlania informacji o błędzie, bądź sukcesie wykonania operacji – służy metoda z klasy AppView – **printLnStatus**. Metoda ta wyświetla daną informację - tekst w odpowiednim miejscu w konsoli.

```
try {
    book->addContact(std::stoi(results[2]), results[0], results[1], results[3]); // sprobuj dodac kontakt
    this->view.printLnStatus("Pomyslnie dodano nowy kontakt!");
    close = true; // jesli sie uda to wyjdz z petli
}
catch (const std::invalid_argument& e) { // wylapuj ewentualne bledy
    this->view.printLnStatus("Podano zle dane! Sprawdz poprawnosc danych.");
}
catch (const std::length_error& e) {
    this->view.printLnStatus(e.what());
}
catch (const std::exception& e) {
    this->view.printLnStatus("Wystapil nieoczekiwany blad! Prosimy o zgloszenie bledu w naszym formularzu kontaktowym.");
}
```

Podsumowując w przypadku wystąpienia wyjątku, kod obsługujący te klasy przechwytuje wyjątek i podejmuje odpowiednie działania, takie jak wyświetlenie komunikatu o występującym błędzie lub podjęcie innych kroków naprawczych. W przypadku pomyślnego przechwycenia wyjątku, informacje są wyświetlane, aby użytkownik mógł je zobaczyć.

3.4 Testowanie programu

Testowanie aplikacji odbywało się niekiedy poprzez badanie zmiennych i wywołań funkcji wbudowanym w środowisko visual studio narzędziem - debuggerem. Ta metoda szczególnie była przydatna podczas sprawdzania działania nawigacji po menu. Nawigację po menu w aplikacji trzeba było zaimplementować w sposób, aby wywołania konkretnych funkcji przypisanych do następnych menów nie zagnieżdżały się. W pewien sposób zamierzony cel osiągnięto, jednak w niektórych przypadkach dalej występują problemy.

4 Podsumowanie i wnioski

Projekt spełniał swoje cele lecz nie został zrealizowany w 100% zgodnie z wcześniejszymi założeniami grupy projektowej.

Program posiadał książki kontaktów tak aby można było podzielić kontakty według uznania oraz zgodnie z założeniami umożliwiał zarządzaniem kontaktów jak i każdym jego rekordem (tj. umożliwiał dodawanie, usuwanie ale także wyszukiwanie czy edytowanie kontaktów oraz książek kontaktów), który umożliwiał przejrzysty i wygodny w użyciu interfejs, który został stworzony zgodnie ze wzorcem **MVC** w celu bardziej przejrzystego kodu.

W projekcie również zgodnie z planem zostały zaimplementowane zabezpieczenia przed wprowadzeniem błędnych lub duplikujących danych oraz obsługa wyjątków, zastosowano również odpowiednią hermetyzację.

Problem okazał się jednak w poszczególnych algorytmach takich jak **algorytm sortowania** dla różnych kryteriów czy **zapis i odczyt do pliku binarnego**.
Sporą trudnością było zidentyfikowanie problemu podczas wdrażania **algorytmu sortowania** do projektu (wersja wcześniejsza działała bez zarzutów), zaś jeśli chodzi o algorytm zapisu i odczytu z **pliku binarnego** problemem był fakt iż używaliśmy kontener **std::vector** obiektów który każdy posiadał inny kontener obiektów, co powodowało duże zagnieżdżenie elementów, przez co program który stworzyliśmy nie działał w 100% poprawnie.

U członków zespołu pojawił się dylemat, jakiej technologii powinno się użyć zamiast plików binarnych do zapisu/odczytu danych. Jedną z opcji okazał się zapis do plików tekstowych. Jednak taki sposób wydaje się zbyt mało wydajny, dlatego następnym celem podczas kontynuacji tego projektu byłaby implementacja **bazy danych**, a co za tym idzie zmiana w znacznej części struktury aplikacji.

Oba te algorytmy zostały stworzone oraz przetestowane w celu wykluczenia różnych błędów aczkolwiek dużą część pracy nad aplikacją zajęło opanowanie biblioteki **pdcurses** oraz jej implementacje, czy też zastosowanie odpowiedniego wzorca MVC, które pomimo tego, że ich implementacja ułatwiła pracę nad aplikacją, potrzeba było dużego nakładu pracy aby je zrozumieć oraz przez lekkie problemy z wprowadzaniem ich do kodu, nie udało się ich przyłączyć do projektu.

Jeżeli nasza grupa projektowa miałaby jeszcze raz napisać ten sam projekt to na pewno przyłączylibyśmy algorytmy z którymi wyszły małe komplikacje, skupilibyśmy na dodaniu klasy użytkownika oraz metodami rejestracji i logowania ale również chcielibyśmy się skupić nad małymi dodatkami wizualnymi takimi jak np. dołożenie trybu nocnego oraz dziennego.
Poświęcilibyśmy też więcej czasu na lepszą optymalizacje całego kodu (w celu lepszej czytelności). Być może lepszym rozwiązaniem byłoby skupić się na prostszych rzeczach, które będą w 100% poprawnie zaimplementowane niż zabieranie się za aspekty typowo wizualne.

Dużą część pracy nad aplikacją zajęło opanowanie biblioteki pdcurses oraz jej implementacje, czy też zastosowanie odpowiedniego wzorca, które pomimo tego, że ich implementacja ułatwiła pracę nad aplikacją, potrzeba było dużego nakładu pracy aby je zrozumieć. Zamiast skupiania się nad tymi zagadnieniami, można było dokończyć implementację zapisu/odczytu do pliku, która ma bardzo duże znaczenie dla użytkownika oraz stanowi dla niego duże ułatwienie oraz poprawnie zaimplementować inne algorytmy.

5 Spis źródeł

[MVC] Opis wzorca MVC https://www.educba.com/what-is-mvc-design-pattern/

[BA1] Biblioteka Algorithms: https://en.cppreference.com/w/cpp/algorithm

[PDC1] Dokumentacja biblioteki pdcurses: https://pdcurses.org/docs/MANUAL.html

[PDC2] Program demo biblioteki pdcurses (tui): https://github.com/wmcbrine/PDCurses/blob/master/demos/tuidemo.c