ЗМІСТ

[ВСТУП 2](#_Toc483478178)

[1 ОПИС ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ РОЗРОБКИ 4](#_Toc483478179)

[1.1 Основні принципи об’єктно-орієнтованого програмування 4](#_Toc483478180)

[1.2 Постановка задачі 6](#_Toc483478181)

[1.3 Аналіз технічного завдання на роботу 8](#_Toc483478182)

[1.4 Наслідування 10](#_Toc483478183)

[1.5 Поліморфізм 11](#_Toc483478184)

[1.6 Інкапсуляція 12](#_Toc483478185)

[1.7 Розробка алгоритму роботи програми 13](#_Toc483478186)

[1.8 Структура програмного забезпечення 15](#_Toc483478187)

[Висновок до розділу 16](#_Toc483478188)

[2 ОПИС ІНТЕРФЕЙСУ КОРИСТУВАЧА 17](#_Toc483478189)

[2.1 Опис структур вхідних та вихідних даних 17](#_Toc483478190)

[2.2 Функціональна схема програми 17](#_Toc483478191)

[2.3 Розробка інтерфейсу програми 19](#_Toc483478192)

[Висновок до розділу 22](#_Toc483478193)

[ВИСНОВОК 23](#_Toc483478194)

[СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 24](#_Toc483478195)

[ДОДАТОК А 25](#_Toc483478196)

# ВСТУП

Об'єктно-орієнтоване програмування (ООП) – одна з парадигм програмування, яка розглядає програму як множину «об'єктів», що взаємодіють між собою. Основу ООП складають три основні концепції: інкапсуляція, успадкування та поліморфізм. Одною з переваг ООП є краща модульність програмного забезпечення (тисячу функцій процедурної мови, в ООП можна замінити кількома десятками класів із своїми методами). Незважаючи на те, що ця парадигма з'явилась в 1960 роках, вона не мала широкого застосування до 1990, коли розвиток комп'ютерів та комп'ютерних мереж дозволив писати надзвичайно об'ємне і складне програмне забезпечення, що змусило переглянути підходи до написання програм. Сьогодні багато мов програмування або підтримують ООП (PHP, Lua) або ж є цілком об'єктно-орієнтованими (зокрема, Java, C#, C++, Python, Ruby та Objective-C, ActionScript 3, Swift, Vala).

ООП виникло в результаті розвитку ідеології процедурного програмування, де дані і підпрограми (процедури, функції) їх обробки формально не пов'язані. Для подальшого розвитку об'єктно-орієнтованого програмування велике значення мають поняття події(так зване подієво-орієнтоване програмування) і компоненти (компонентне програмування, КОП).

Об'єктно-орієнтоване програмування сягає своїм корінням до створення мови програмування Симула в 1960 роках, одночасно з посиленням дискусій про кризу програмного забезпечення. Разом із тим, як ускладнювалось апаратне та програмне забезпечення, було дуже важко зберегти якість програм. Об'єктно-орієнтоване програмування частково розв'язує цю проблему шляхом наголошення на модульності програми[1].

На відміну від традиційних парадигм, коли програму розглядали як набір підпрограм, або як перелік інструкцій комп'ютеру, ООП програми можна вважати сукупністю об'єктів. Відповідно до парадигми об'єктно-орієнтованого програмування, кожний об'єкт здатний отримувати повідомлення, обробляти дані, та надсилати повідомлення іншим об'єктам. Кожен об'єкт – своєрідний незалежний автомат з окремим призначенням та відповідальністю[2].

Темою даної курсової роботи є розробка програми для бронювання квитків на потяг на платформі WPF на ООП мові C#, з описом можливих операцій над даними, з використанням подій, які можуть бути прив’язані до об`єктів, керівництва користувача у вигляді пояснювальної записки.

Мета роботи: розглянути й реалізувати систему бронювання квитків на потяг; навчитися вирішувати комплексні задачі за допомогою засобів ООП.

Завдання курсової роботи:

1. вивчити теоретичну основу ООП,
2. проаналізувати предметну область залізнодорожніх перевезень,
3. реалізувати систему бронювання квитків на потяг,
4. провести аналіз роботи сортування.

Сьогодні існує багато програм, які можна використовувати для управління відділом кадрів. Розроблювана програма є подібною до вже існуючих, має простий та зручний інтерфейс та достатньо функцій, що дають змогу користувачеві здійснити все потрібне для його цілей.

Система розроблена на мові програмування C#, яка є новою мовою програмування, орієнтованою на розробку для платформи .NET. Мова має строгу статистичну типізацію, перевантаження операторів, атрибути, події, підтримує поліморфізм, вказівники функції-члени класів, винятки, коментарі у форматі XML. Windows Presentation Foundation (WPF) – це система наступного покоління для побудови клієнтських додатків Windows з візуально привабливими можливостями взаємодії з користувачем. За допомогою WPF можна створювати широкий спектр як автономних, так і розміщених в браузері додатків. WPF, що є компонентом Microsoft.NET Framework 4, надає засоби для створення інтерфейсів користувача, що поєднують мультимедіа, апаратне прискорення, векторну графіку, документи та множину неменше цікавих засобів обробки.

# ОПИС ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ РОЗРОБКИ

## Основні принципи об’єктно-орієнтованого програмування

Об'єктно-орієнтоване програмування – це методологія програмування, заснована на представленні програми у вигляді сукупності взаємодіючих між собою об'єктів, кожен з яких є екземпляром певного класу, а класи можуть утворювати ієрархію спадкування. Програма буде об'єктно-орієнтованої тільки при дотриманні трьох вимог: 1) в якості базових елементів використовуються об'єктивним ти, а не алгоритми; 2) кожен об'єкт є екземпляром якогось певного класу; 3) класи організовані ієрархічно[4].

У центрі ООП знаходиться поняття об'єкта. Об'єкт - це сутність, екземпляр класу, якій можна посилати повідомлення, і яка може на них реагувати, використовуючи свої дані. Дані об'єкта приховані від решти програми. Приховування даних називається інкапсуляціею. Наявність інкапсуляції досить для об'єктно мови програмування, але ще не означає його об'єктної орієнтованості - для цього потрібна наявність успадкування.

Але навіть наявність інкапсуляції і наслідування не робить мову програмування в повній мірі об'єктним з точки зору ООП. Основні переваги ООП проявляються тільки в тому випадку, коли в мові програмування реалізований поліморфізм; тобто можливість об'єктів з однаковою специфікацією мати різну реалізацію

Розрізняють поліморфні і мономорфні мови. Для мономорфних мов характерно те, що використовувані функції, процедури й оператори мають унікальний тип. Поліморфні мови підтримують концепцію поліморфізму в теорії типів, коли одне і те ж ім'я може бути використано для вираження різних дій. Підтримка поліморфізму здійснюється через віртуальні функції, механізм перевантаження функцій і операторів.

Абстрагування - це спосіб виділити набір значущих характеристик об'єкта, виключаючи з розгляду значимі Відповідно, абстракція - це набір всіх таких характеристик. [3]. Концепція абстрактних типів і абстрактних типів даних є ключовою в програмуванні. Абстракція передбачає поділ і незалежний розгляд інтерфейсу і реалізації.

Інтерфейс і внутрішня реалізація є визначальними властивостями об'єктів навколишнього нас світу. Інтерфейс – це засіб взаємодії з деяким об'єктом. Реалізація – це внутрішня властивість об'єкта. Найбільший інтерес представляє ефективність реалізації.

Модульність і абстракція доповнюють один одного. Модульність передбачає приховування деталей реалізації, а абстракція дозволяє уточняти кожен модуль перед тим, як буде написана відповідна програма.

Об'єктно-орієнтоване програмування ґрунтується на трьох основних концепціях: інкапсуляції, поліморфізму і наслідування.

Інкапсуляція (пакетування) являє собою механізм, зв'язуючий разом дані і код, що обробляє ці дані, і який зберігає їх від зовнішнього впливу і помилкового використання. інкапсуляція позволяє створювати об'єкт, який є логічним цілим, що включає дані і код для роботи з цими даними. Об'єкт забезпечує захист проти випадкової або несанкціонованої модифікації приватних (private) складових його членів. Закриті дані або коди (методи) доступні тільки для інших частин цього об'єкту і недоступні поза ним. Відкрита частина об'єкта призначена для забезпечення контрольованого інтерфейсу його закритій частині.

Поліморфізм забезпечує можливість реагувати по-різному на одне й те саме повідомлення (виклик функції-члена). Поліморфізм дозволяє зменшити складність програми за допомогою використання одного і того ж інтерфейсу для завдання цілого класу дій. Підтримка поліморфізму в ООП здійснюється через віртуальні функції та механізм перевантаження і перевизначення.

Ключовим в розумінні поліморфізму є те, що він дозволяє маніпулювати об'єктами різного ступеня складності шляхом створення спільного для них стандартного інтерфейсу для реалізації схожих дій.

Наслідування є механізмом, завдяки якому новий (похідний) клас може створюватися, наслідуючи (набуваючи) властивості від вже існуючого (базового) класу. Новий клас, використовуючи успадкування, потребує тільки в визначенні специфічних тільки для цього класу компонентів. Спадкування дозволяє підтримувати концепцію ієрархії класів.

## Постановка задачі

Для виконання курсового проекту передбачено функціональні та  
нефункціональні вимоги до програмного забезпечення, а також вимоги до логічної структури даних. Розглянемо ці вимоги.

Спрощені вимоги до програмного забезпечення

Функціональні вимоги:

1. Управління потягами
   1. Можливість додавати потяг
   2. Можливість видаляти потяг
   3. Можливість переглянути всі потяги
   4. Можливість переглянути дані конкретного потягу
   5. Можливість переглянути вагони потягу з відсотками заброньованих в них місць
2. Управління вагонами
   1. Можливість додавати вагон до потягу
   2. Можливість видаляти вагон з потягу (тільки якщо в ньому не має заброньованих місць)
   3. Можливість переглянути зайняті (не зайняті) місця вагону
3. Управління бронювання
   1. Можливість додавати бронювання
   2. Можливість видаляти бронювання
   3. Можливість змінити бронювання
   4. Можливість переглянути інформацію про бронювання
4. Пошук
   1. Можливість пошуку по ключовому слову серед потягів
   2. Можливість пошуку броні по даті бронювання
   3. Пошук постачальника, у якого закуплено товару на найбільшу суму
5. Побудувати графіки щодо інформації про заброньовані місця протягом тижня на довільній потяг.

Нефункціональні вимоги:

* 1. Зовнішній інтерфейс повинен бути віконним
  2. Повинні бути реалізовані основні принципи ООП – поліморфізм, наслідування та інкапсуляція
  3. Дані повинні зберігатися у файлах після виходу з програми та зчитуватись
  4. Система має забезпечити коректність ведених даних

Логічна структура даних

Таблиця 1.1 – Інформаційний об’єкт потяг

|  |  |  |  |
| --- | --- | --- | --- |
| Елемент даних | Тип | Опис | Коментар |
| Номер | Текст | Номер потягу | - |
| Призначення | Текст | Звідки і куди їже потяг | - |
| Час відправлення | Дата | Час відправлення потягу | - |
| Час прибуття | Дата | Час прибуття потягу | - |
| Кількість вагонів | Ціле число | Кількість вагонів у потягу | - |
| Заброньовані місця | Дробове число | Кількість заброньованих місць у кожному вагоні |  |

Таблиця 1.2 – Інформаційний об’єкт вагон

|  |  |  |  |
| --- | --- | --- | --- |
| Елемент даних | Тип | Опис | Коментар |
| Загальна кількість місць | Ціле число | Загальна кількість місць у вагоні | - |
| Кількість зайнятих місць | Ціле число | Кількість зайнятих місць у вагоні | - |
| Кількість вільних місць | Ціле число | Кількість вільних місць у вагоні | - |

Таблиця 1.3 – Інформаційний об’єкт бронювання

|  |  |  |  |
| --- | --- | --- | --- |
| Елемент даних | Тип | Опис | Коментар |
| Ім’я | Текст | Ім’я клієнта | - |
| Прізвище | Текст | Прізвище клієнта | - |
| Номер потягу | Ціле число | Номер потягу, де заброньоване місце | - |
| Призначення | Текст | Звідки і куди їде потяг | - |
| Час відправлення | Дробове число | Час відправлення потягу |  |
| Вагон | Ціле число | Номер вагону |  |
| Місця | Список | Заброньовані місця |  |
| Дата | Дата | Дата бронювання |  |

## Аналіз технічного завдання на роботу

Для реалізації завдання в роботі було реалізовано два типи класів бізнес моделей. Перший тип використовується на рівні бази даних, при збережені в SQLite. Другий тип використовується на рівні бізнес логіки.

Розглянемо класи Car, CarSeat, Train, Reservation, які відповідають нашим бізнес об’єктам Вагон, Місце в вагоні, Потяг, Бронювання. Ці класи представляють собою моделі бізнес логіки та фундаментальні дані, необхідні для роботи додатку. Реалізація описана нижче:

public class CarSeat

{

public int SeatNumber { get; set; }

public bool IsReserved { get; set; }

public Reservation Reservation { get; set; }

}

Аналізуючи предметну область було вирішено ввести модель місця в вагоні (CarSeat) для можливості зв’язування бронювання з конкретним місцем. У полі Reservation міститься посилання на об’єкт бронювання. Якщо місце ще не заброньоване то значення цього поля буде містити null.

public class Car: BaseModel

{

public int CarNumber { get; set; }

public int SeatsCount { get; set; }

public int ReservationsCount { get; set; }

public int AvailableSeatsCount { get; set; }

public List<CarSeat> Seats { get; set; }

}

Модель Car відповідає інформаційному об’єкту Вагон. Поля SeatsCount, ReservationsCount, AvailableSeatsCount відповідають загальній кількісті місць, кількості зайнятих та заброньованих місць. Також було додано поле Seats, в якому зберігається список об’єктів місць які відносяться до цього вагону.

public class Train : BaseModel

{

public int Number { get; set; }

public string DeparturePoint { get; set; }

public string ArrivalePoint { get; set; }

public DateTime DepartureDate { get; set; }

public DateTime ArrivalDate { get; set; }

public List<Car> Cars { get; set; }

public Train(TrainEntity entity)

{

Id = entity.Id;

Number = entity.Number;

DeparturePoint = entity.DeparturePoint;

ArrivalePoint = entity.ArrivalePoint;

ArrivalDate = entity.ArrivalDate;

DepartureDate = entity.DepartureDate;

}

}

Модель Train відповідає інформаційному об’єкту потяг. Поле Cars містить список вагонів які відповідають цьому потягу. Всі інші поля відповідають   
елементам інформаційного об’єкту. Середня кількість заброньованих місць у   
кожному вагоні буде вираховуватись під час виконання програми. Конструктор з параметром дозволяє зручно переводити бізнес модель в модель, яка буде зберігатися в базі даних.

Для реалізації збереження стану даних при выході з додатку, було використано SQLite. Роботу з базою даних SQLite було реалізовано в класі Database який створює таблиці при старті та має загально реалізацію запису, зчитування, видалення та обновлення даних. Будь яка з цих операцій буде блокувати таблицю бази даних доки вона не буде завершена. Для можливості зберігання даних в SQLite було реалізовано дублікати всіх моделей, які використовуються тільки при роботі з бази даних. В цих моделях відсутні зв’язки між об’єктами, а тому на рівні бізнес логіки реалізовані спеціальні методи які створюють необхідні нам зв’язки. Так з’єднуються моделі CarSeat з моделлю Car, моделі Car з моделлю Train.

## Наслідування

Підкласи успадковують атрибути та поведінку своїх батьківських класів, і можуть вводити свої власні. Успадкування може бути одиничне (один безпосередній батьківський клас) та множинне (кілька батьківських класів).

Для оголошення наслідування потрібно використати наступну структуру:

Модифікатор доступу class Унаслідований клас: Базовий клас

{

//дані і функціональні члени

}

В курсовій роботі наслідування було використано для виділення загальної реалізації роботи з базою, а також для виділення базових частин складних об’єктів. Приклади наслідування наведені нижче.

public class Database: SQLiteConnection

{

...

}

Класс Database успадкований від класу SQLiteConnection, який містить реалізацію роботи с файловою системою, та базові CRUD операції по роботі з даними в таблицях.

public class UserEntity: BaseEntity

{

...

}

public class CarSeatEntity : BaseEntity

{

...

}

public class PassengerTrainCarEntity: TrainCarEntity

{

...

}

public class ReservationEntity: BaseEntity

{

...

}

public class BaseEntity

{

[PrimaryKey, AutoIncrement]

public int Id { get; set; } = 0;

}

Всі моделі для роботи з базою даних мають спільний базовий клас BaseEntity, який містить поле Id, яке використовується як первинний ключ в таблицях, це дозволили зробити загальну реалізацію для операцій запису, зчитування, оновлення та видалення елементів.

## Поліморфізм

. Поліморфізм дозволяє писати більш абстрактні програми і підвищити коефіцієнт повторного використання коду. У даній роботі поліморфізм реалізований за допомогою загальних методів, на рівні роботи з базою даних. Приклад реалізований у наступній частині коду:

private static object locker = new object();

public IEnumerable<T> GetItems<T>() where T : IBusinessEntity, new()

{

return (from i in Table<T>() select i).ToList();

}

public T GetItem<T>(int id) where T : IBusinessEntity, new()

{

return Table<T>().First<T>(x => x.ID == id);

}

public int SaveItem<T>(T item) where T : IBusinessEntity

{

lock (locker)

{

if (item.ID != 0)

{

Update(item);

return item.ID;

}

return Insert(item);

}

}

public int DeleteItem<T>(int id) where T : IBusinessEntity, new()

{

lock (locker)

{

return Delete<T>( id );

}

}

Узагальнення методів на рівні бази даних дозволило повторно використовувати код роботи з базою даних для будь яких моделей успадкованих від BaseEntity.

Також при розробці архітектури було вирішено використовувати шаблон проектування Inversion Of Control (Інверсія управління). Він дозволяє значно зменшити час на ініціалізацію об’єктів та зменшити час розробки.

При використанні інверсії управління модулі верхніх рівнів не повинні залежати від модулів нижчих рівнів. Обидва типи модулів повинні залежати від абстракцій. Абстракції не повинні залежати від деталей. Деталі повинні залежати від абстракцій. Так наприклад клас TrainManager залежить від 4 компонентов, які в свою чергу також може залежати від інших компонентів. Приклад організації залежностей продемонстрований у наступному коді:

public class TrainManager: ITrainManager

{

private ITrainsRepository trainsRepository;

private IPassengerTrainCarsRepository carsRepository;

private ICarSeatsRepository carSeatsRepository;

private IReservationRepository reservationRepository;

public TrainManager(IPassengerTrainCarsRepository carsRepository,

ITrainsRepository trainsRepository,

IReservationRepository reservationRepository,

ICarSeatsRepository carSeatsRepository)

{

this.carsRepository = carsRepository;

this.trainsRepository = trainsRepository;

this.reservationRepository = reservationRepository;

this.carSeatsRepository = carSeatsRepository;

}

}

Приклад реєстрації залежностей продемонстрованій у наступному коді:

var container = TinyIoCContainer.Current;

container.Register(new Database(Environment.CurrentDirectory +"/trains.db"));

container.Register<IUsersRepository, UsersRepository>();

container.Register<ICarSeatsRepository, CarSeatsRepository>();

container.Register<IReservationRepository, ReservationsRepository>();

container.Register<ICarSeatsRepository, CarSeatsRepository>();

container.Register<ITrainsRepository, TrainsRepository>();

container.Register<ITrainManager, TrainManager>();

var trainRepository = continer.Resolve<ITrainManager>();

В прикладі вище метод контейнеру Register використовується для реєстрації типів а метод Resolve відповідно для створення об’єктів.

## Інкапсуляція

Інкапсуляція - це принцип, згідно з яким будь-який клас повинен розглядатися як чорний ящик - користувач класу повинен бачити і використовувати тільки інтерфейсну частину класу (тобто список декларованих властивостей і методів класу) і не вникати в його внутрішню реалізацію. Тому дані прийнято інкапсулювати в класі таким чином, щоб доступ до них з читання або запису здійснювався не безпосередньо, а за допомогою методів. Принцип інкапсуляції (теоретично) дозволяє мінімізувати кількість зв'язків між класами і, відповідно, спростити незалежну реалізацію та модифікацію класів. Інкапсуляція використана на різних архітектурних рівнях для приховування роботи з компонентами нижніх рівнів. Так наприклад в класах менеджерах, які реалізовують бізнес логіку з даними є поля інтерфейсів репозиторіїв з модифікатором доступу private, які передаються через конструктор класу. Приклад представлений у коді нижче

public class TrainManager: ITrainManager

{

private ITrainsRepository trainsRepository;

private IPassengerTrainCarsRepository carsRepository;

public TrainManager(IPassengerTrainCarsRepository carsRepository, ITrainsRepository trainsRepository)

{

this.carsRepository = carsRepository;

this.trainsRepository = trainsRepository;

}

}

## Розробка алгоритму роботи програми

Програма існує тоді, коли вона знайшла своє застосування у споживача. А програма знаходить своє застосування тоді, коли вона спрощує певну кількість роботи і водночас не вимагає великих зусиль при своєму застосуванні. Програма має такі властивості лише тоді, коли постійно удосконалюється. Програма, що розробляється, завжди записується за допомогою блок-схем – це алгоритм виконання програми у вигляді блоків, геометричних фігур, які відповідають за певні дії чи блоки дій у програмі, тобто блок-схема це представлення задачі для її аналізу або розв'язування за допомогою спеціальних символів (геометричних образів), які позначають такі елементи, як операції, потік, дані тощо.

Наприклад фігура у вигляді еліпса відповідає за початок та кінець алгоритму, паралелограм – введення і виведення даних, блок прийняття рішень має вигляд ромба, блок обчислень – прямокутник.

Завдяки блок-схемі, набагато швидше, можна знайти необхідний блок програми і удосконалити його, або просто перевірити на помилки. Схема головного модуля програми, що розробляється у курсовому проекті, зображена на рисунку 1.1.



Рисунок 1.1 – Алгоритм роботи з головним модулем програми

## Структура програмного забезпечення

Модульне програмування має на увазі організацію програми як сукупності незалежних блоків невеликих розмірів, які прийнято іменувати модулями, поведінку і структура яких повністю підпорядковані цілком певним принципам.

Застосування модульного програмування дає можливість спростити тестування програми та своєчасне виявлення помилок.

Для зручності підтримки додатку ядро програми було розділено на декілька рівнів:

* рівень роботи з базою даних;
* рівень роботи з даними;
* рівень представлення даних.

Кожному з цих рівнів відповідає свій набір класів.

Таблиця 1.4 – Основні вікна додатку та їх призначення.

|  |  |
| --- | --- |
| Назва модуля | Призначення модуля |
| MainWindow.xaml | Головне вікно(контейнер) для дочірніх представлень. |
| Trains.xaml | Дочірнє представлення поїздів |
| Car.xaml | Дочірнє представлення вагонів поїзду |
| Statistic.xaml | Дочірнє представлення для статистики |
| Reservations.xaml | Дочірнє представлення резервацій |
| ReservationDetails.xaml | Дочірнє представлення резервації |
| TrainDetails.xaml | Дочірнє представлення поїзду |

Таблиця 1.5 – Призначення об’єктів рівня роботи з даними

|  |  |
| --- | --- |
| Назва модуля | Призначення модуля |
| StatisticManager | Аналіз статистики бронювання по кожному потягу |
| TrainsManager | Отримання повного списку потягів, з’єднування об’єктів вагонів і місць в вагоні з об’єктом потягу |
| ReservationManager | Отримання повного списку резервацій, знаходження резервацій по потягу та вагону |

Таблиця 1.6 – Призначення об’єктів рівня роботи з базою даних

|  |  |
| --- | --- |
| Назва модуля | Призначення модуля |
| Database | Робота з файловою системою, містить |
| BaseRepository | Загальна реалізація CRUD операцій |
| TrainsRepository | Реалізація CRUD операцій для поїздів |
| CarRepository | Реалізація CRUD операцій для вагонів |
| UserRepository | Реалізація CRUD операцій для клієнтів |
| CarSeatRepository | Реалізація CRUD операцій для місць в вагоні |

Таблиця 1.7 – Призначення об’єктів рівня роботи з даними

|  |  |
| --- | --- |
| Назва модуля | Призначення модуля |
| StatisticViewModel | Відображення даних статистики на вікні статистики резервування |
| TrainsViewModel | Відображення даних про потяги на вікні управлінні потягами, логіка роботи вікна додавання та редагування потягів |
| ReservationViewModel | Відображення даних про резервації на вікні вагону, резервацій та пошуку |

## Висновок до розділу

У першому розділі розглянуто основи об’єктно-орієнтованого програмування, принцип його дії та як це реалізовані в програмі, що розробляється. Було розглянуто три парадигми ООП. Наведені приклади застосування цих принципів при розробці нашої програми. При аналізі технічного завдання, нами розроблені основні класи та методи класів. Також, нами розроблено алгоритм роботи програми, по якому ми і конструювали логіку нашої програми. Розроблено структуру програмного забезпечення та описані основні модулі. Важлива особливість модулів є те, що компілятор розміщує їх програмний код в окремому сегменті пам`яті.

# ОПИС ІНТЕРФЕЙСУ КОРИСТУВАЧА

## Опис структур вхідних та вихідних даних

Розглянемо дані, які поступають при запуску программи. На входу про-грами потрібно авторизуватися. Після авторизації на вхід поступають дані про робітників, посади, проекти та підрозділи.

Після входу в програму відкривається головне вікно програми з відкритою вкладкою списку потягів. Завантажуються додані раніше потяги з бази даних. На даній вкладці є кнопка додавання нового потягу, яка відкриє діалог, в якому можно вести дані поїзду та додати його. Також можна видалити поїзд з бази даних, це видалить всі вагони та резервації цього поїзду з база даних.

Для того щоб додати поїзд необхідно обов’язково зазначити номер потягу, дату відправлення та прибуття, та місце відправлення та призначення. Якщо якесь поле не введено то покажеться діалог про необхідність заповнення поля.

Аналогічно додаються вагони в поїзд та місця до вагону.

Також на вкладці Пошук користувач може ввести місце призначення або відправлення і відповідні поїзди покажуться в таблиці.

На виході програми дані зберігаються у базу даних.

## Функціональна схема програми

Функціональна схема призначена для роз'яснення процесів, що відбуваються в окремих функціональних колах програми або програмі в цілому. Для складнї програми розробляється кілька функціональних схем, що пояснюють процеси, що відбуваються при різних передбачених режимах роботи. Кількість функціональних схем, що розробляються на програму, ступінь їх деталізації і обсяг розміщених відомостей визначається розробником з урахуванням особливостей програми. Схему зображена на рис. 2.1.

На схемі зображують функціональні частини програми (елементи, функції, функціональні групи) і зв'язку межу ними. Графічна побудова схеми має наочно відображати послідовність функціональних процесів, що відбуваються у програмі.



Рисунок 2.1 – Функціональна схема програми

## Розробка інтерфейсу програми

Інтерфейс користувача ([англ.](https://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) *user interface*, дружній [інтерфейс](https://uk.wikipedia.org/wiki/%D0%86%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81)) — сукупність засобів, методів і правил взаємодії (управління, контролю і т. д.) між елементами системи [7]. Засіб зручної взаємодії користувача з інформаційною системою. Сукупність засобів для обробки та відображення інформації, максимально пристосованих для зручності користувача; у графічних системах інтерфейс користувача реалізовується багатовіконним режимом, змінами кольору, розміру, видимості (прозорість, напівпрозорість, невидимість) вікон, їхнім розташуванням, сортуванням елементів вікон, гнучкими налаштовуваннями як самих вікон, так і окремих їхніх елементів (файли, папки, ярлики, шрифти тощо), доступністю багатокористувацьких налаштувань.

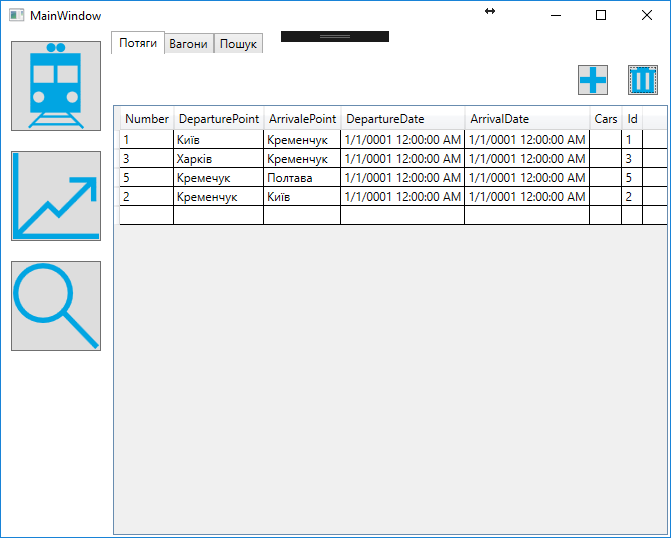


Рисунок 2.2 – Головне вікно програми

Основне вікно програми, що показано на рисунку 2.2, відкривається на вкладці «Потяги», що дозволяє переглянути всі потяги, редагувати, додавати та видаляти потяги. На цій вкладці є кнопка додавання потягу, яка показана на рисунку 2.3. Вона відкриває діалог додавання нового поїзду, який показаний на рисунку 2.4. Також на рисунку 2.3 можна побачити кнопку видалення потягу, яка видалить виділений в таблиці потяг.

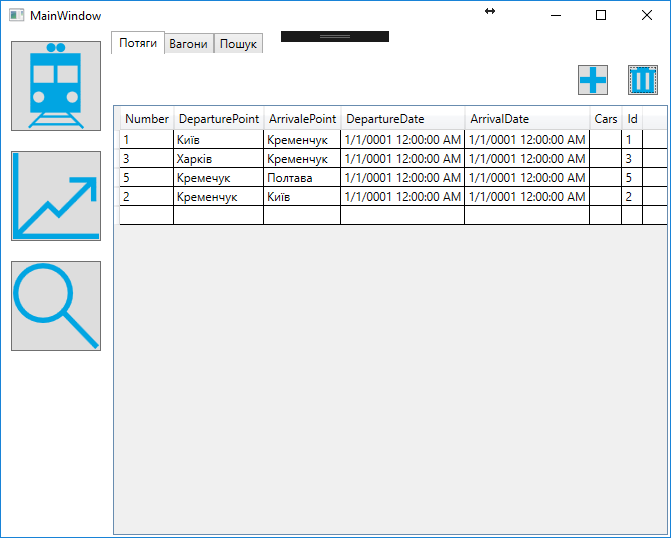


Рисунок 2.3 – Вікно додавання нового потягу

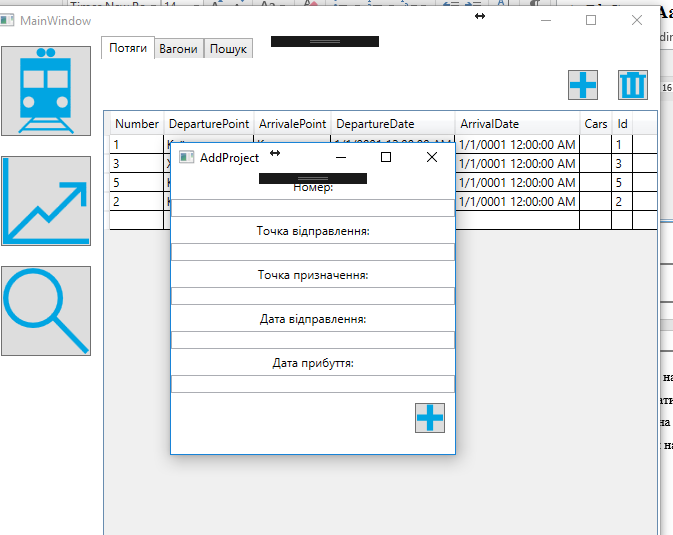


Рисунок 2.4 – Вікно додавання нового потягу

При подвійному кліку по записі потяга в таблиці покажеться діалог редагування потягу, в якому можна змінити дані конкретного потягу. Приклад діалогу показаний на рисунку 2.5.

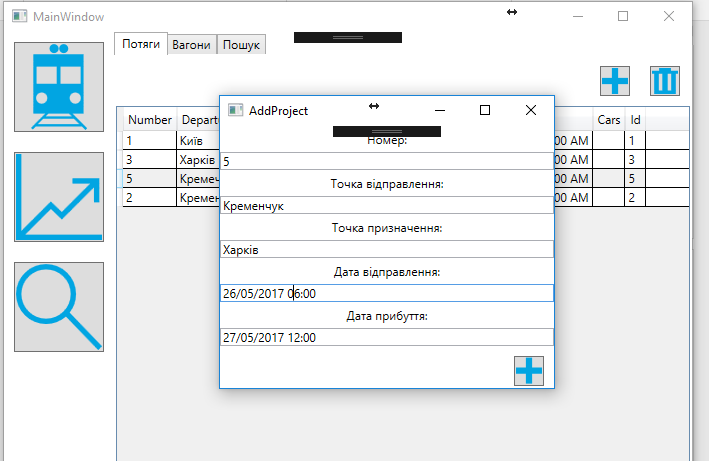


Рисунок 2.5 – Вікно редагування даних потягу

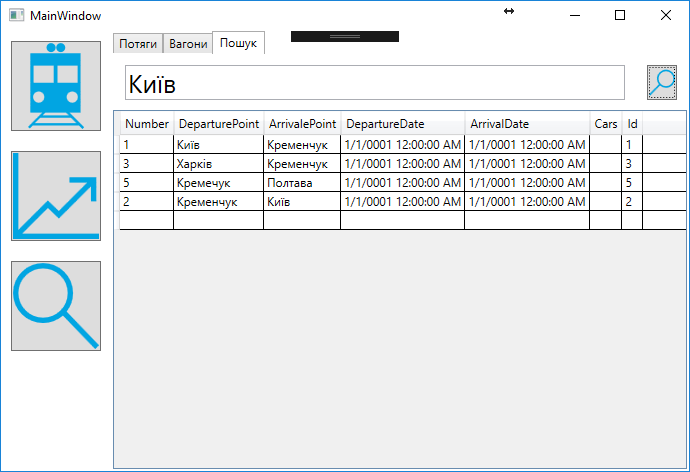


Рисунок 2.6 – Вікно пошуку потягів

На рисунку 2.6 показана вкладка пошуку потягів. На ній є можливість пошуку по місцю відправлення та призначення. Також при подвійному кліку на запис потягу в таблиці викликається діалог додавання нового бронювання, яке показано на рисунку 2.7.

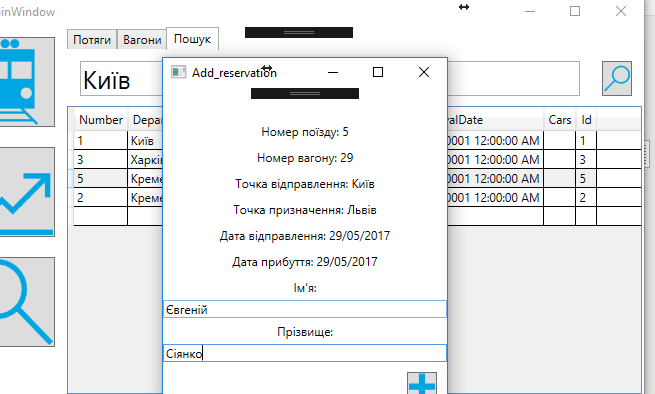


Рисунок 2.7 – Діалог додавання резервації

## Висновок до розділу

У другому розділі описано структури вхідних та вихідних даних. Розроблено функціональну схему програми «Системи бронювання квитків на потяг», та спроектовано програмний інтерфейс.

Інтерфейс розроблено для зручної роботи робітників системи залізнодорожних перевезень. Простота управління здійснюється завдяки інтуїтивно зрозумілому меню.

Реалізацію можна вважати завершеною. Повний код програми знаходиться в Додатку А.

# ВИСНОВОК

У ході роботи було поглиблено вивчено концепції об’єктно-орієнтованого програмування в .NetFramework. Для розробки інтерфейсу програми використовувалися базові інтерфейсні компоненти платформи WPF.

Дана програма розроблена для робітників, що працюють у сфері залізнодорожних перевезень. Інтерфейс був розроблений таким чином. щоб навіть непідготованому користувачеві було легко та зручно з ним працювати. При виконанні курсової роботи виконано всі вимоги, що поставлені у технічному завданні.

Розглядаючи готову програму, слід наголосити, що вона подана у вигляді сукупності взаємодіючих об’єктів, кожен з яких є екземпляром певного класу, а класи є членами певної ієрархії наслідування. Дана програма розроблена в середовищі Visual Studio 2015 на об’єктно-орієнтованій мові програмування з безпечною системою типізації для платформи .NET C#.

# СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Медведев В.І. Особливості об’єктно-орієнтованого програмування на C++/CLI, C# i Java/ В. І. Медведев –Київ: РІЦ «Школа», 2010. – 96с.
2. Буч Г. Історія ООП / Г. Буч, А. Роберт. – 2-ге вид., стер. – Пітер : Юрінком Інтер, 2012. – 46 с.
3. Кознов, В. В. Програмирование C# / В. В. Кознов – Москва: СИНТЕГ, 2006. – 958 с.
4. Кормен Т. Алгоритмы. Вводный курс / Т. Комен, Ч. Лейзерсон – Москва: МЦНМО, 2002. – 208 с.
5. Левитин А. Алгоритмы. Введение в разработку и анализ / А. Левитин – Москва: Вильямс, 2006. – 576 с
6. Трахтенгерц Э. А. Реализация решений / Э. А. Трахтенгерц. – Москва: СИНТЕГ, 2009. – 224 с.
7. Окулов С. Программирование в алгоритмах / С. Окулов – Москва: Бином, 2004. – 607 с.
8. Макконнелл Дж. Основы современных алгоритмов / Дж. Макконнелл  
   – Москва: Техносфера-информ, 2003. – 709 с.
9. Медведев В.І. Особливості об’єктно-орієнтованого програмування на C++/CLI, C# i Java/ В. І. Медведев –Київ: РІЦ «Школа», 2010. – 96с.
10. Мак-Дональд М. WPF: WindowsPresentationFoundation в .NET 4.0 з прикладами на C # 2011 / Метью Мак-Дональд., 2011. – 20 с.
11. Натан А. WPF 4. Докладне керівництво / Адам Натан., 2010. – 83 с.

# ДОДАТОК А

Код програми

using System;

using System.Collections.Generic;

using System.Linq;

using SQLite;

using Trains.Core.BL.Models;

using Trains.Core.DAL.Entities;

using Trains.Core.DAL.Entities.Common;

namespace Trains.Core.DL

{

public class Database : SQLiteConnection

{

private static object locker = new object();

public Database(string path) : base(path)

{

CreateTable<CarSeatEntity>();

CreateTable<ReservationEntity>();

CreateTable<PassengerTrainCarEntity>();

CreateTable<TrainEntity>();

CreateTable<UserEntity>();

}

public IEnumerable<T> GetItems<T>() where T : BaseEntity, new ()

{

lock (locker)

{

return (from i in Table<T>() select i).ToList();

}

}

public IEnumerable<T> FindAll<T>(Func<T, bool> predicate) where T : BaseEntity, new()

{

lock (locker)

{

return Table<T>().Where<T>(predicate);

}

}

public T GetItem<T>(int id) where T : BaseEntity, new()

{

lock (locker)

{

return Table<T>().First<T>(x => x.Id == id);

}

}

public int SaveItem<T>(T item) where T : BaseEntity

{

lock (locker)

{

if (item.Id != 0)

{

Update(item);

return item.Id;

}

else

{

return Insert(item);

}

}

}

public int DeleteItem<T>(int id) where T : BaseEntity, new()

{

lock (locker)

{

return Delete<T>( id );

}

}

public int DeleteItem<T>(T item) where T : BaseEntity

{

return DeleteItem(item);

}

/// <summary>

/// Delete every object from Table that fits the condition of predicate.

/// </summary>

/// <param name="predicate">

/// Condition for items that need to be removed.

/// </param>

/// <returns>

/// Count of deleted items

/// </returns>

/// <exception cref="Exception">Thrown when delete transaction failed</exception>

public int DeleteGroupBy<T>(Func<T,bool> predicate) where T : BaseEntity, new()

{

lock (locker)

{

var count = 0;

BeginTransaction();

try

{

var listOfItems = FindAll<T>(predicate).ToList();

for (int i = 0; i < listOfItems.Count; i++, count++)

{

Delete(listOfItems[i]);

}

Commit();

return count;

}

catch (Exception ex)

{

Rollback();

throw;

}

}

}

public int DeleteGroup<T>(IList<T> group) where T : BaseEntity, new()

{

lock (locker)

{

var count = 0;

BeginTransaction();

try

{

for (int i = 0; i < group.Count; i++, count++)

{

Delete(group[i]);

}

Commit();

return count;

}

catch (Exception ex)

{

Rollback();

throw;

}

}

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using SQLite;

using Trains.Core.BL.Models;

using Trains.Core.DAL.Entities;

using Trains.Core.DAL.Entities.Common;

namespace Trains.Core.DL

{

public class Database : SQLiteConnection

{

private static object locker = new object();

public Database(string path) : base(path)

{

CreateTable<CarSeatEntity>();

CreateTable<ReservationEntity>();

CreateTable<PassengerTrainCarEntity>();

CreateTable<TrainEntity>();

CreateTable<UserEntity>();

}

public IEnumerable<T> GetItems<T>() where T : BaseEntity, new ()

{

lock (locker)

{

return (from i in Table<T>() select i).ToList();

}

}

public IEnumerable<T> FindAll<T>(Func<T, bool> predicate) where T : BaseEntity, new()

{

lock (locker)

{

return Table<T>().Where<T>(predicate);

}

}

public T GetItem<T>(int id) where T : BaseEntity, new()

{

lock (locker)

{

return Table<T>().First<T>(x => x.Id == id);

}

}

public int SaveItem<T>(T item) where T : BaseEntity

{

lock (locker)

{

if (item.Id != 0)

{

Update(item);

return item.Id;

}

else

{

return Insert(item);

}

}

}

public int DeleteItem<T>(int id) where T : BaseEntity, new()

{

lock (locker)

{

return Delete<T>( id );

}

}

public int DeleteItem<T>(T item) where T : BaseEntity

{

return DeleteItem(item);

}

/// <summary>

/// Delete every object from Table that fits the condition of predicate.

/// </summary>

/// <param name="predicate">

/// Condition for items that need to be removed.

/// </param>

/// <returns>

/// Count of deleted items

/// </returns>

/// <exception cref="Exception">Thrown when delete transaction failed</exception>

public int DeleteGroupBy<T>(Func<T,bool> predicate) where T : BaseEntity, new()

{

lock (locker)

{

var count = 0;

BeginTransaction();

try

{

var listOfItems = FindAll<T>(predicate).ToList();

for (int i = 0; i < listOfItems.Count; i++, count++)

{

Delete(listOfItems[i]);

}

Commit();

return count;

}

catch (Exception ex)

{

Rollback();

throw;

}

}

}

public int DeleteGroup<T>(IList<T> group) where T : BaseEntity, new()

{

lock (locker)

{

var count = 0;

BeginTransaction();

try

{

for (int i = 0; i < group.Count; i++, count++)

{

Delete(group[i]);

}

Commit();

return count;

}

catch (Exception ex)

{

Rollback();

throw;

}

}

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using Trains.Core.BL.Contracts;

using Trains.Core.BL.Models;

using Trains.Core.DAL.Entities;

using Trains.Core.DAL.Repositories.Contracts;

namespace Trains.Core.BL

{

public class TrainManager: ITrainManager

{

private ITrainsRepository trainsRepository;

private IPassengerTrainCarsRepository carsRepository;

private ICarSeatsRepository carSeatsRepository;

private IReservationRepository reservationRepository;

public TrainManager(IPassengerTrainCarsRepository carsRepository,

ITrainsRepository trainsRepository,

IReservationRepository reservationRepository,

ICarSeatsRepository carSeatsRepository)

{

this.carsRepository = carsRepository;

this.trainsRepository = trainsRepository;

this.reservationRepository = reservationRepository;

this.carSeatsRepository = carSeatsRepository;

}

public List<Train> GetAllTrains()

{

return trainsRepository.GetAll().OrderBy(t => t.DepartureDate).Select(train=>new Train(train)).ToList();

}

public void AddTrain(Train train)

{

trainsRepository.Save(new TrainEntity

{

Id = train.Id,

Number = train.Number,

DeparturePoint = train.DeparturePoint,

ArrivalePoint = train.ArrivalePoint,

ArrivalDate = train.ArrivalDate,

DepartureDate = train.DepartureDate

});

}

public void RemoveTrain(int id)

{

trainsRepository.Delete(id);

}

public void GetTrain(int id)

{

trainsRepository.GetById(id);

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Navigation;

using System.Windows.Shapes;

using TinyIoC;

using Trains.Core.BL;

using Trains.Core.BL.Models;

using Trains.WPF.View;

namespace Trains.WPF

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

private TrainManager manager;

private List<Train> trains;

public MainWindow()

{

InitializeComponent();

manager = TinyIoCContainer.Current.Resolve<TrainManager>();

trains = manager.GetAllTrains();

projectDataGrid.ItemsSource = trains;

projectDataGrid.MouseDoubleClick += open\_Click;

searchdata.ItemsSource = trains;

searchdata.MouseDoubleClick += Searchdata\_MouseDoubleClick;

}

private void Searchdata\_MouseDoubleClick(object sender, MouseButtonEventArgs e)

{

var addvindow = new Add\_reservation();

addvindow.ShowDialog();

}

private void button\_Click(object sender, RoutedEventArgs e)

{

var addvindow = new AddProject();

addvindow.ShowDialog();

addvindow.Closed += (a, t) =>

{

projectDataGrid.ItemsSource = manager.GetAllTrains();

};

}

private void open\_Click(object sender, RoutedEventArgs e)

{

if (projectDataGrid.SelectedIndex >= 0)

{

var addvindow = new AddProject();

addvindow.ShowDialog();

addvindow.Closed += (a, t) =>

{

projectDataGrid.ItemsSource = manager.GetAllTrains();

};

addvindow.Train = trains[projectDataGrid.SelectedIndex];

}

}

private void button\_Delete(object sender, RoutedEventArgs e)

{

manager.RemoveTrain(projectDataGrid.SelectedIndex);

}

}

}

<Window x:Class="Trains.WPF.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

mc:Ignorable="d"

Title="MainWindow" Height="450.477" Width="862.023"

>

<DockPanel >

<DockPanel LastChildFill="False" DockPanel.Dock="Left" >

<Button x:Name="button" HorizontalAlignment="Left" VerticalAlignment="Top" Width="90" Click="button\_Click" DockPanel.Dock="Top" Margin="10">

<WrapPanel Orientation="Vertical">

<ContentControl Template="{StaticResource TrainIcon}" />

</WrapPanel>

</Button>

<Button x:Name="button1" HorizontalAlignment="Left" VerticalAlignment="Top" Width="90" Click="button\_Click" DockPanel.Dock="Top" Margin="10">

<WrapPanel Orientation="Vertical">

<ContentControl Template="{StaticResource StatisticsIcon}" />

</WrapPanel>

</Button>

<Button x:Name="searchButton" HorizontalAlignment="Left" VerticalAlignment="Top" Width="90" Click="button\_Click" DockPanel.Dock="Top" Margin="10">

<WrapPanel Orientation="Vertical">

<ContentControl Template="{StaticResource SearchIcon}" />

</WrapPanel>

</Button>

</DockPanel>

<TabControl DockPanel.Dock="Top" BorderThickness="0" >

<TabItem x:Name="trainstab" Header="Потяги" >

<DockPanel >

<DockPanel DockPanel.Dock="Top" Margin="0,0,0,0" FlowDirection="RightToLeft" >

<Button x:Name="delete" HorizontalAlignment="Left" Width="30" Click="button\_Delete" DockPanel.Dock="Left" Margin="10">

<WrapPanel Orientation="Vertical">

<ContentControl Template="{StaticResource DeleteIcon}" HorizontalContentAlignment="Right" />

</WrapPanel>

</Button>

<Button x:Name="add" HorizontalAlignment="Left" Width="30" Click="button\_Click" DockPanel.Dock="Left" Margin="10">

<WrapPanel Orientation="Vertical">

<ContentControl Template="{StaticResource AddIcon}" HorizontalContentAlignment="Right" />

</WrapPanel>

</Button>

</DockPanel >

<DataGrid x:Name="projectDataGrid" DockPanel.Dock="Top" />

</DockPanel>

</TabItem>

<TabItem Header="Вагони" IsSelected="True" >

<DockPanel >

<DockPanel DockPanel.Dock="Top" Margin="0,0,0,0" FlowDirection="RightToLeft" >

<Button x:Name="delete\_car" HorizontalAlignment="Left" Width="30" Click="button\_Delete" DockPanel.Dock="Left" Margin="10">

<WrapPanel Orientation="Vertical">

<ContentControl Template="{StaticResource DeleteIcon}" HorizontalContentAlignment="Right" />

</WrapPanel>

</Button>

<Button x:Name="add\_train" HorizontalAlignment="Left" Width="30" Click="button\_Click" DockPanel.Dock="Left" Margin="10">

<WrapPanel Orientation="Vertical">

<ContentControl Template="{StaticResource AddIcon}" HorizontalContentAlignment="Right" />

</WrapPanel>

</Button>

</DockPanel >

<DataGrid x:Name="carDataGrid" DockPanel.Dock="Top" />

</DockPanel>

</TabItem>

<TabItem Header="Пошук" IsSelected="True" >

<DockPanel >

<DockPanel DockPanel.Dock="Top" Margin="0,0,0,0" FlowDirection="RightToLeft" LastChildFill="True">

<Button x:Name="search" HorizontalAlignment="Left" Width="30" Click="button\_Click" DockPanel.Dock="Left" Margin="10">

<WrapPanel Orientation="Vertical">

<ContentControl Template="{StaticResource SearchIcon}" HorizontalContentAlignment="Right" />

</WrapPanel>

</Button>

<TextBox Text="Київ" x:Name="DepartureLocation" TextAlignment="Right" FontSize="25" DockPanel.Dock="Right" Width="500" Margin="10"/>

</DockPanel >

<DataGrid x:Name="searchdata" DockPanel.Dock="Top" />

</DockPanel>

</TabItem>

</TabControl>

</DockPanel>

</Window>