



**WYŻSZA SZKOŁA
INFORMATYKI I ZARZĄDZANIA**
z siedzibą w Rzeszowie

KOLEGIUM INFORMATYKI STOSOWANEJ

Kierunek: Informatyka

Specjalność: Programowanie

Karolina Rydzik

Numer albumu: w67264

System Zarządzania Informacjami Lotniczymi

Prowadzący: mgr inż. Ewa Żesławska

Projekt z Programowania Obiektowego w języku C#

Rzeszów 2024

Spis treści

Wstęp	4
1 Wymagania Funkcjonalne i Niefunkcjonalne	5
1.1 Funkcjonalne:	5
1.2 Niefunkcjonalne:	5
2 Struktura Projektu	6
2.1 Pliki projektu.	6
2.2 Główne koncepcje	6
2.3 Przedstawienie i opis klas.	7
2.4 Diagram klas	10
3 Opis Techniczny Projektu	11
3.1 Język programowania	11
3.2 Wykorzystane narzędzia.	11
3.3 Rozbudowane Funkcje Visual Studio	12
3.4 Wykorzystane technologie	13
3.5 Minimalne wymagania sprzętowe	13
3.6 Omówienie zarządzania danymi	13
3.7 Zapis i odczyt danych	14
3.8 Eliminacja cyklicznych referencji	14
3.9 Baza Danych - JSON.	15
4 Harmonogram Realizacji Projektu	16
4.1 Diagram Gantta	16
5 Repozytorium i System Kontroli Wersji	17
5.1 Link do repozytorium	17
5.2 System kontroli wersji.	17
6 Warstwa Użytkowa Projektu.	19
6.1 Menu i obsługa wejścia	19
6.2 Funkcje operacyjne.	20
6.3 Obsługa Bazy Danych	21
7 Podsumowanie Projektu i Dalsze Realizacje	22
7.1 Podsumowanie projektu	22
7.2 Dalsze plany rozwojowe projektu	22
Bibliografia.	23
Spis rysunków.	24

Wstęp

Projekt ma na celu stworzenie systemu rezerwacji miejsc w samolotach, umożliwiającego skuteczną organizację lotów i zarządzanie danymi pasażerów. Główne założenia i cele projektu obejmują:

- **Wyszukiwanie Dostępnych Lotów**

System umożliwia wyszukiwanie lotów na podstawie miasta startu, miasta lądowania oraz daty. Wyniki wyszukiwania powinny zawierać informacje o dostępnych miejscach, godzinie, cenie.

- **Rezerwacja Miejsc**

Pasażerowie mogą dokonywać rezerwacji miejsc na wybranych lotach. Rezerwacje powinny być powiązane z konkretnymi pasażerami, system powinien wprowadzać do systemu rezerwację wraz z danymi podróżującego.

- **Odwoływanie Rezerwacji**

System umożliwia pasażerom odwoływanie swoich rezerwacji. Po odwołaniu rezerwacji, miejsce zostaje zwolnione i staje się dostępne dla innych pasażerów.

- **Lista Pasażerów**

Możliwość wygenerowania listy pasażerów dla konkretnego lotu. Lista powinna zawierać imiona, nazwiska oraz inne istotne dane pasażerów.

Rozdział 1

Wymagania Funkcjonalne i Niefunkcjonalne

1.1 Funkcjonalne:

- Wyszukiwanie lotów na podstawie miasta startu, miasta lądowania i daty.
- Rezerwacja miejsc na lotach, z przypisaniem danych pasażera.
- Odwoływanie rezerwacji, zwalnianie miejsc.
- Generowanie listy pasażerów dla konkretnego lotu.
- Operacje CRUD dla danych o lotach, miejscach, rezerwacjach i pasażerach.
- Intuicyjny interfejs użytkownika umożliwiający łatwą obsługę systemu.

1.2 Niefunkcjonalne:

- Wydajność: System powinien działać sprawnie nawet przy dużej ilości danych.
- Bezpieczeństwo: Zapewnienie bezpieczeństwa danych pasażerów i informacji o lotach.
- Przenośność: System powinien być łatwo przenośny między różnymi środowiskami.
- Dokumentacja: Dokumentacja projektowa, umożliwiająca zrozumienie i rozwijanie systemu.
- Niezawodność: Minimalizacja błędów i awarii systemu poprzez solidne testowanie.

Rozdział 2

Struktura Projektu

Projekt "Flight System" został zorganizowany w formie konsolowej aplikacji w języku C#. Poniżej zaprezentuję opis struktury projektu, obejmujący kluczowe elementy, ich funkcje i relacje.

2.1 Pliki projektu

- **FlightSystem.csproj** - Plik projektu zawiera konfigurację i ustawienia kompilacji.
- **Program.cs** - Główny plik programu zawiera funkcję 'Main' i służy jako punkt wejścia do aplikacji.
- **FileDatabase.cs** - Klasa statyczna FileDatabase obsługuje operacje zapisu i odczytu danych do/z plików JSON.

2.2 Główne koncepcje

- **Klasa BaseEntity** - Klasa bazowa zawierająca pole Id, służące jako identyfikator dla wszystkich klas.
Klasa BaseEntity stanowi fundament dla hierarchii klas w systemie. Jej główną cechą jest posiadanie pola Id, które służy jako unikalny identyfikator każdego obiektu stworzonego na bazie tej klasy. Głównym celem jest dostarczenie wspólnego mechanizmu identyfikacji dla obiektów.
- **Klasa FileDatabase** - pełni kluczową rolę w obszarze trwałości danych, umożliwiając zapis i odczyt informacji z plików JSON.
Klasa FileDatabase to kluczowy komponent aplikacji, który umożliwia przechowywanie danych między różnymi sesjami programu. Dane są zapisywane do plików JSON, co umożliwia trwałe przechowywanie informacji między kolejnymi uruchomieniami programu.

2.3 Przedstawienie i opis klas

- **BaseEntity** - Klasa bazowa BaseEntity zawiera jedno pole Id typu int, które służy jako identyfikator dla wszystkich klas w systemie.

```
1 namespace FlightSystem;
2
3 public class BaseEntity
4 {
5     public int Id { get; set; }
6 }
```

Rysunek 2.1: Klasa BaseEntity

- **Airport** - Klasa Airport reprezentuje lotnisko. Posiada pola takie jak Name (nazwa lotniska), City (miasto, w którym się znajduje) oraz Country (kraj, w którym się znajduje).

```
1 namespace FlightSystem;
2
3 public class Airport: BaseEntity
4 {
5     public string Name { get; set; }
6     public string City { get; set; }
7     public string Country { get; set; }
8 }
```

Rysunek 2.2: Klasa Airport

- **Passenger** - Klasa Passenger reprezentuje pasażera. Zawiera informacje o imieniu (FirstName), nazwisku (LastName) oraz listę biletów (Tickets), które dany pasażer posiada.

```
1 namespace FlightSystem;
2
3 public class Passenger: BaseEntity
4 {
5     public string FirstName { get; set; }
6     public string LastName { get; set; }
7     public List<Ticket> Tickets { get; set; }
8 }
```

Rysunek 2.3: Klasa Passenger

- **PassengerAircraft** - Klasa PassengerAircraft opisuje samolot pasażerski. Zawiera informacje o liczbie miejsc w klasach: ekonomicznej (EconomySeats), biznesowej (BusinessSeats) i pierwszej klasie (FirstClassSeats).

```

1 namespace FlightSystem;
2
3 public class PassengerAircraft : Aircraft
4 {
5     public int EconomySeats { get; set; }
6     public int BusinessSeats { get; set; }
7     public int FirstClassSeats { get; set; }
8 }

```

Rysunek 2.4: Klasa PassengerAircraft

- **Flight** - Klasa Flight reprezentuje lot. Posiada informacje o lotnisku odlotu i przylotu, czasie odlotu i przylotu, samolocie, cenie bazowej oraz liście rezerwacji. Metody isBusinessSeat(), isEconomySeat(), isFirstClassSeat() sprawdzają dostępność miejsc w poszczególnych klasach.

```

1 namespace FlightSystem;
2
3 public class Flight : BaseEntity
4 {
5     public Airport DepartureAirport { get; set; }
6     public Airport ArrivalAirport { get; set; }
7     public DateTime DepartureTime { get; set; }
8     public DateTime ArrivalTime { get; set; }
9     public PassengerAircraft Aircraft { get; set; }
10    public List<Reservation> Reservations { get; set; }
11    public int BasePrice { get; set; }
12
13    public bool isBusinessSeat()
14    {
15        int businessReservationsCount = Reservations.Count(r => r.Ticket.Class == TicketClass.Business);
16        return businessReservationsCount < Aircraft.BusinessSeats;
17    }
18
19    public bool isEconomySeat()
20    {
21        int economyReservationsCount = Reservations.Count(r => r.Ticket.Class == TicketClass.Economy);
22        return economyReservationsCount < Aircraft.EconomySeats;
23    }
24
25    public bool isFirstClassSeat()
26    {
27        int firstClassReservationsCount = Reservations.Count(r => r.Ticket.Class == TicketClass.FirstClass);
28        return firstClassReservationsCount < Aircraft.FirstClassSeats;
29    }
30 }

```

Rysunek 2.5: Klasa Flight

- **Ticket** - Klasa Ticket reprezentuje bilet. Zawiera informacje o locie, pasażerze, numerze miejsca, cenie i klasie. Posiada metodę adjustPrice(), która dostosowuje cenę biletu w zależności od klasy. Enum TicketClass reprezentuje różne klasy biletów: Economy, Business, FirstClass.

```

1 namespace FlightSystem;
2
3 public class Ticket : BaseEntity
4 {
5     public Flight Flight { get; set; }
6     public Passenger Passenger { get; set; }
7     public string SeatNumber { get; set; }
8     public decimal Price { get; set; }
9     public TicketClass Class { get; set; }
10
11    public void adjustPrice()
12    {
13        if (Class == TicketClass.Business)
14        {
15            Price *= (decimal)1.4;
16        }
17        else if (Class == TicketClass.FirstClass)
18        {
19            Price *= 2;
20        }
21    }
22
23    public enum TicketClass
24    {
25        Economy,
26        Business,
27        FirstClass
28    }
29 }

```

Rysunek 2.6: Klasa Ticket

- **Reservation** - Klasa Reservation opisuje rezerwację. Zawiera informacje o bilecie (Ticket), kodzie rezerwacji (ReservationCode) oraz statusie potwierdzenia (IsConfirmed).

```

1 namespace FlightSystem;
2
3 public class Reservation: BaseEntity
4 {
5     public Ticket Ticket { get; set; }
6     public string ReservationCode { get; set; }
7     public bool IsConfirmed { get; set; }
8 }

```

Rysunek 2.7: Klasa Reservation

- **FileDatabase** - Klasa statyczna FileDatabase zawiera metody do operacji zapisu i odczytu danych z plików JSON, przechowujących listy lotów, rezerwacji i pasażerów.

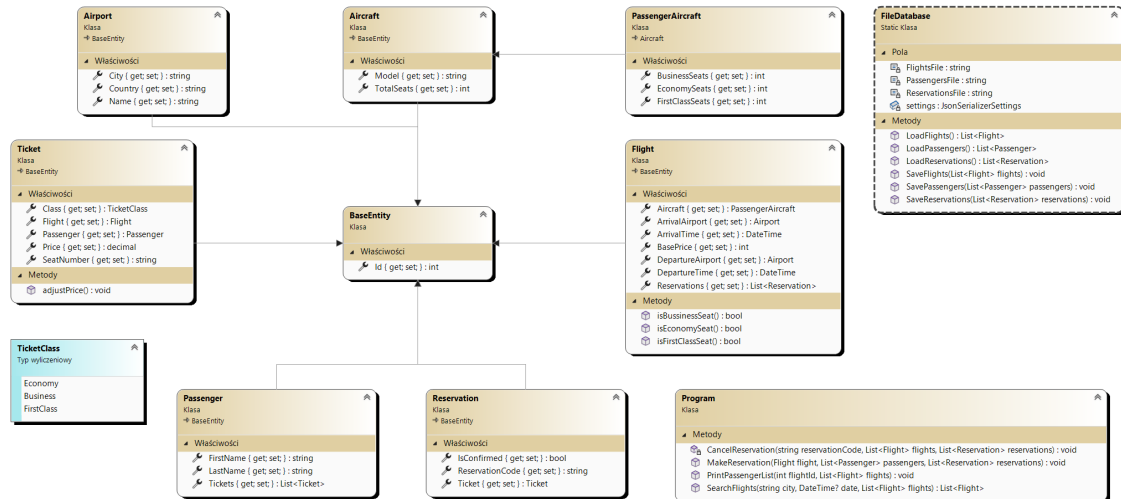
```

4
5 public static class FileDatabase
6 {
7     private const string FlightsFile = "flights.json";
8     private const string ReservationsFile = "reservations.json";
9     private const string PassengersFile = "passengers.json";
10    private static JsonSerializerSettings settings = new JsonSerializerSettings
11    {
12        ReferenceLoopHandling = ReferenceLoopHandling.Ignore
13    };
14    // Metoda do zapisu listy lotów do pliku
15    public static void SaveFlights(List<Flight> flights)
16    {
17        string json = JsonConvert.SerializeObject(flights, Formatting.Indented, settings);
18        File.WriteAllText(FlightsFile, json);
19    }
20
21    // Metoda do odczytu listy lotów z pliku
22    public static List<Flight> LoadFlights()
23    {
24        if (!File.Exists(FlightsFile))
25            return new List<Flight>();
26
27        string json = File.ReadAllText(FlightsFile);
28        return JsonConvert.DeserializeObject<List<Flight>>(json);
29    }
30
31    // Metoda do zapisu listy rezerwacji do pliku
32    public static void SaveReservations(List<Reservation> reservations)
33    {
34        string json = JsonConvert.SerializeObject(reservations, Formatting.Indented, settings);
35        File.WriteAllText(ReservationsFile, json);
36    }
37
38    // Metoda do odczytu listy rezerwacji z pliku
39    public static List<Reservation> LoadReservations()
40    {
41        if (!File.Exists(ReservationsFile))
42            return new List<Reservation>();
43
44        string json = File.ReadAllText(ReservationsFile);
45        return JsonConvert.DeserializeObject<List<Reservation>>(json);
46    }
47
48    // Metoda do zapisu listy pasażerów do pliku
49    public static void SavePassengers(List<Passenger> passengers)
50    {
51        string json = JsonConvert.SerializeObject(passengers, Formatting.Indented, settings);
52        File.WriteAllText(PassengersFile, json);
53    }
54
55    // Metoda do odczytu listy pasażerów z pliku
56    public static List<Passenger> LoadPassengers()
57    {
58        if (!File.Exists(PassengersFile))
59        {
60            return new List<Passenger>();
61        }
62
63        string json = File.ReadAllText(PassengersFile);
64        return JsonConvert.DeserializeObject<List<Passenger>>(json);
65    }
66 }

```

Rysunek 2.8: Klasa Statyczna FileDatabase

2.4 Diagram klas



Rysunek 2.9: Przedstawienie diagramu klas programu 'Flight System'

Rozdział 3

Opis Techniczny Projektu

3.1 Język programowania

Projekt został stworzony przy użyciu języka C#, który jest szczególnie docenianym narzędziem w świecie programistycznym, dzięki swoim możliwościom programowania obiektowego. C# jest rozwijany przez firmę Microsoft i stanowi solidny fundament dla aplikacji, zwłaszcza gdy korzystamy z frameworku .NET. Przedstawiany przeze mnie projekt wykorzystuje potencjał C# i frameworka .NET.

3.2 Wykorzystane narzędzia

Visual Studio 2022

Środowisko programistyczne Visual Studio, zastosowane w procesie tworzenia i edycji przedstawianego przeze mnie projektu reprezentuje jedno z najbardziej zaawansowanych narzędzi dostępnych dla programistów.



3.3 Rozbudowane Funkcje Visual Studio

- **Edycja Kodu**

Bogate funkcje edytora umożliwiają szybkie i intuicyjne pisanie, modyfikowanie oraz analizowanie kodu w języku C#. IntelliSense wspomaga programistę, oferując sugestie składni, automatyczne uzupełnianie i podpowiedzi, co przyspiesza proces programowania.

- **Debugowanie**

Zaawansowany debugger umożliwia precyzyjne śledzenie kodu, identyfikowanie błędów i optymalizację działania programu. Profilowanie kodu wspomaga w identyfikacji potencjalnych obszarów optymalizacji.

- **Integracja z .NET**

Visual Studio jest silnie zintegrowane z platformą .NET, co ułatwia tworzenie aplikacji na tym frameworku. Obsługuje najnowsze wersje języka C# oraz frameworku .NET, umożliwiając korzystanie z nowych funkcji i możliwości.

- **Zarządzanie Projektem**

System zarządzania projektem w Visual Studio ułatwia organizację plików, bibliotek oraz zasobów projektu. Intuicyjne narzędzia do kontroli wersji wspomagają efektywne zarządzanie kodem źródłowym.

- **Narzędzia Wspomagające Testowanie**

Wbudowane narzędzia do testowania jednostkowego pozwalają na tworzenie i wykonywanie testów, co wpływa na jakość i niezawodność oprogramowania.

Podsumowanie - Visual Studio

Do utworzonego przeze mnie Projektu “Flight System”, środowisko Visual Studio służyło jako centralne narzędzie do rozwijania aplikacji lotniczej. Dzięki swoim rozbudowanym funkcjom programistycznym, debugowaniu oraz integracji z frameworkiem .NET, wspomagało efektywny proces tworzenia projektu.

3.4 Wykorzystane technologie

Projekt wykorzystuje framework .NET, który dostarcza środowisko wykonawcze, biblioteki klas, a także narzędzia do tworzenia aplikacji. W szczególności, kod korzysta z funkcji związanych z C# oraz komponentów .NET.

3.5 Minimalne wymagania sprzętowe

- Komputer z zainstalowanym systemem operacyjnym wspierającym .NET (np. Windows)
- Procesor zgodny z architekturą x86 lub x64 - Projekt oparty jest na platformie .NET w języku C#. Architektura x86 i x64 odnosi się do rodzaju instrukcji procesora obsługiwane przez kompilator .NET. Oznacza to, że aplikacja może być uruchamiana zarówno na procesorach 32-bitowych (x86), jak i 64-bitowych (x64).
- Pamięć RAM zgodna z minimalnymi wymaganiami systemu operacyjnego.
- Wolne miejsce na dysku twardym wystarczające do instalacji Visual Studio.

3.6 Omówienie zarządzania danymi

Dane w projekcie są zapisywane i odczytywane z plików JSON przy użyciu klasy FileDatabase. Ta klasa pełni rolę warstwy trwałości danych, umożliwiając trwałe przechowywanie informacji między kolejnymi uruchomieniami programu. Poniżej przedstawię kluczowe klasy mojego programu, które zostały zapisane w plikach JSON:

- **Flight** - Reprezentuje informacje o locie, w tym lotniska, czasy odlotu i przylotu, używany samolot, cenę bazową biletu oraz listę rezerwacji.
- **Passenger** - Opisuje pasażera, zawierając informacje o imieniu, nazwisku i listę biletów pasażera.
- **Reservation** - Przedstawia rezerwację danego biletu, w tym informacje o kodzie rezerwacji, potwierdzeniu oraz powiązanym bilecie.

3.7 Zapis i odczyt danych

Zapis danych SaveFlights, SaveReservations, SavePassengers

Klasa udostępnia metody pozwalające na zapisywanie danych różnych typów obiektów (np. lotów, rezerwacji, pasażerów) do plików w formacie JSON.

Odczyt danych LoadFlights, LoadReservations, LoadPassengers

Klasa dostarcza metody do wczytywania danych z plików JSON z powrotem do list obiektów.

```
48 // Metoda do zapisu listy pasażerów do pliku
49 public static void SavePassengers(List<Passenger> passengers)
50 {
51     string json = JsonConvert.SerializeObject(passengers, Formatting.Indented, settings);
52     File.WriteAllText(PassengersFile, json);
53 }
54
55 // Metoda do odczytu listy pasażerów z pliku
56 public static List<Passenger> LoadPassengers()
57 {
58     if (!File.Exists(PassengersFile))
59     {
60         return new List<Passenger>();
61     }
62     string json = File.ReadAllText(PassengersFile);
63     return JsonConvert.DeserializeObject<List<Passenger>>(json);
64 }
65 }
```

Rysunek 3.1: Przykład wygenerowanego kodu, na bazie zapisu i odczytu pasażerów z pliku JSON.

3.8 Eliminacja cyklicznych referencji

Klasa FileDatabase wykorzystuje ustawienie ReferenceLoopHandling.Ignore, co pozwala uniknąć problemów związanych z cyklicznymi zależnościami w danych.

```
10 private static JsonSerializerSettings settings = new JsonSerializerSettings
11 {
12     ReferenceLoopHandling = ReferenceLoopHandling.Ignore
13 };
```

Rysunek 3.2: Przedstawienie zastosowania w kodzie

W takiej sytuacji ReferenceLoopHandling.Ignore mówi bibliotece JSON, aby zignorować cykliczne referencje podczas przetwarzania danych. Oznacza to, że jeśli natrafi na cykliczną referencję, po prostu ją zignoruje i nie będzie jej analizować, aby uniknąć nieskończonego przetwarzania danych.

3.9 Baza Danych - JSON

Baza Danych - Pliki JSON

Do projektu 'Flight System' nie została użyta tradycyjna baza danych, zamiast tego dane są przechowywane w plikach JSON. Klasy FileDatabase odpowiadają za obsługę tych plików, umożliwiając zapis i odczyt danych. Działa to jako prosty mechanizm składowania danych, który jest łatwo przenośny i nie wymaga konfiguracji bazy danych.

```
1  {
2
3  "DepartureAirport": {
4    "Name": "John F. Kennedy International Airport",
5    "City": "New York",
6    "Country": "USA",
7    "Id": 1
8  },
9  "ArrivalAirport": {
10    "Name": "Los Angeles International Airport",
11    "City": "Los Angeles",
12    "Country": "USA",
13    "Id": 2
14  },
15  "DepartureTime": "2024-01-20T08:00:00",
16  "ArrivalTime": "2024-01-20T11:30:00",
17  "Aircraft": {
18    "EconomySeats": 280,
19    "BusinessSeats": 70,
20    "FirstClassSeats": 16,
21    "Model": "Boeing 747",
22    "TotalSeats": 366,
23    "Id": 1
24  },
25  "Reservations": [
26    {
27      "Ticket": {
28        "Flight": null,
29        "Passenger": {
30          "FirstName": "John",
31          "LastName": "Doe",
32          "Tickets": null,
33          "Id": 0
34        },
35        "SeatNumber": null,
36        "Price": 140.0,
37        "Class": 1,
38        "Id": 0
39      },
40      "ReservationCode": "76ded559-1d21-4a4b-af13-40b29a30435a",
41      "IsConfirmed": false,
42      "Id": 0
43    }
44  ],
```

Rysunek 3.3: Przykład wygenerowanej Bazy Danych w pliku JSON.

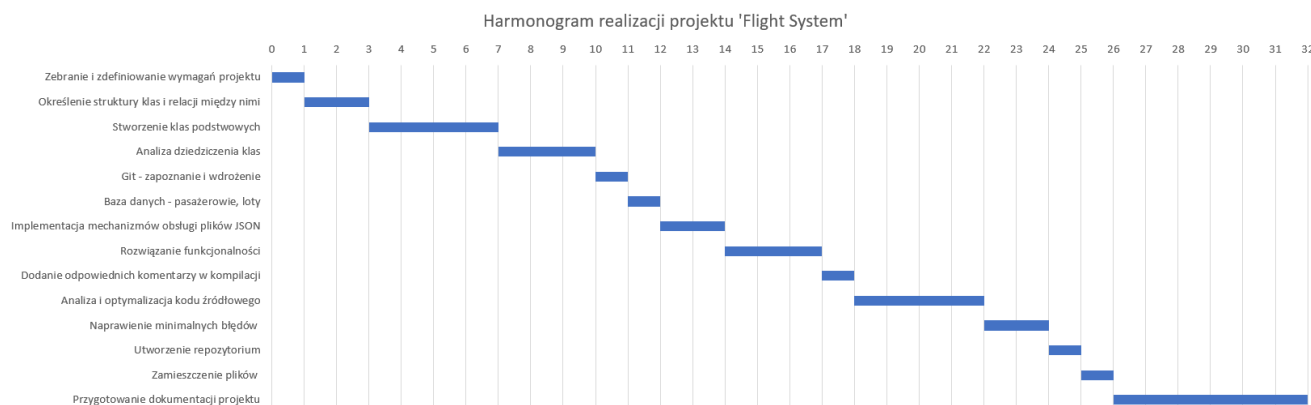
Rozdział 4

Harmonogram Realizacji Projektu

4.1 Diagram Gantta

Wykres Gantta to narzędzie do zarządzania projektami, które ilustruje prace ukończone w okresie czasu w stosunku do czasu zaplanowanego na pracę. Zazwyczaj ma on dwie sekcje: lewa strona przedstawia listę zadań, natomiast prawa strona zawiera oś czasu z paskami harmonogramu, które wizualizują prace.

Poniżej przedstawiam diagram Gantta, przedstawiający harmonogram realizacji projektu 'Flight System'



Rysunek 4.1: Diagram Gantta dla projektu 'Flight System'.

Rozdział 5

Repozytorium i System Kontroli Wersji

5.1 Link do repozytorium

Repozytorium zawiera pliki programu 'Flight System' w języku C# —> https://github.com/kxy220/Projekt_csharp

5.2 System kontroli wersji

W projekcie "FlightSystem" wykorzystuję system kontroli wersji Git, co pozwala nam na efektywne zarządzanie kodem źródłowym. Poniżej przedstawię główne funkcje korzystania z Git, wspomagające pracę przy projekcie.

- **Centralna Baza Danych**

Wszystkie pliki projektu, takie jak kod źródłowy, dokumentacja czy konfiguracje, są przechowywane w repozytorium Git. To centralne źródło gromadzi historię projektu, co ułatwia śledzenie zmian.

- **Branching**

Git pozwala na pracę na różnych gałęziach jednocześnie, co umożliwia równoczesną pracę nad różnymi funkcjonalnościami czy problemami.

- **Bezpieczne Eksperymentowanie**

Każda gałąź to specyficzny obszar do eksperymentów, minimalizujący ryzyko wprowadzenia nieoczekiwanych zmian do głównej linii projektu.

- **Commitowanie Zmian**

Każda zmiana w kodzie jest zapisywana jako commit z dokładnym komunikatem, co ułatwia zrozumienie, co dokładnie zostało zmienione.

- **Znajdowanie Autora**

Git dba o to, abym wiedziała, kto wprowadził daną zmianę, co ułatwia śledzenie wkładu w przypadku pracy w zespole.

- **Merge (Scalanie Gałęzi)**

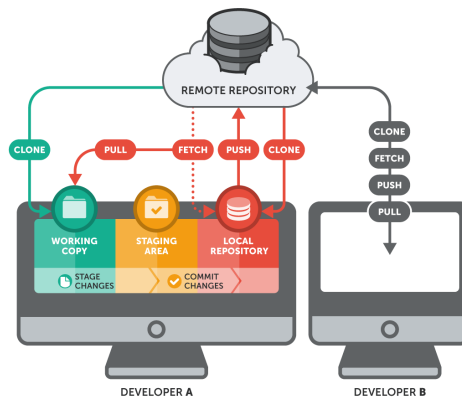
Po zakończeniu prac na danej gałęzi, łatwo mogę połączyć moje zmiany z główną gałęzią projektu, zapewniając integrację nowych funkcji czy poprawek.

- **Historia i Rewizje**

Otrzymuję dostęp do pełnej historii projektu, z informacjami o każdej rewizji, jej autorze, dacie i komunikacie commita.

- **Cofanie i Resetowanie**

W razie potrzeby mogę wrócić do poprzednich rewizji (revert) lub przywrócić projekt do konkretnego stanu (reset).



System Kontroli Wersji 'Git'

Rozdział 6

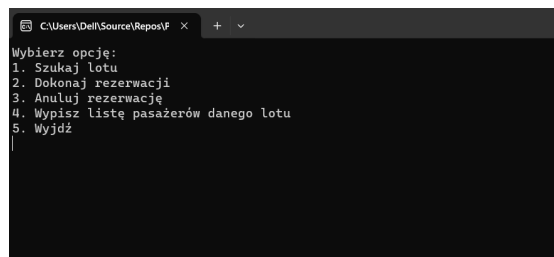
Warstwa Użytkowa Projektu

Warstwa użytkowa w moim projekcie "FlightSystem" została zaprojektowana w celu zapewnienia intuicyjnej interakcji użytkownika z aplikacją konsolową. Poniżej uwzględnię kluczowe elementy oraz fragmenty kodu, które wyjaśniają funkcjonalność.

Program "FlightSystem" wyróżnia się hierarchiczną strukturą, z głównym celem obsługi operacji związanych z zarządzaniem informacjami o lotach. Struktura ta umożliwia użytkownikowi dostęp do funkcji dodawania, usuwania oraz przeglądania danych lotów.

6.1 Menu i obsługa wejścia

Interfejs użytkownika rozpoczyna się od dynamicznego menu prezentującego główne opcje. Obsługa wejścia odbywa się przy użyciu instrukcji warunkowych, kierując użytkownika do odpowiednich funkcji na podstawie dokonanego wyboru.



```
C:\Users\De\Source\Repos\F x + v
Wybierz opcję:
1. Szukaj lotu
2. Dokonaj rezerwacji
3. Anuluj rezerwację
4. Wypisz listę pasażerów danego lotu
5. Wyjdź
```

Rysunek 6.1: Kompilacja Programu w Konsoli.

6.2 Funkcje operacyjne

Kluczowe operacje, takie jak dodawanie, usuwanie oraz przeglądanie lotów, są implementowane jako oddzielne funkcje w celu utrzymania czytelności i modularności kodu. Każda z tych funkcji wywołuje odpowiednie metody klas obsługujących operacje na danych.

```
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103

switch (option)
{
    case 1:
        Console.WriteLine("Podaj miasto odlotu lub przylotu:");
        string city = Console.ReadLine();
        Console.WriteLine("Wprowadź datę (rrrr-mm-dd) lub zostaw puste pole:");
        string dateString = Console.ReadLine();
        DateTime? date = string.IsNullOrEmpty(dateString) ? (DateTime?)null : DateTime.Parse(dateString);
        var matchingFlights = SearchFlights(city, date, flights);
        foreach (var flight in matchingFlights)
        {
            Console.WriteLine($"{flight.DepartureAirport.City} do {flight.ArrivalAirport.City} dnia {flight.DepartureTime}");
        }
        break;
    case 2:
        Console.WriteLine("Podaj ID lotu w celu rezerwacji:");
        flightId = Convert.ToInt32(Console.ReadLine());
        var searchFlight = flights.FirstOrDefault(f => f.Id == flightId);
        MakeReservation(searchFlight, passengers, reservations);
        break;
    case 3:
        Console.WriteLine("Wprowadź kod rezerwacji, którą chcesz anulować:");
        string reservationCode = Console.ReadLine();
        CancelReservation(reservationCode, flights, reservations);
        break;
    case 4:
        Console.WriteLine("Wprowadź ID lotu:");
        flightId = Convert.ToInt32(Console.ReadLine());
        PrintPassengerList(flightId, flights);
        break;
    case 5:
        exit = true;
        FileDatabase.SaveReservations(reservations);
        FileDatabase.SaveFlights(flights);
        FileDatabase.SavePassengers(passengers);
        break;
    default:
        Console.WriteLine("Niepoprawna opcja. Spróbuj ponownie.");
        break;
}
```

Rysunek 6.2: Wykorzystanie instrukcji 'switch' do przeprowadzania operacji na danych.

6.3 Obsługa Bazy Danych

Warstwa użytkowa ściśle współpracuje z klasą FileDatabase, odpowiedzialną za trwałość danych poprzez zapis i odczyt z plików JSON.

Struktura warstwy użytkowej projektu "FlightSystem" została zaprojektowana z priorytetem na modularność, klarowność oraz spójność operacji. To podejście miało na celu nie tylko ułatwienie utrzymania kodu, ale również zapewnienie użytkownikowi końcowemu optymalnej i intuicyjnej obsługi aplikacji. Dzięki temu, projekt finalnie prezentuje się jako efektywne narzędzie, zgodne z standardami.



Rozdział 7

Podsumowanie Projektu i Dalsze Realizacje

7.1 Podsumowanie projektu

Projekt 'FlightSystem' reprezentuje obsługę lotów w środowisku programowania C# i frameworku .NET. Podczas rozwoju tego projektu założeniem była funkcjonalność oraz prostota, oferując praktyczne rozwiązania dla użytkowników programu. Choć nie pretenduje do miana większych projektów, prezentuje funkcjonalność, która może być przydatne w obszarze zarządzania lotami.

Struktura projektu wyróżnia się spójnością kodu, co przekłada się na jego zrozumienie i łatwość utrzymania. Każda z klas pełni precyzyjnie określoną rolę.

Kluczowym fundamentem zarządzania danymi jest klasa FileDatabase, służąca do trwałego przechowywania danych w plikach JSON.

Klasa BaseEntity to fundament projektu, zawierający kluczowe pole Id, które służy jako unikalny identyfikator dla każdego obiektu utworzonego na bazie tej klasy. To pole Id jest esencjalne dla identyfikacji obiektów w systemie.

Dzięki zastosowaniu dziedziczenia, każda klasa pochodna od BaseEntity odziedzicza fundamentalne cechy, jednocześnie zachowując elastyczność wprowadzania unikalnych funkcji dostosowanych do specyfiki danego elementu systemu. Ta struktura projektu zapewnia spójność, umożliwiając łatwą rozbudowę kodu źródłowego. Taka organizacja stanowi kluczowy element dla zrozumienia, utrzymania oraz efektywnego rozwijania projektu.

7.2 Dalsze plany rozwojowe projektu

Rozwój projektu 'FlightSystem' kryje w sobie szereg perspektyw i możliwości, które mogą przyczynić się do jego doskonalenia oraz zwiększenia wartości dla użytkowników. Posiada potencjał do rozbudowy funkcjonalności, uwzględniającej obsługę różnorodnych rodzajów lotów oraz implementację bardziej zaawansowanych algorytmów.

Projekt 'FlightSystem' może znacząco rozwinąć swoją funkcjonalność, poprzez wprowadzenie obsługi różnorodnych rodzajów lotów, zastosowanie wysoko zaawansowanych algorytmów planowania tras oraz integrację z systemami rezerwacji hoteli i transportu. Znacząca jest także rozbudowa modułów obsługi biletów, wprowadzenia systemu lojalnościowego, rozwinięcia systemu płatności i implementacji inteligentnych powiadomień dla pasażerów. Rozbudowa statystyk i raportowania oraz implementacja monitorowania lotów w czasie rzeczywistym to kolejne obszary, które mogą zwiększyć i wzbogacić system, spełniając różnorodne oczekiwania użytkowników i dostosowując się do najnowszych trendów w dziedzinie lotnictwa.

Istotnym obszarem rozwoju będzie podniesienie atrakcyjności interfejsu użytkownika. Kierując się ku bardziej intuicyjnym i estetycznym rozwiązaniom, w projekcie 'FlightSystem' będę dążyć do stworzenia przyjemnego środowiska obsługi dla użytkowników, zapewniając użytkownikom pozytywne odczucia i doświadczenia interakcji z programem.

Bibliografia

- [1] <https://learn.microsoft.com/pl-pl/dotnet/csharp/programming-guide/> z dnia 20.01.2024
- [2] <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-operator> z dnia 20.01.2024
- [3] <https://www.c-sharpcorner.com/article/json-serialization-and-deserialization-in-c-sharp/> z dnia 20.01.2024
- [4] <https://www.youtube.com/watch?v=Y14gG9IJ230> z dnia 20.01.2024
- [5] <https://stackoverflow.com/questions/36128882/method-overloading-c-sharp> z dnia 20.01.2024

Spis rysunków

2.1	Klasa BaseEntity	7
2.2	Klasa Airport	7
2.3	Klasa Passenger	7
2.4	Klasa PassengerAircraft	8
2.5	Klasa Flight	8
2.6	Klasa Ticket	8
2.7	Klasa Reservation	9
2.8	Klasa Statyczna FileDatabase	9
2.9	Przedstawienie diagramu klas programu 'Flight System'	10
3.1	Przykład wygenerowanego kodu, na bazie zapisu i odczytu pasażerów z pliku JSON.	14
3.2	Przedstawienie zastosowania w kodzie	14
3.3	Przykład wygenerowanej Bazy Danych w pliku JSON.	15
4.1	Diagram Gantta dla projektu 'Flight System'.	16
6.1	Kompilacja Programu w Konsoli.	19
6.2	Wykorzystanie instrukcji 'switch' do przeprowadzania operacji na danych.	20