ВІДОКРЕМЛЕНИЙ СТРУКТУРНИЙ ПІДРОЗДІЛ

«ТЕРНОПІЛЬСЬКИЙ ФАХОВИЙ КОЛЕДЖ

ТЕРНОПІЛЬСЬКОГО НАЦІОНАЛЬНОГО ТЕХНІЧНОГО УНІВЕРСИТЕТУ ІМЕНІ ІВАНА ПУЛЮЯ»

ВІДДІЛЕННЯ ТЕЛЕКОМУНІКАЦІЙ ТА ЕЛЕКТРОННИХ СИСТЕМ

Циклова комісія комп’ютерних наук

КУРСОВА РОБОТА

з дисципліни:

**«Об’єктно-орієнтоване програмування»**

на тему: **«****Автоматизація процесу управління замовленнями в сфері створення та реалізації** **індивідуального інтер'єру та екстер'єру»**

Студента  3  курсу групи КН-321 спеціальності 122 «Комп’ютерні науки»

Бідловський І. Ю.

(прізвище та ініціали)

Керівник: викладач   Слободян Р. О.         
Національна шкала \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кількість балів: \_\_\_\_\_\_ Оцінка: ECTS \_\_\_\_\_\_

Члени комісії: \_\_\_\_\_\_\_\_\_ Руслан СЛОБОДЯН

(підпис)

\_\_\_\_\_\_\_\_ Роман ДІЛАЙ

(підпис)

м. Тернопіль – 2024

ЗМІСТ

[Вступ 4](#_Toc183281614)

[1 Технічне завдання 5](#_Toc183281615)

[1.1 Найменування та область застосування 5](#_Toc183281616)

[1.2 Підстави для розробки 5](#_Toc183281617)

[1.3 Призначення розробки 5](#_Toc183281618)

[1.4 Вимоги до програми чи програмного виробу 6](#_Toc183281619)

[1.5 Вимоги до програмної документації 7](#_Toc183281620)

[1.6 Техніко-економічні показники 7](#_Toc183281621)

[1.7 Стадії та етапи розробки 8](#_Toc183281622)

[1.8 Порядок контролю та прийому 9](#_Toc183281623)

[2 Розробка технічного та робочого проєкту 10](#_Toc183281624)

[2.1 Розробка загальної структури і варіантів використання програми 10](#_Toc183281625)

[2.2 Розробка системи класів 13](#_Toc183281626)

[2.3 Розробка методів 14](#_Toc183281627)

[2.4 Проєктування і опис інтерфейсу користувача 15](#_Toc183281628)

[2.5 Опис файлової структури програми 19](#_Toc183281629)

[2.6 Опис структури бази даних програми 20](#_Toc183281630)

[3 Тестування програми і результати її виконання 22](#_Toc183281631)

[Висновки 28](#_Toc183281632)

[Перелік посилань 29](#_Toc183281633)

[Додаток А Лістинг файлу «main.cpp» 30](#_Toc183281634)

[Додаток Б Лістинг файлу «mainwindow.h» 31](#_Toc183281635)

[Додаток В Лістинг файлу «mainwindow.cpp» 34](#_Toc183281636)

[Додаток Г Лістинг файлу «databasemanager.h» 51](#_Toc183281637)

[Додаток Ґ Лістинг файлу «databasemanager.cpp» 53](#_Toc183281638)

Вступ

Сучасний світ програмування вимагає нового підходу до розробки програмного забезпечення. Розробникам потрібне рішення, яке дозволить їм створювати додатки швидше, простіше та з меншими витратами, забезпечуючи при цьому швидкість, надійність, безпеку та простоту використання. Особливо важливо враховувати психологічні та фізіологічні особливості сприйняття інформації людьми для створення програмних продуктів, зручних для тривалого використання і спрямованих на задоволення потреб користувачів. Це ставить перед розробниками складне завдання, що вимагає не тільки глибоких технічних знань, а й розуміння принципів людського сприйняття і взаємодії з інформацією.  
 Зростання приватного сектору економіки спонукало до оптимізації витрат на комп’ютерну техніку та спеціалізоване програмне забезпечення. Сьогодні вигідніше створювати компактні програмні рішення для вузькоспеціалізованих завдань. У цьому контексті перед навчальними закладами стоїть завдання підготувати фахівців, здатних створювати ефективні, гнучкі програмні засоби з мінімальними витратами часу та ресурсів.

У даній курсовій роботі представлено розробку спеціалізованого програмного забезпечення **«****Tailored Design Solutions»**, яке реалізовано мовою програмування С++ із використанням принципів об’єктно-орієнтованого програмування (ООП). Метою розробки є автоматизація процесу комунікації між клієнтом – дизайнером та бригадою майстрів.

Об’єктно-орієнтоване програмування як основа цієї розробки є однією з провідних парадигм програмування. Воно ґрунтується на уявленні програми у вигляді сукупності взаємодіючих об'єктів, кожен із яких є екземпляром певного класу. Класи можуть формувати ієрархії через механізм наслідування, що дає змогу розширювати базові класи й створювати більш функціональні програмні засоби. Завдяки ООП структура програми стає зрозумілою, а розробка програмного забезпечення — ефективнішою.

Розроблений програмний засіб **«****Tailored Design Solutions»** демонструє практичне застосування принципів ООП для створення зручного та економічно вигідного рішення, орієнтованого на потреби сучасних замовників.

# Технічне завдання

## Найменування та область застосування

Найменування програми – Програма «Управління замовленнями у сфері створення та реалізації кастомного інтер'єру та екстер'єру».

Стисла назва – «Tailored Design Solutions».  
 Область застосування програми – Це програмне забезпечення створюється для спрощення процесу замовлення та реалізації проєктів, дизайнерських ідей у сфері індивідуального інтер'єру та екстер'єру. Підходить як для невеликих підприємців, так і для великих компаній у даній сфері діяльності. Допомагає спростити процес комунікації між клієнтом – дизайнером та бригадою майстрів.

## Підстави для розробки

Підставами для проведення розробки являється індивідуальне завдання на курсову роботу з дисципліни «Об’єктно-орієнтоване програмування».

Найменування теми курсової роботи – «Автоматизація процесу управління замовленнями в сфері створення та реалізації індивідуального інтер'єру та екстер'єру».

Виконавець ­– студент групи КН-321 Бідловський Іван Юрійович.

## Призначення розробки

Програма призначена для підвищення ефективності роботи і взаємодії клієнтів, дизайнерів і будівельними бригадами, щоб забезпечити оптимізацію процесу обміну інформацією, координації рішень і контролю за виконанням робіт. «Tailored Design Solutions».  
 Функціональне призначення програми – оптимізація та спрощення обліку замовлень, процесу обміну інформації, автоматизації підбору будівельних інструментів та матеріалів.

### Вимоги до функціональних характеристик

Програма для ведення обліку пасіки повинна забезпечувати виконання та оптимізацію наступних функцій:

* Облік даних – програма повинна зберігати та відображати інформацію щодо продуктів пасіки, збору врожаю та характеристик вуликів.
* Обробка помилок – програма повинна перевіряти коректність введених даних і надавати користувачеві повідомлення про помилки або некоректні дії.

### Вимоги до часових характеристик

Програмне забезпечення повинно швидко реагувати на запити користувачів, забезпечуючи доступ до необхідних даних про продукти, вулики чи зібраний урожай.

### Вимоги до надійності

Для забезпечення надійності і безпеки програми необхідно впровадити механізми, що перевіряють правильність даних, що вводяться. Це запобігає помилкам, які можуть виникнути при введенні інформації користувачами, і забезпечує цілісність даних у базі даних.

### Умови до експлуатації

Розроблений програмний продукт буде простий у використанні завдяки простому і інтуїтивно зрозумілому інтерфейсу. Для запуску програми користувачу досить базових навичок з інформатики.

### Вимоги до складу і параметрів технічних засобів

До складу технічних засобів повинно входити: монітор, клавіатура, мишка, IBM-сумісний персональний комп’ютер із такими мінімальними характеристиками:

* Процесор – AMD Ryzen 3;
* оперативна пам’ять – 4 ГБ;
* обсяг дискової пам’яті – 10 ГБ.

### Вимоги до інформаційної і програмної сумісності

Вихідний код програми повинен бути реалізований мовою С++. Як середовище для розробки програмного забезпечення використовується фреймворк Qt Creator.

Вимоги до захисту інформації і програми не пред’являються.

## Вимоги до програмної документації

Після завершення розробки програмного забезпечення потрібно підготувати наступну документацію:

* інструкція з інсталяції програми;
* загальні відомості про функціональні можливості програми;
* інструкція з використання та експлуатації програми.

## Техніко-економічні показники

Розрахунок економічної ефективності та вартості розробки програмного продукту не проводиться.

Передбачуване використовування розробленої програми в рік залежить від обсягу замовлень.

## Стадії та етапи розробки

Розробка програми для ведення обліку пасіки буде складатися з таких етапів:

* Аналіз.

Етап аналізу включає детальне вивчення предметної області, пов’язаної з обліком сфери створення індивідуального, дизайнерського ремонту. Метою аналізу є визначення функціональних і нефункціональних вимог до програми, аналіз взаємозв’язків між об’єктами предметної області (замовники, дизайнери, працівники). Під час аналізу створюється схема зв'язків між об'єктами для виводу логікі їх взаємодії.

* Проектування.

На етапі проектування розробляється архітектура програми, яка включає структурну схему роботи системи, опис бази даних, зокрема таблиць для зберігання даних про клієнтів, замовлення, дизайнерів, інструментів, будівельного плану, кошторису та робітників. Крім того, визначається взаємодія між компонентами програми і створюється дизайн користувальницького інтерфейсу, який орієнтований на зручність і простоту використання. Проектування передбачає створення гнучкої та ефективної системи з використанням об'єктно-орієнтованих принцип, таких як інкапсуляція, успадкування та композиція. Результатом проектування є ієрархії класів, схеми баз даних та прототипи інтерфейсу користувача.

* **Тестування.**

Етап тестування включає перевірку функціональності програми в різних варіантах використання. Також присутня функціональна перевірка (робота бази даних, правильний облік даних, оновлення репозиторію), перевірки стабільності програми в разі неправильного використання і роботи системи під навантаженням. Метою тестування є не тільки виявлення та усунення помилок, а й забезпечення стабільної та надійної роботи програми перед її запуском.

## Порядок контролю та прийому

Розроблений проект повинен прийматися Замовником відповідно до заданих термінів, які були визначені в індивідуальному завданні.

Для прийняття роботи Виконавець має надати:

* повністю працездатну програму, яка відповідає вимогам технічного завдання;
* вихідний код програми, записаний разом із програмним продуктом на оптичний або інший погоджений носій інформації.

Прийом програмного забезпечення повинен відбуватися перед комісією з двох чоловік (один з яких – Замовник) у такій послідовності:

* доповідь Виконавця про виконану роботу;
* демонстрація Виконавцем роботи програми;
* контрольні випробовування роботи програми;
* відповіді на запитання і зауваження комісії.

# Розробка технічного та робочого проєкту

## Розробка загальної структури і варіантів використання програми

В даному підрозділі пояснювальної записки описується такий етап розробки програми, як аналіз. Метою аналізу є максимально повний опис поставленої задачі, відповідно до технічного завдання.

Під час проходження етапу аналізу спочатку було визначено такі питання:

* Хто буде діючими суб'єктами?
* Хто вводитиме інформацію?
* Хто запрошуватиме?
* Чи буде хто-небудь ще взаємодіяти з програмою?
* Чи буде сама програма взаємодіяти з іншими програмами?

В даному випадку в програмі Tailored Design Solutions відбувається взаємодія між трьома користувачами. Є три типи користувачів: Замовник (Customer), Дизайнер (Inspector) та Робітник (Worker).  
 Наступне, що потрібно було з'ясувати, це групу дій, які ініціюватиме діючі суб'єкти. Виходячи із технічного завдання будуть виконуватись такі дії:  
 Для замовника:

− почати роботу з програмою;

− створення замовлення з коротким описом ідей та побажань, вказуючи адресу та контактні дані;

− підтвердження замовлення;

− очікування;

Для дизайнера:

− почати роботу з програмою;

− перегляд замовлення;

− підтвердження роботи над замовленням;

−внесення у програму таких даних як: всі необхідні заміри об’єкту, математичні розрахунки, медіадані, об’єм будівельних матеріалів, кошторис та будівельний план;  
 − підтвердження нових даних для робітника;

− ревізія роботи, оцінка виконаних робіт;

− фінальне підтверждення;  
 Для працівника:

− почати роботу з програмою;

− перегляд замовлення від дизайнера;  
 −перегляд списку рекомендованих інструментів та будівельних матеріалів;

− підтвердження роботи над замовленням;  
 − підтвердження виконання роботи у вигляді завантаження медіаданих виконаної роботи;  
 Таким чином, діаграму варіантів використання, яка отримується внаслідок наведеного переліку дій, можна подати так, як це показано на рисунку 2.1.

Наступний крок етапу аналізу це опис варіантів використання, де необхідно детально описати всі варіанти використання.

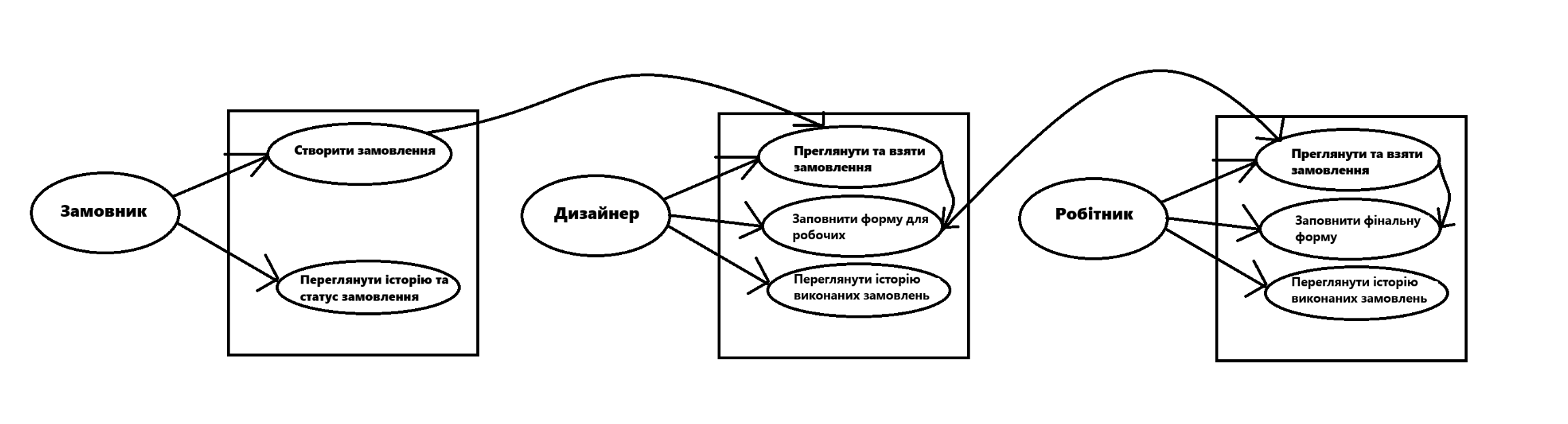


Рисунок 2.1 – Діаграма варіантів використання програми Tailored Design Solutions

Детальний опис основних варіантів використання:

1. Запустити програму.

При запуску програми повинен відображатися головний екран інтерфейсу користувача, який містить три елементи Push Button, які в свою чергу виконують такі дії як Вхід для користувача який зареєструвався, Реєстрація, та Вихід з програми, відповідно.

1. Реєстрація.

Натиснувши кнопку “Реєстрація”, користувачу відкриється діалогове вікно реєстрації, де присутня форма в яку потрібно ввести свої дані, а також вибрати тип користувача (Працівник, Дизайнер, Замовник). Щоб закінчити реєстрацію потрібно натиснути кнопку підтвердження.

1. Вхід.

Натиснувши кнопку “Вхід”, користувачу відкриється діалогове вікно авторизації, де присутня форма в яку потрібно ввести свої дані які ви вказували при реєстрації. Щоб закінчити авторизацію потрібно натиснути кнопку підтвердження.

4) Вихід**.**

Натиснувши кнопку “Вихід”, програма закривається.  
Після успішної авторизації, користувачу відкриється діалогове вікно, в залежності від його типу (Працівник, Дизайнер, Замовник). Спочатку опишемо використання програми від лиця замовника.  
1) Відкриття вікна.  
При відкритті вікна, повинен відображатися головний екран інтерфейсу користувача який містить Tab Widget у якому є дві вкладки Create Order та Order History.  
2) Відкриття вкладки Create Order.  
На вкладці користувача зустрічає форма для створення замовлення, щоб підтвердити форму, потрібно натиснути кнопку. Замовлення збережеться в базі даних.  
3) Відкриття вкладки Order History.  
На вкладці користувача зустрічає List Widget, який містить в собі інформацію отриману з бази даних про всі замовлення користувача, а також присутня кнопка. Щоб оновити історію замовлень потрібно натиснути кнопку.  
  
Тепер розглянемо сценарій подій від лиця дизайнера.  
1) Відкриття вікна.  
При відкритті вікна, повинен відображатися головний екран інтерфейсу користувача, який містить Tab Widget у якому є три вкладки Orders, Work Details та Done Orders.  
2) Відкриття вкладки Orders.  
На вкладці користувача зустрічає List Widget, який містить в собі інформацію отриману з бази даних про замовлення користувача в статусі ‘В очікувані’, а також присутні кнопки для оновлення списку та підтвердження роботи над замовленням відповідно. Після натискання кнопки ‘Взяти замовлення’, статус зміниться з ‘В очікуванні’ на ‘В процесі’.  
3) Відкриття вкладки Work Details.   
На вкладці користувача зустрічає List Widget, який містить в собі інформацію отриману з бази даних про замовлення користувача в статусі ‘В процесі’, форма для внесення даних, а також присутні кнопки для оновлення списку та збереження всіх даних для робітників відповідно. Після натискання кнопки ‘Зберегти деталі’, статус зміниться з ‘В процесі’ на ‘Надіслано робітникам’, а також форма збережеться в базі даних.  
4) Відкриття вкладки Done Orders.  
На вкладці користувача зустрічає List Widget, який містить в собі інформацію отриману з бази даних про замовлення користувача в статусі ‘Відправлено робітникам’ та ‘Завершено’, кнопка для оновлення списку, а також присутня кнопка для пошуку та рядок Line Edit для введення по якому критерію буде здійснений пошук.  
  
Тепер розглянемо сценарій подій від лиця робітника.  
1) Відкриття вікна.  
При відкритті вікна, повинен відображатися головний екран інтерфейсу користувача, який містить Tab Widget у якому є три вкладки Orders, Work Details та Done Orders.  
2) Відкриття вкладки Orders.  
На вкладці користувача зустрічає List Widget, який містить в собі інформацію отриману з бази даних про замовлення дизайнер в статусі ‘Надіслано працівникам’, а також присутні кнопки для оновлення списку та підтвердження роботи над замовленням відповідно. Після натискання кнопки ‘Взяти замовлення’, статус зміниться з ‘Надіслано робітникам’ на ‘Прийнято робітниками’.  
3) Відкриття вкладки Work Details.   
На вкладці користувача зустрічає List Widget, який містить в собі інформацію отриману з бази даних про замовлення користувача в статусі ‘Прийнято робітниками’, форма для внесення даних, а також присутні кнопки для оновлення списку та збереження фінальних даних. Після натискання кнопки ‘Зберегти деталі’, статус зміниться з ‘Прийнято робітниками’ на ‘Завершено’, а також форма збережеться в базі даних.  
4) Відкриття вкладки Done Orders.  
На вкладці користувача зустрічає List Widget, який містить в собі інформацію отриману з бази даних про замовлення користувача в статусі ‘Завершено’, кнопка для оновлення списку, а також присутня кнопка для пошуку та рядок Line Edit для введення по якому критерію буде здійснений пошук.

Розробка системи класів

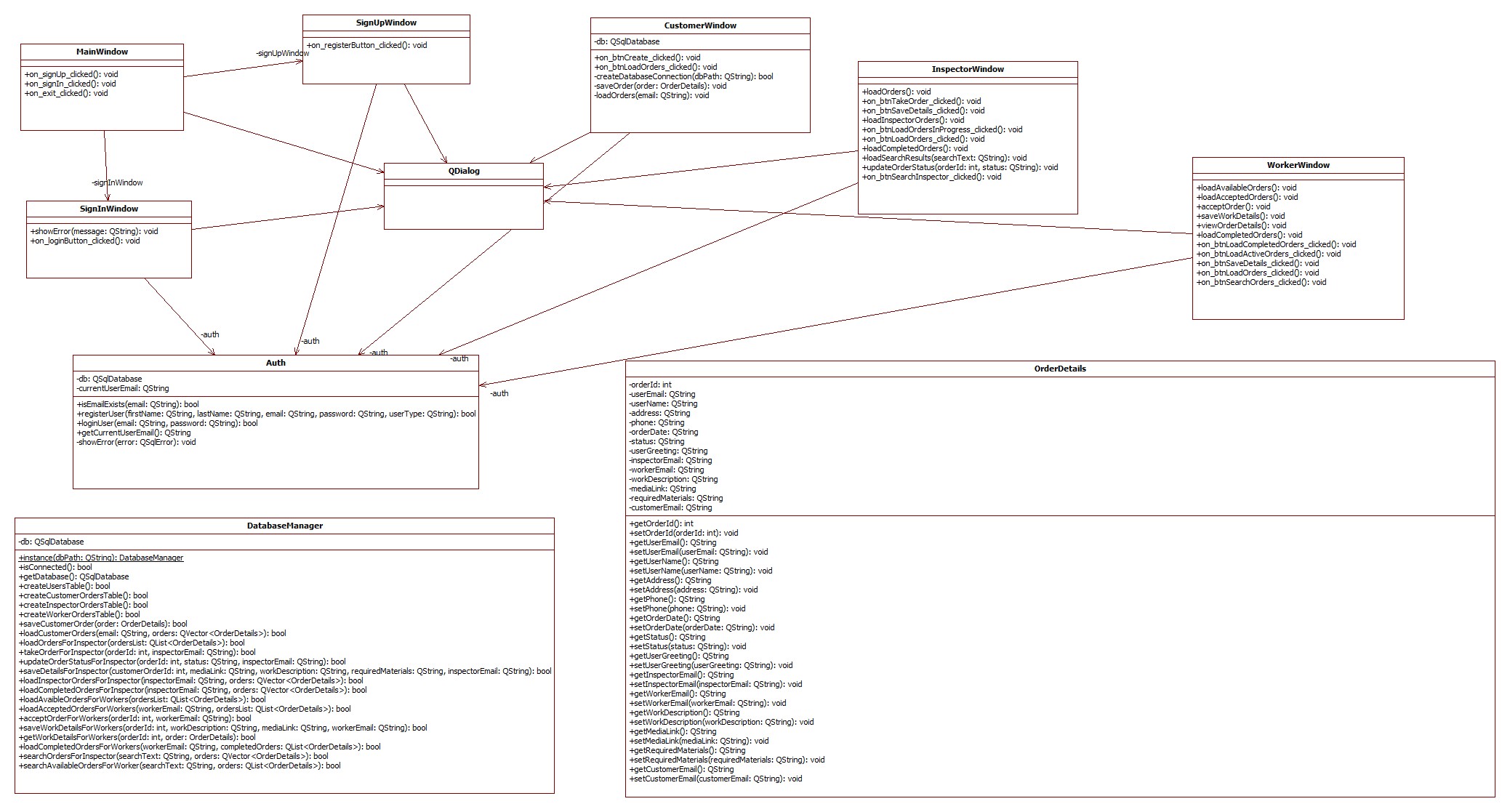
На етапі проектування одним із ключових аспектів є розробка структури класів. Для цього спочатку необхідно ідентифікувати основні класи, які будуть складати програму. Оскільки головними об'єктами в розробленій програмі є замовлення, клієнти, робітники, дизайнери та записи стану замовлень, ці об'єкти доцільно розглядати як окремі класи, кожен із яких матиме відповідні атрибути та методи для управління їх станом та взаємодією.   
  


Рисунок 2.2 **–** Діаграма зв’язків між класами

Оскільки усі ключові дані для керування цими суб’єктами вводяться через окремі вікна, доцільно також розглянути ці вікна як класи. Крім того, для забезпечення зручної роботи з базою даних необхідно створити клас, який реалізує стандартні методи для взаємодії з базою даних, такі як створення таблиць.

## Розробка методів

Наступним кроком є розробка методів, які включають створення алгоритмів для роботи з даними програми та їх реалізацію. У фреймворку Qt для цього використовуються сигнали та слоти, які організовують взаємодію між класами: один об'єкт генерує сигнал, а метод іншого об'єкта реагує на нього.

Кожен тип сутності (дизайнер, замовник, робітник) має своє окреме діалогове вікно, яке дозволяє користувачеві виконувати операції додавання, редагування та видалення даних. Для цього створені класи як СustomerWindow, WorkerWindow, InspectorWindow, які забезпечують взаємодію з відповідними об’єктами.

У кожному з цих класів реалізовані методи для додавання та редагування даних. Наприклад:

InspectorWindow. Цей клас реалізує інтерфейс для інспектора, який відповідає за моніторинг і контроль виконання замовлень.   
 Метод InspectorWindow::loadOrders() завантажує всі замовлення з бази даних зі статусом "Pending" (Очікує) і виводить їх у вигляді списку у вікні. Він виконує SQL-запит для вибірки всіх замовлень з бази даних, що мають статус "Pending", і потім виводить інформацію про замовлення в список.   
 Метод InspectorWindow::on\_btnTakeOrder\_clicked() викликається при натисканні на кнопку ‘Прийняти замовлення’. Інспектор може прийняти замовлення і змінити його статус на "In Progress" (В процесі). Цей метод перевіряє, чи обране замовлення в списку, потім змінює його статус на "In Progress" у базі даних і записує, який інспектор прийняв замовлення.   
 Метод InspectorWindow::updateOrderStatus(int orderId, const QString &status) оновлює статус замовлення в базі даних, виконуючи SQL-запит для оновлення статусу замовлення на новий і зберігає імейл інспектора.   
 Метод InspectorWindow::on\_btnSaveDetails\_clicked() викликається при натисканні на кнопку ‘Зберегти деталі’. Він зберігає додаткові деталі виконаних робіт для замовлення та змінює його статус на "Sent to workers" (Відправлено до робітників). Цей метод оновлює замовлення з новим статусом і додає запис в таблицю inspectororders, що містить опис робіт, матеріали та медіа-лінки.

Метод InspectorWindow::loadInspectorOrders() завантажує всі замовлення, які знаходяться в процесі виконання, та відображає їх у списку. Він вибирає всі замовлення зі статусом "In Progress", що були прийняті поточним інспектором, і виводить їх у список з описом виконаних робіт і медіа-лінками.

Метод InspectorWindow::loadCompletedOrders() завантажує замовлення, які були завершені або відправлені до робітників, і відображає їх у списку. Цей метод виконує SQL-запит для вибірки замовлень зі статусами "Done" або "Sent to workers", і виводить деталі замовлень, включаючи медіа-лінки, матеріали та іншу інформацію.  
 Метод InspectorWindow::on\_btnSearch\_clicked() реалізує пошук замовлень. Користувач може ввести текст пошуку, і програма перевірить базу даних на наявність замовлень, що відповідають цьому запиту. Пошук може проводитись за такими критеріями, як ідентифікатор замовлення, імейл користувача, адреса, статус, дата замовлення тощо.

Метод InspectorWindow::loadSearchResults(const QString &searchText) виконує пошуковий запит до бази даних за введеним текстом. Якщо знайдено відповідні замовлення, вони відображаються в списку у вікні інтерфейсу користувача. У разі відсутності результатів виводиться повідомлення про відсутність замовлень, що відповідають критеріям.

## Проєктування і опис інтерфейсу користувача

Інтерфейс користувача (ІК) - це сукупність засобів, за допомогою яких користувач взаємодіє з різними пристроями (з комп'ютером або побутовою технікою) або іншим складним інструментарієм (системою). Інтерфейс користувача - це такий різновид інтерфейсів, в якому з одного боку - людина, з іншого - машина (пристрій, програмне забезпечення). За визначенням Національного банку стандартизованих науково-технічних термінів, інтерфейс користувача - це комплекс апаратних і програмних засобів, що забезпечує взаємодію користувача з комп'ютером.

ІК часто розуміють лише як зовнішній вигляд програмного забезпечення (П3), але таке розуміння є надто вузьким, оскільки саме за допомогою інтерфейсу користувач сприймає програму в цілому та використовує її функціональність. ІК забезпечує підтримку прийняття рішень у визначеній предметній галузі та визначає порядок використання ПЗ і документації до нього.

В дійсності, ІК об'єднує усі елементи і компоненти ПЗ, які здатні впливати на взаємодію користувача з програмним забезпеченням. До таких елементів належать: набір задач, які користувач розв'язує за допомогою ПЗ; використовувана програмним забезпеченням метафора (наприклад, "робочий стіл" у операційній системі Windows); елементи управління ПЗ; навігація між блоками ПЗ; візуальний (і не тільки) дизайн вікон та екранних форм програми та інші складові.

При створенні ПЗ використовуватиметься середовище Qt Designer. Основні класи, які використовуватимуться для роботи з вікнами: QMainWindow, QWidget, QMessagebox, QSqTableModel.

Також при розробці та оформленні вікон будуть використовуватись такі

елементи фреймворку Qt:

* QLineEdit — поле для вводу;
* QDateTimeEdit — поле для введення тексту;
* QListWidget — інструмент для виводу таблиці на екран;
* QPushButon — кнопки;
* QRadioButton — кнопки для вибору;
* QGгоupВоx — впорядковує елементи по групам;
* QTabWidget — віджет для перемикання між кількома екранами.

Переглянути як ці елементи працюють можна B З розділі даної записки.

При оформлені та створенні вікна “Реєстрація” використовуватись такі

елементи Qt: QLineEdit, QRadioButton, QPushButon, їx розташування зображено на рисунку 2.3.

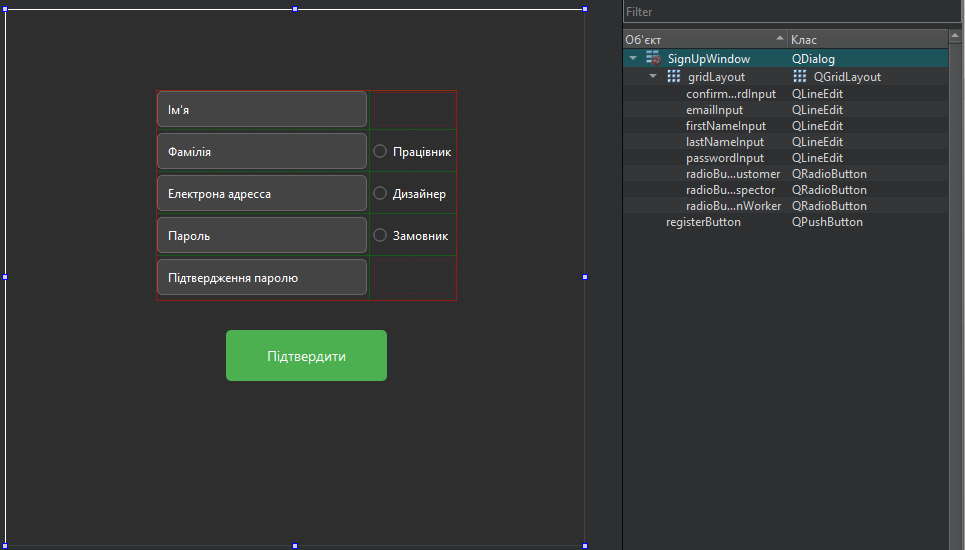


Рисунок 2.3 – Вікно “Реєстрація”

При оформлені та створенні вікна “Увійти” використовуватись такі елементи Qt: QLineEdit, QPushButon, їx розташування зображено на рисунку 2.4.

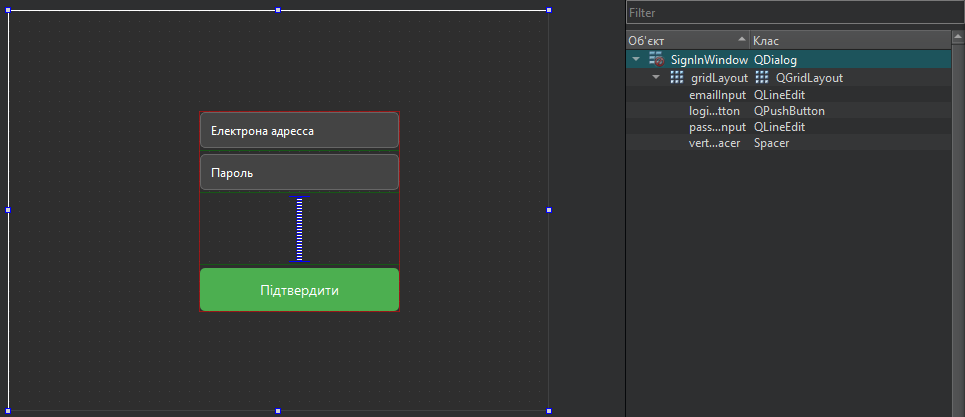


Рисунок 2.4 – Вікно “Увійти”

При оформлені та створенні вкладки “Створити замовлення” у вікні Customer використовуватись такі елементи Qt: QLineEdit, QDateTimeEdit, QPushButon, їx розташування зображено на рисунку 2.5.

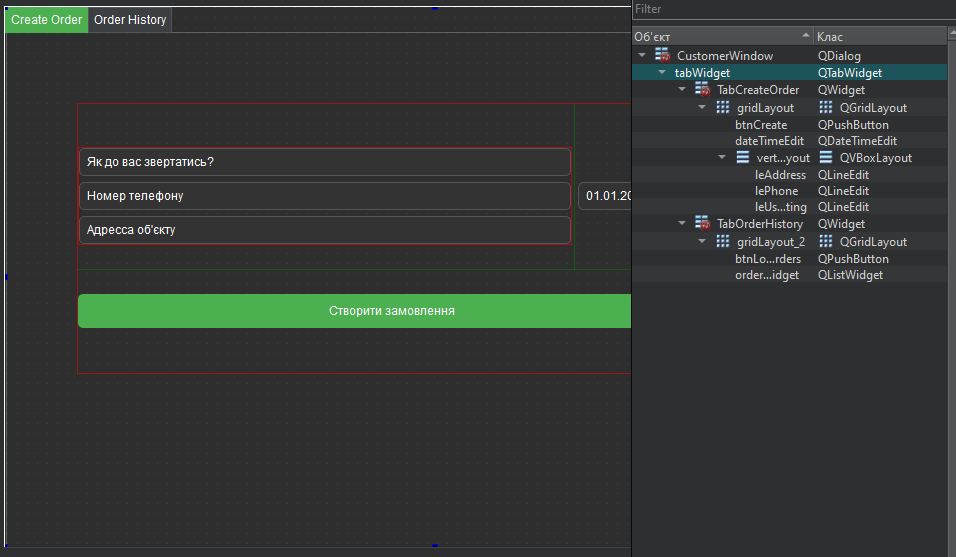


Рисунок 2.5 – Вкладка “Створити замовлення” у вікні Customer.

При оформлені та створенні вкладки “Історія замовлень” у вікні Customer використовуватись такі елементи Qt: QListWidget, QPushButon, їx розташування зображено на рисунку 2.6.

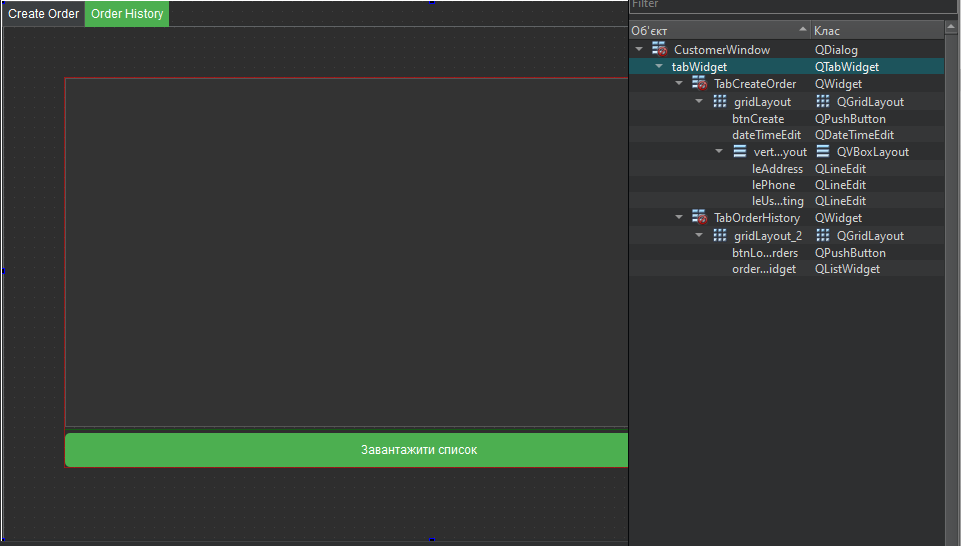


Рисунок 2.6 – Вкладка “Історія замовлень” у вікні Customer.

При оформлені та створенні вкладки “Замовлення” у вікні Inspector використовуватись такі елементи Qt: QListWidget, QPushButon, їx розташування зображено на рисунку 2.7.

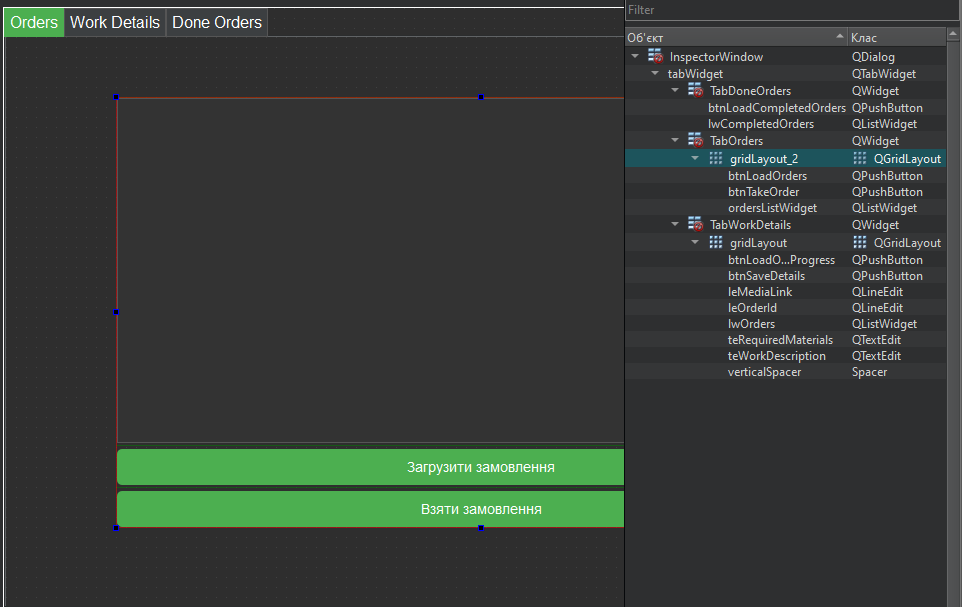


Рисунок 2.7 – Вкладка “Замовлення” у вікні Inspector.

При оформлені та створенні вкладки “Деталі замовлень” у вікні Inspector використовуватись такі елементи Qt: QListWidget, QPushButon, QLineEdit, їx розташування зображено на рисунку 2.8.

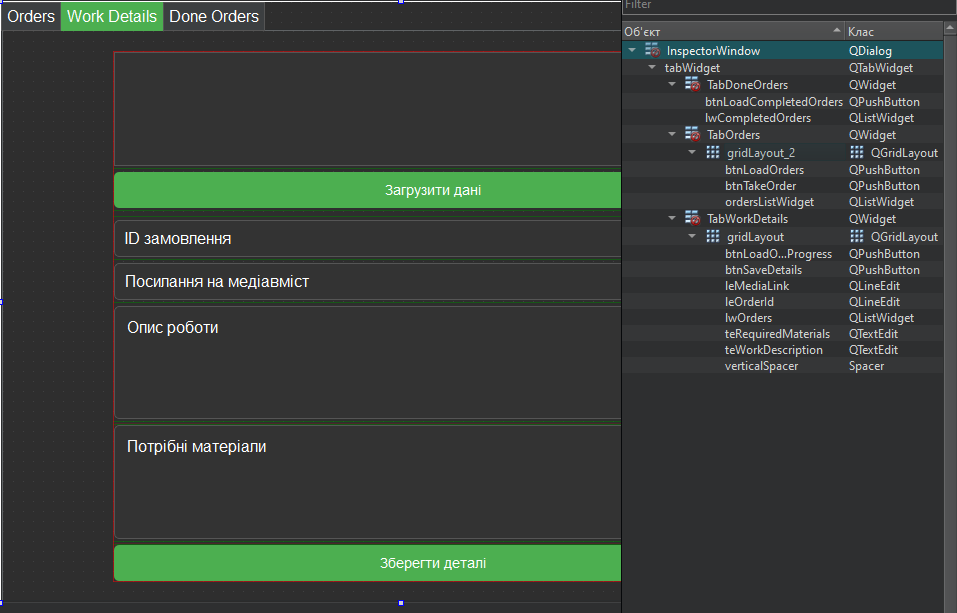


Рисунок 2.8 – Вкладка “Деталі замовлень” у вікні Inspector.  
  
 При оформлені та створенні вкладки “Виконані замовленя” у вікні Inspector використовуватись такі елементи Qt: QListWidget, QPushButon, та QLineEdit, їx розташування зображено на рисунку 2.9.

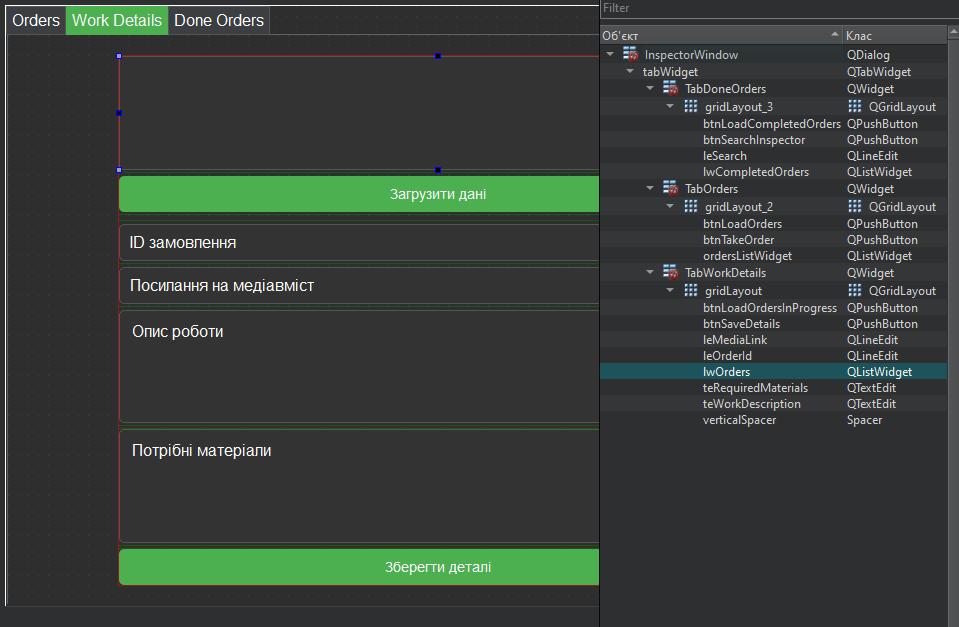


Рисунок 2.9 – Вкладка “Виконані замовлення” у вікні Inspector.

При оформлені та створенні вкладки “Замовлення” у вікні Worker використовуватись такі елементи Qt: QListWidget, QPushButon, їx розташування зображено на рисунку 2.10.

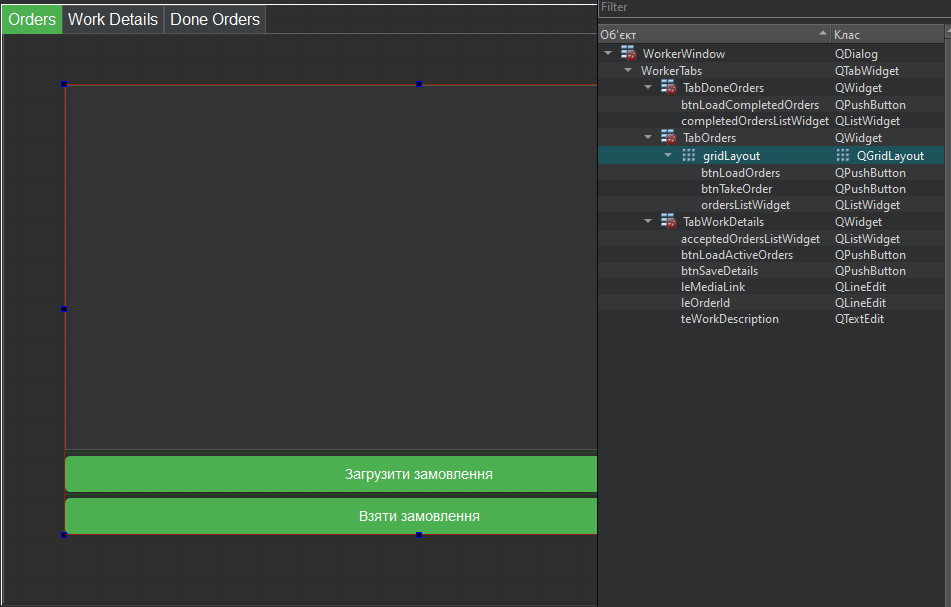


Рисунок 2.10 – Вкладка “Замовлення” у вікні Worker.  
  
 При оформлені та створенні вкладки “Деталі замовлень” у вікні Worker використовуватись такі елементи Qt: QListWidget, QPushButon, QLineEdit, їx розташування зображено на рисунку 2.11.

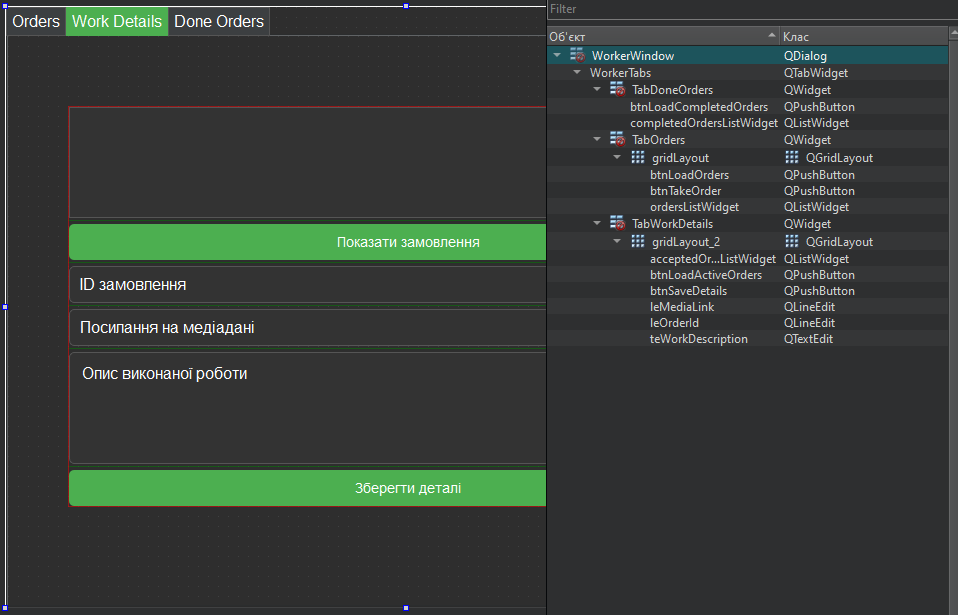


Рисунок 2.11 – Вкладка “Деталі замовлень” у вікні Worker.

При оформлені та створенні вкладки “Виконані замовленя” у вікні Worker використовуватись такі елементи Qt: QListWidget, QPushButon, та QLineEdit, x розташування зображено на рисунку 2.12.

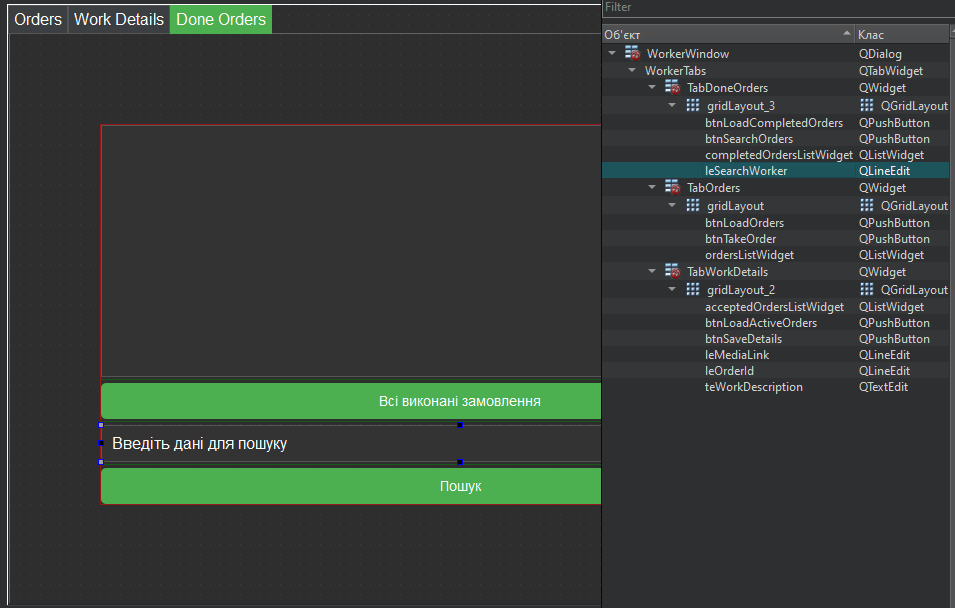


Рисунок 2.12 – Вкладка “Виконані замовлення” у вікні Worker.

## Опис файлової структури програми

Програма складається із таких вихідних файлів:

* «main.cpp» - лістинг цього файла знаходиться в додатку А.   
   В цьому файлі міститься головна функція програми;
* «mainwindow.h» - лістинг цього файла знаходиться в додатку Б;
* «mainwindow.cpp» - лістинг цього файла знаходиться в додатку В;
* «databasemanager.h» - лістинг цього файла знаходиться в додатку Г;
* «databasemanager.cpp» - лістинг цього файла знаходиться в додатку Д;
* «auth.h» – файл заголовку класу Auth;
* «auth.cpp» ­– файл реалізації класу Auth;
* «customerwindow.h» – файл заголовку класу CustomerWindow;
* «customerwindow.cpp» – файл реалізації класу CustomerWindow;
* «inspectorwindow.h» – файл заголовку класу InspectorWindow;
* «inspectorwindow.cpp» – файл реалізації класу InspectorWindow;
* «workerwindow.h» – файл заголовку класу WorkerWindow;
* «workerwindow.cpp» – файл реалізації класу WorkerWindow;
* «signupwindow.h» – файл заголовку класу SignUpWindow;
* «signupwindow.cpp» – файл реалізації класу SignUpWindow;
* «signinwindow.h» – файл заголовку класу SignInWindow;
* «signinwindow.cpp» – файл реалізації класу SignInWindow;
* «Kyrsova.pro» - це проектний файл, який використовується для опису налаштувань і структури проекту.

## Опис структури бази даних програми

База даних програми складається із шести таблиць, які зображено на ER-діаграмі, яка представлена на рисунку 2.13. В якості системи управління бази даних було використано SQLite — полегшена реляційна система керування базами даних.

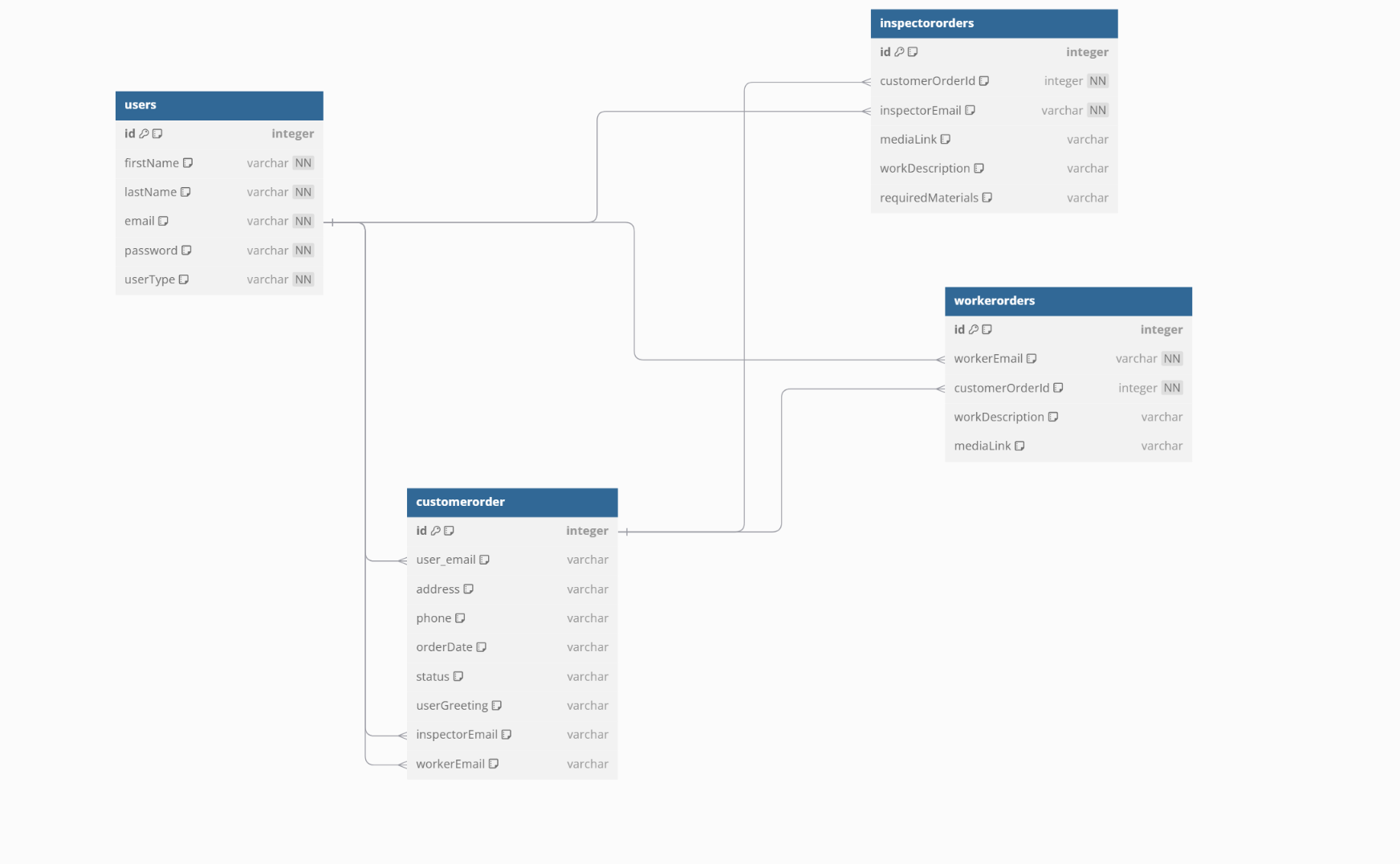


Рисунок 2.13 – ER-діаграма бази даних

# Тестування програми і результати її виконання

В ході тестування програми було виявлено і виправлено незначні помилки.

Запустивши програму на екрані монітора відображається головне вікно, яке представлене на рисунку 3.1.

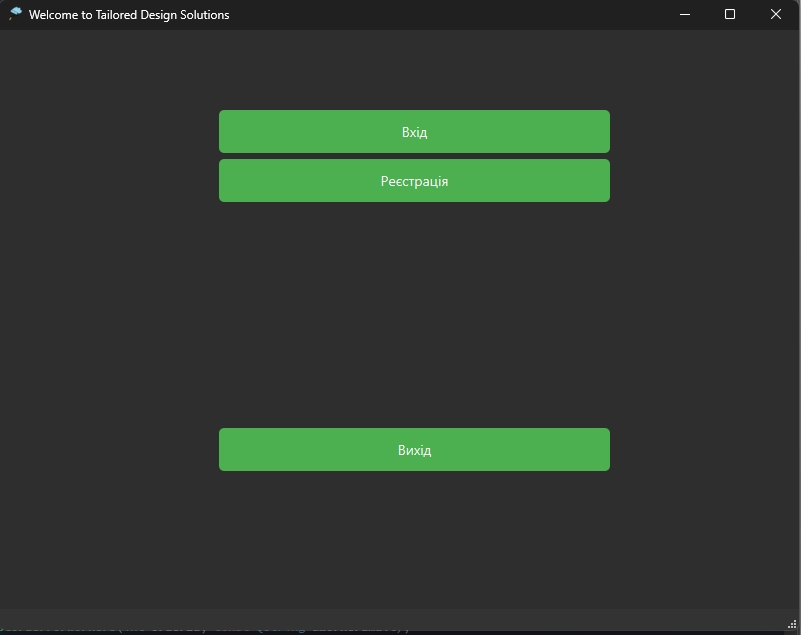


Рисунок 3.1 – Головне вікно програми “Tailored Design Solutions”

На головному вікні програми розташовані 3 головні кнопки.

* Вхід;
* Реєстрація;
* Вихід;

Натиснувши на кнопку «Реєстрація», відкривається вікно, яке зображене на рисунку 3.2.

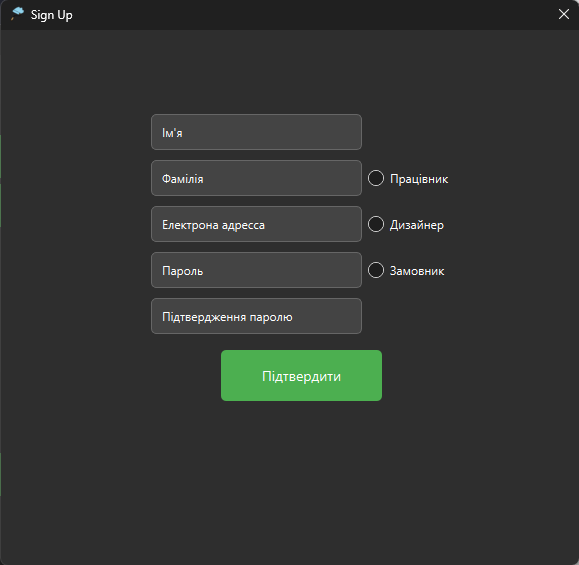


Рисунок 3.2 – Реєстрація

Натиснувши на кнопку «Вхід», відкривається вікно, яке зображене на рисунку 3.3.

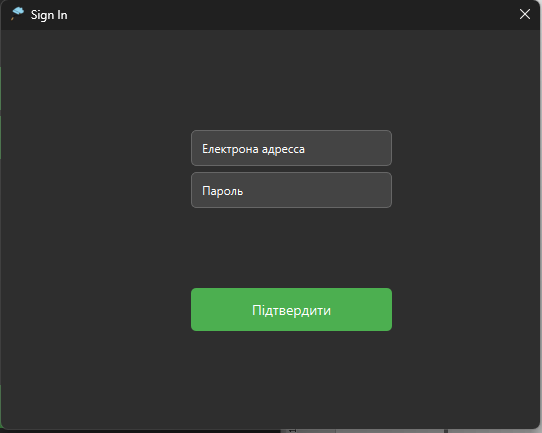


Рисунок 3.3 – Вхід

Натиснувши на кнопку «Підтвердити», відкривається вікно в залежності від вашого типу користувача, яке зображене на рисунку 3.4.

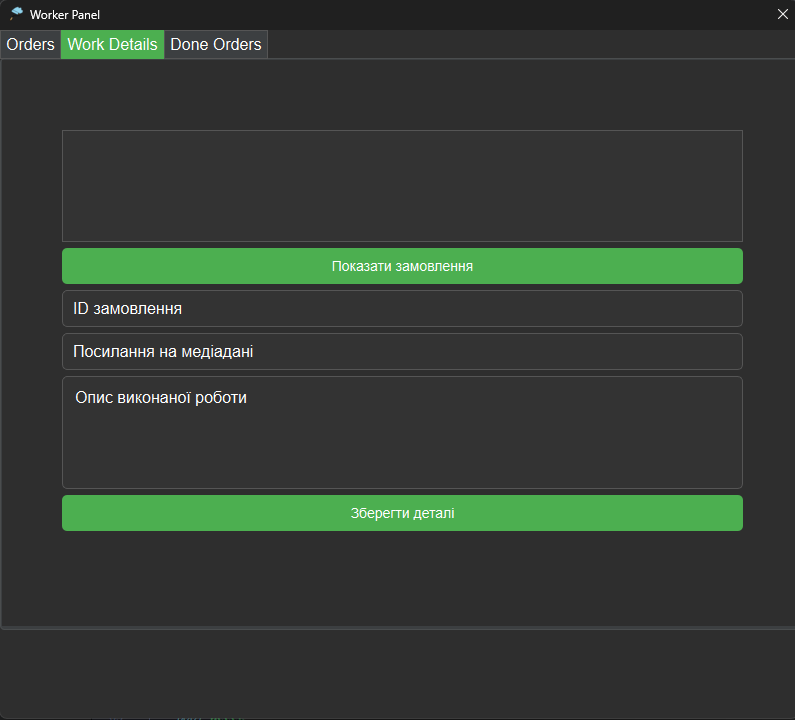


Рисунок 3.4 – Вікно працівника, вкладка Work Details

Натиснувши на кнопку «Показати замовлення», оновиться список замовлень, натиснувши кнопку «Зберегти деталі», форма відправиться в базу даних, натиснувши вкладку на «Orders», TabWidget перемикається на вкладку Замовлення, яке зображене на рисунку 3.5.

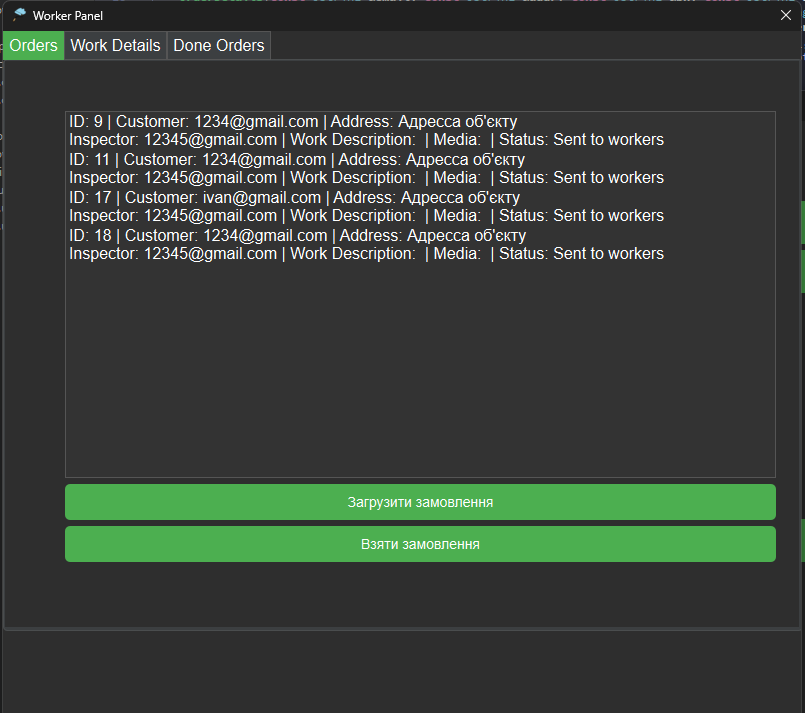


Рисунок 3.5 – Вкладка замовлення

Натиснувши на кнопку «Загрузити замовлення», оновиться список замовлень, натиснувши кнопку «Взяти замовлення», статус замовлення поміняється, натиснувши вкладку на «Done Orders», TabWidget перемикається на вкладку Виконані замовлення, яке зображене на рисунку 3.6.

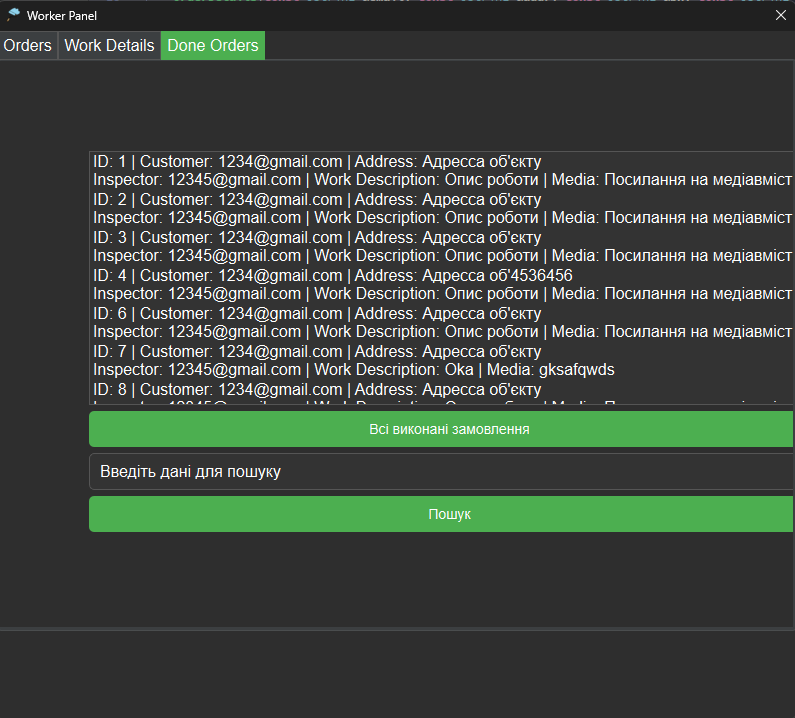


Рисунок 3.6 – Вкладка замовлення

Натиснувши на кнопку «Всі виконані замовлення», оновиться список замовлень які виконав працівник, натиснувши кнопку «Пошук», після того як введете критерій в рядок LineEdit, виконається пошук. Тепер входимо як дизайнер. Нас зустрічає вікно із трьома вкладками, яке зображене на рисунку 3.7.

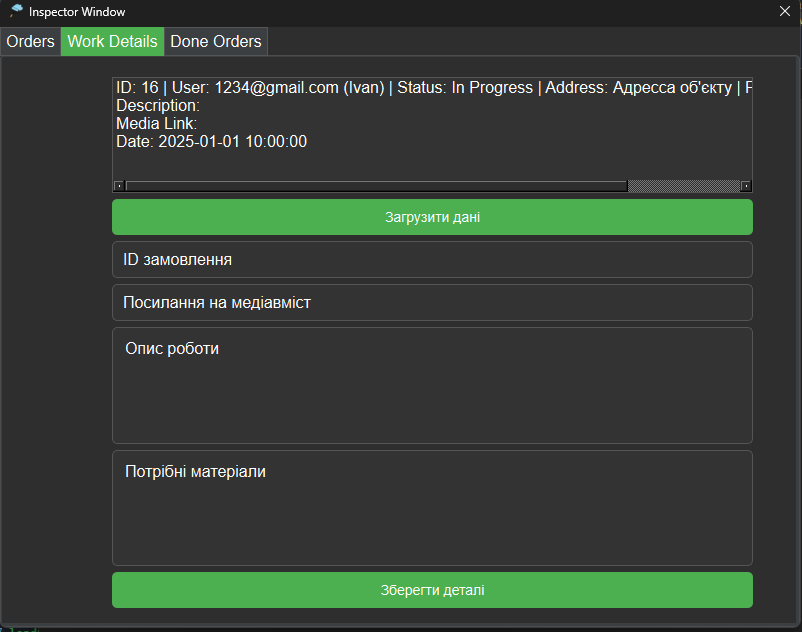


Рисунок 3.7 – Вікно дизайнера, вкладка деталі роботи

Натиснувши на кнопку «Загрузити дані», список із замовленнями обновиться, натиснувши «Зберегти деталі», формам відправиться в базу даних, натиснувши вкладку на «Orders», TabWidget перемикається на вкладку Замовлення, яке зображене на рисунку 3.8.

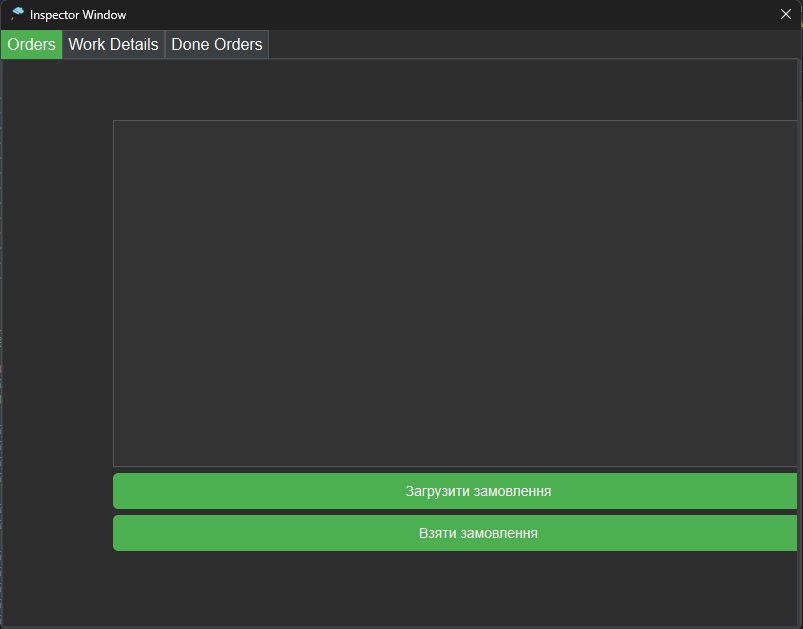


Рисунок 3.8 – Вкладка замовлення

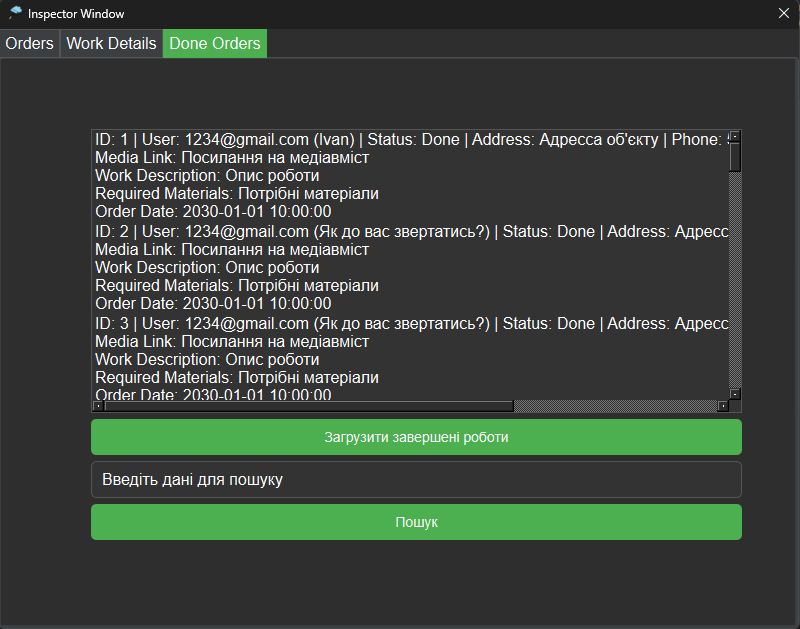
Натиснувши на кнопку «Загрузити замовлення», оновиться список замовлень, натиснувши кнопку «Взяти замовлення», статус замовлення зміниться, натиснувши вкладку на «Done Orders», TabWidget перемикається на вкладку Виконані замовлення, яке зображено на рисунку 3.9.   
  
 

Рисунок 3.9 – Вкладка виконані замовлення

Натиснувши на кнопку «Загрузити завершені роботи», оновиться список замовлень які виконав працівник, натиснувши кнопку «Пошук», після того як введете критерій в рядок LineEdit, виконається пошук. Тепер входимо як замовник. Нас зустрічає вікно із двома вкладками, яке зображено на рисунку 3.10.

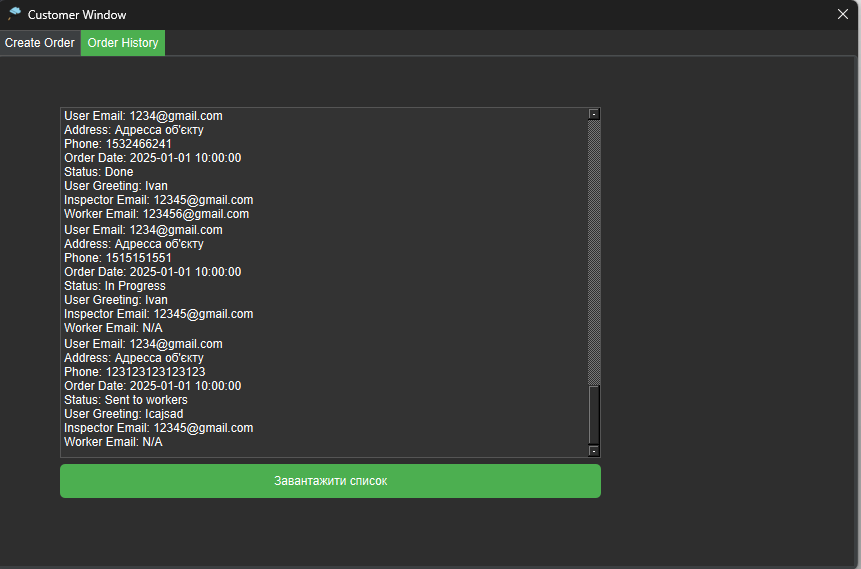


Рисунок 3.10 – Вікно замовника, вкладка історія замовлень

Натиснувши на кнопку «Завантажити список», список оновиться. Натиснувши на вкладку «Create Order», TabWidget перемикається на вкладку Створити замовлення, яке зображене на рисунку 3.11.

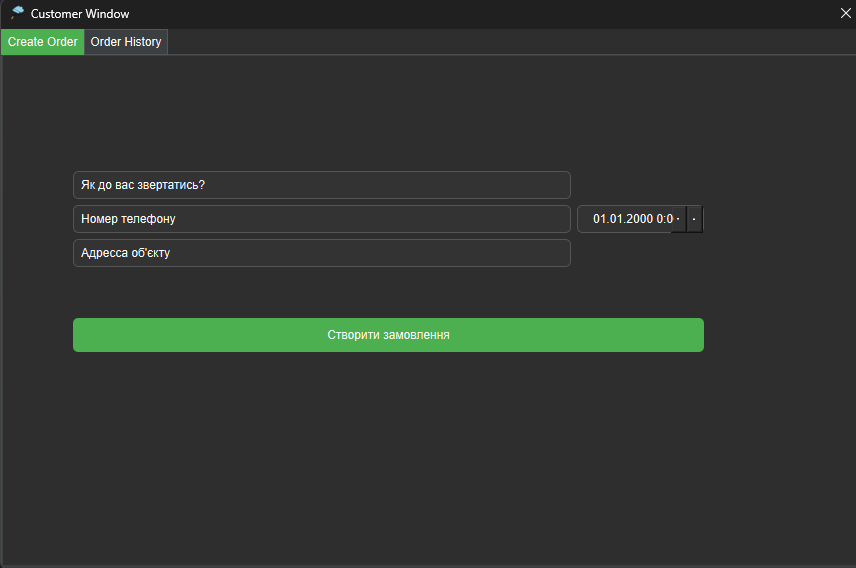


Рисунок 3.11 – Вкладка створити замовлення  
  
Натиснувши на кнопку створити замовлення форма відправляється в базу даних

Висновки

В даній курсовій роботі описано етапи розробки програми Tailored Design Solutions, яка призначена для полегшення ведення обліку інформації у сфері створення та реалізації індивідуального інтер’єру та екстер’єру. Розроблена програма відповідає усім поставленим вимогам технічного завдання.

Під час виконання курсової роботи було закріплено і поглиблено знання, одержанні за час вивчення предмету «Об'єктно-орієнтоване програмування», більш детально вивчено мову програмування С++. Для розробки даного програмного забезпечення використовувались принципи і концепції об'єктно- орієнтованого програмування, Використане середовище програмування Qt Creator, його особливі можливості і переваги, що дозволило швидко конструювати програмний засіб. Також під час розробки використано стандартні бібліотеки Qt, що дозволило в значній мірі пришвидшити розробку даної програми.

Перелік посилань

1. Слободян Р.О. Методичні вказівки до виконання курсової роботи з дисципліни «Об’єктно-орієнтоване програмування». Тернопіль : ВСП «ТФК ТНТУ», 2023. 45 с.
2. Грицюк Ю.І. Рак Т.Є. Об’єктно-орієнтоване програмування мовою С++ : навч. посіб. Львів : Вид-во Львівського ДУ БЖД, 2011. 404 с.
3. EVILEG : вебсайт. URL: https://evileg.com/uk/knowledge/qt.
4. Build with Qt. URL: <https://www1.qt.io/built-with-qt>.
5. Qt Documentation : вебсайт. URL: https://doc.qt.io/.

Додаток А  
Лістинг файлу «main.cpp»

#include "mainwindow.h"

#include <QApplication>

#include <QSqlDatabase>

#include <QSqlError>

#include <QSqlQuery>

#include <QDebug>

#include <QIcon>

#include "databasemanager.h"

void initializeDatabase(const QString& dbPath) {

DatabaseManager& dbManager = DatabaseManager::instance(dbPath);

if (!dbManager.isConnected()) {

qDebug() << "Database connection failed!";

return;

}

if (dbManager.createUsersTable()) {

qDebug() << "Users table initialized successfully.";

} else {

qDebug() << "Failed to initialize 'users' table.";

}

if (dbManager.createCustomerOrdersTable()) {

qDebug() << "Customer orders table initialized successfully.";

} else {

qDebug() << "Failed to initialize 'customerorder' table.";

}

if (dbManager.createInspectorOrdersTable()) {

qDebug() << "Inspector orders table initialized successfully.";

} else {

qDebug() << "Failed to initialize 'inspectororders' table.";

}

if (dbManager.createWorkerOrdersTable()) {

qDebug() << "Worker orders table initialized successfully.";

} else {

qDebug() << "Failed to initialize 'workerorders' table.";

}

}

int main(int argc, char \*argv[]) {

QApplication a(argc, argv);

Auth auth;

initializeDatabase("database.db");

MainWindow w;

w.show();

return a.exec();

}

Додаток Б  
Лістинг файлу «mainwindow.h»

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

#include "signupwindow.h"

#include "signinwindow.h"

QT\_BEGIN\_NAMESPACE

namespace Ui { class MainWindow; }

QT\_END\_NAMESPACE

class MainWindow : public QMainWindow

{

Q\_OBJECT

public:

MainWindow(QWidget \*parent = nullptr);

~MainWindow();

private slots:

void on\_signUp\_clicked();

void on\_signIn\_clicked();

void on\_exit\_clicked();

private:

Ui::MainWindow \*ui;

SignUpWindow \*signUpWindow;

SignInWindow \*signInWindow;

};

#endif // MAINWINDOW\_H

Додаток В  
Лістинг файлу «mainwindow.cpp»

#include "mainwindow.h"

#include "ui\_mainwindow.h"

MainWindow::MainWindow(QWidget \*parent)

: QMainWindow(parent)

, ui(new Ui::MainWindow)

, signUpWindow(nullptr)

, signInWindow(nullptr)

{

ui->setupUi(this);

setWindowTitle("Welcome to Tailored Design Solutions");

}

MainWindow::~MainWindow()

{

delete ui;

if (signUpWindow) delete signUpWindow;

if (signInWindow) delete signInWindow;

}

void MainWindow::on\_signUp\_clicked()

{

if (!signUpWindow) {

signUpWindow = new SignUpWindow();

}

signUpWindow->exec();

}

void MainWindow::on\_signIn\_clicked()

{

if (!signInWindow) {

signInWindow = new SignInWindow();

}

signInWindow->exec();

}

void MainWindow::on\_exit\_clicked()

{

close();

}

Додаток Г  
Лістинг файлу «databasemanager.h»

#ifndef DATABASEMANAGER\_H

#define DATABASEMANAGER\_H

#include <QSqlDatabase>

#include <QSqlError>

#include <QSqlQuery>

#include <QDebug>

#include <QMessageBox>

#include <QDate>

#include <QList>

#include <QString>

#include <QVector>

class OrderDetails {

public:

OrderDetails() = default;

OrderDetails(const QString &email, const QString &addr, const QString &ph, const QString &date, const QString &greeting,

const QString &status = "", const QString &inspectorEmail = "", const QString &workerEmail = "",

const QString &workDesc = "", const QString &media = "", const QString &materials = "", const QString &customerEmail = "");

int getOrderId() const;

void setOrderId(int orderId);

QString getUserEmail() const;

void setUserEmail(const QString &userEmail);

QString getUserName() const;

void setUserName(const QString &userName);

QString getAddress() const;

void setAddress(const QString &address);

QString getPhone() const;

void setPhone(const QString &phone);

QString getOrderDate() const;

void setOrderDate(const QString &orderDate);

QString getStatus() const;

void setStatus(const QString &status);

QString getUserGreeting() const;

void setUserGreeting(const QString &userGreeting);

QString getInspectorEmail() const;

void setInspectorEmail(const QString &inspectorEmail);

QString getWorkerEmail() const;

void setWorkerEmail(const QString &workerEmail);

QString getWorkDescription() const;

void setWorkDescription(const QString &workDescription);

QString getMediaLink() const;

void setMediaLink(const QString &mediaLink);

QString getRequiredMaterials() const;

void setRequiredMaterials(const QString &requiredMaterials);

QString getCustomerEmail() const;

void setCustomerEmail(const QString &customerEmail);

~OrderDetails () = default;

private:

int orderId = -1;

QString userEmail;

QString userName;

QString address;

QString phone;

QString orderDate;

QString status;

QString userGreeting;

QString inspectorEmail;

QString workerEmail;

QString workDescription;

QString mediaLink;

QString requiredMaterials;

QString customerEmail;

};

class DatabaseManager {

public:

static DatabaseManager& instance(const QString& dbPath = "database.db");

bool isConnected() const;

QSqlDatabase getDatabase() const;

bool createUsersTable();

bool createCustomerOrdersTable();

bool createInspectorOrdersTable();

bool createWorkerOrdersTable();

bool saveCustomerOrder(const OrderDetails &order);

bool loadCustomerOrders(const QString &email, QVector<OrderDetails> &orders);

bool loadOrdersForInspector(QList<OrderDetails>& ordersList);

bool takeOrderForInspector(int orderId, const QString &inspectorEmail);

bool updateOrderStatusForInspector(int orderId, const QString &status, const QString &inspectorEmail);

bool saveDetailsForInspector(int customerOrderId, const QString &mediaLink, const QString &workDescription, const QString &requiredMaterials, const QString &inspectorEmail);

bool loadInspectorOrdersForInspector(const QString &inspectorEmail, QVector<OrderDetails> &orders);

bool loadCompletedOrdersForInspector(const QString &inspectorEmail, QVector<OrderDetails> &orders);

bool loadAvaibleOrdersForWorkers(QList<OrderDetails>& ordersList);

bool loadAcceptedOrdersForWorkers(const QString &workerEmail, QList<OrderDetails> &ordersList);

bool acceptOrderForWorkers(int orderId, const QString &workerEmail);

bool saveWorkDetailsForWorkers(int orderId, const QString& workDescription, const QString& mediaLink, const QString& workerEmail);

bool getWorkDetailsForWorkers(int orderId, OrderDetails &order);

bool loadCompletedOrdersForWorkers(const QString &workerEmail, QList<OrderDetails> &completedOrders);

bool searchOrdersForInspector(const QString &searchText, QVector<OrderDetails> &orders);

bool searchAvailableOrdersForWorker(const QString &searchText, QList<OrderDetails> &orders);

~DatabaseManager();

private:

QSqlDatabase db;

explicit DatabaseManager(const QString& dbPath);

DatabaseManager(const DatabaseManager&) = delete;

DatabaseManager& operator=(const DatabaseManager&) = delete;

};

#endif // DATABASEMANAGER\_H

Додаток Ґ  
Лістинг файлу «databasemanager.cpp»

#include "databasemanager.h"

#include <QMessageBox>

DatabaseManager& DatabaseManager::instance(const QString& dbPath) {

static DatabaseManager instance(dbPath);

return instance;

}

DatabaseManager::DatabaseManager(const QString& dbPath) {

db = QSqlDatabase::addDatabase("QSQLITE");

db.setDatabaseName(dbPath);

if (!db.open()) {

qDebug() << "Error opening database:" << db.lastError().text();

}

}

bool DatabaseManager::isConnected() const {

return db.isOpen();

}

QSqlDatabase DatabaseManager::getDatabase() const {

return db;

}

DatabaseManager::~DatabaseManager() {

if (db.isOpen()) {

db.close();

}

}

OrderDetails::OrderDetails(const QString &email, const QString &addr, const QString &ph, const QString &date, const QString &greeting,

const QString &status, const QString &inspectorEmail, const QString &workerEmail,

const QString &workDesc, const QString &media, const QString &materials, const QString &customerEmail)

: userEmail(email), address(addr), phone(ph), orderDate(date), userGreeting(greeting),

status(status), inspectorEmail(inspectorEmail), workerEmail(workerEmail),

workDescription(workDesc), mediaLink(media), requiredMaterials(materials), customerEmail(customerEmail) {

}

int OrderDetails::getOrderId() const {

return orderId;

}

void OrderDetails::setOrderId(int orderId) {

this->orderId = orderId;

}

QString OrderDetails::getUserEmail() const {

return userEmail;

}

void OrderDetails::setUserEmail(const QString &email) {

this->userEmail = email;

}

QString OrderDetails::getUserName() const {

return userName;

}

void OrderDetails::setUserName(const QString &name) {

this->userName = name;

}

QString OrderDetails::getAddress() const {

return address;

}

void OrderDetails::setAddress(const QString &addr) {

this->address = addr;

}

QString OrderDetails::getPhone() const {

return phone;

}

void OrderDetails::setPhone(const QString &ph) {

this->phone = ph;

}

QString OrderDetails::getOrderDate() const {

return orderDate;

}

void OrderDetails::setOrderDate(const QString &date) {

this->orderDate = date;

}

QString OrderDetails::getStatus() const {

return status;

}

void OrderDetails::setStatus(const QString &st) {

this->status = st;

}

QString OrderDetails::getUserGreeting() const {

return userGreeting;

}

void OrderDetails::setUserGreeting(const QString &greeting) {

this->userGreeting = greeting;

}

QString OrderDetails::getInspectorEmail() const {

return inspectorEmail;

}

void OrderDetails::setInspectorEmail(const QString &email) {

this->inspectorEmail = email;

}

QString OrderDetails::getWorkerEmail() const {

return workerEmail;

}

void OrderDetails::setWorkerEmail(const QString &email) {

this->workerEmail = email;

}

QString OrderDetails::getWorkDescription() const {

return workDescription;

}

void OrderDetails::setWorkDescription(const QString &desc) {

this->workDescription = desc;

}

QString OrderDetails::getMediaLink() const {

return mediaLink;

}

void OrderDetails::setMediaLink(const QString &link) {

this->mediaLink = link;

}

QString OrderDetails::getRequiredMaterials() const {

return requiredMaterials;

}

void OrderDetails::setRequiredMaterials(const QString &materials) {

this->requiredMaterials = materials;

}

QString OrderDetails::getCustomerEmail() const {

return customerEmail;

}

void OrderDetails::setCustomerEmail(const QString &email) {

this->customerEmail = email;

}

bool DatabaseManager::createUsersTable() {

QSqlQuery query(db);

bool success = query.exec(

"CREATE TABLE IF NOT EXISTS users ("

"id INTEGER PRIMARY KEY AUTOINCREMENT, "

"firstName TEXT NOT NULL, "

"lastName TEXT NOT NULL, "

"email TEXT NOT NULL UNIQUE, "

"password TEXT NOT NULL, "

"userType TEXT NOT NULL)"

);

if (!success) {

qDebug() << "Error creating users table:" << query.lastError().text();

}

return success;

}

bool DatabaseManager::createCustomerOrdersTable() {

QSqlQuery query(db);

bool success = query.exec(

"CREATE TABLE IF NOT EXISTS customerorder ("

"id INTEGER PRIMARY KEY AUTOINCREMENT, "

"user\_email TEXT, "

"address TEXT, "

"phone TEXT, "

"orderDate TEXT, "

"status TEXT DEFAULT 'В обробці', "

"userGreeting TEXT, "

"inspectorEmail TEXT DEFAULT NULL, "

"workerEmail TEXT DEFAULT NULL, "

"FOREIGN KEY (user\_email) REFERENCES users(email))"

);

if (!success) {

qDebug() << "Error creating customerorder table:" << query.lastError().text();

}

return success;

}

bool DatabaseManager::createInspectorOrdersTable() {

QSqlQuery query(db);

bool success = query.exec(

"CREATE TABLE IF NOT EXISTS inspectororders ("

"id INTEGER PRIMARY KEY AUTOINCREMENT, "

"customerOrderId INTEGER NOT NULL, "

"inspectorEmail TEXT NOT NULL, "

"mediaLink TEXT, "

"workDescription TEXT, "

"requiredMaterials TEXT, "

"FOREIGN KEY (customerOrderId) REFERENCES customerorder(id), "

"FOREIGN KEY (inspectorEmail) REFERENCES users(email))"

);

if (!success) {

qDebug() << "Error creating inspectororders table:" << query.lastError().text();

}

return success;

}

bool DatabaseManager::createWorkerOrdersTable() {

QSqlQuery query(db);

bool success = query.exec(

"CREATE TABLE IF NOT EXISTS workerorders ("

"id INTEGER PRIMARY KEY AUTOINCREMENT, "

"workerEmail TEXT NOT NULL, "

"customerOrderId INTEGER NOT NULL, "

"workDescription TEXT, "

"mediaLink TEXT, "

"FOREIGN KEY (customerOrderId) REFERENCES customerorder(id), "

"FOREIGN KEY (workerEmail) REFERENCES users(email))"

);

if (!success) {

qDebug() << "Error creating workerorders table:" << query.lastError().text();

}

return success;

}

bool DatabaseManager::saveCustomerOrder(const OrderDetails &order) {

QSqlQuery query;

query.prepare("INSERT INTO customerorder (user\_email, address, phone, orderDate, status, userGreeting, inspectorEmail) "

"VALUES (:user\_email, :address, :phone, :orderDate, :status, :userGreeting, :inspectorEmail)");

query.bindValue(":user\_email", order.getUserEmail());

query.bindValue(":address", order.getAddress());

query.bindValue(":phone", order.getPhone());

query.bindValue(":orderDate", order.getOrderDate());

query.bindValue(":status", order.getStatus().isEmpty() ? "Pending" : order.getStatus());

query.bindValue(":userGreeting", order.getUserGreeting());

query.bindValue(":inspectorEmail", order.getInspectorEmail().isEmpty() ? QVariant() : order.getInspectorEmail());

qDebug() << "Executing query with values:"

<< order.getUserEmail()

<< order.getAddress()

<< order.getPhone()

<< order.getOrderDate()

<< order.getUserGreeting();

if (!query.exec()) {

qDebug() << "Failed to execute query. Error:" << query.lastError().text();

return false;

}

return true;

}

bool DatabaseManager::loadCustomerOrders(const QString &email, QVector<OrderDetails>& orders) {

QSqlQuery query;

query.prepare("SELECT user\_email, address, phone, orderDate, status, userGreeting, inspectorEmail, workerEmail "

"FROM customerorder WHERE user\_email = :email");

query.bindValue(":email", email);

if (!query.exec()) {

qDebug() << "Failed to load orders: " << query.lastError().text();

return false;

}

while (query.next()) {

OrderDetails order;

order.setUserEmail(query.value("user\_email").toString());

order.setAddress(query.value("address").toString());

order.setPhone(query.value("phone").toString());

order.setOrderDate(query.value("orderDate").toString());

order.setStatus(query.value("status").toString());

order.setUserGreeting(query.value("userGreeting").toString());

order.setInspectorEmail(query.value("inspectorEmail").toString());

order.setWorkerEmail(query.value("workerEmail").toString());

orders.append(order);

}

return true;

}

bool DatabaseManager::loadOrdersForInspector(QList<OrderDetails>& ordersList)

{

QSqlQuery query;

query.prepare("SELECT id, user\_email, address, phone, orderDate, status FROM customerorder WHERE status = 'Pending'");

if (!query.exec()) {

qDebug() << "Error loading orders:" << query.lastError().text();

return false;

}

while (query.next()) {

OrderDetails order;

order.setOrderId(query.value("id").toInt());

order.setUserEmail(query.value("user\_email").toString());

order.setAddress(query.value("address").toString());

order.setPhone(query.value("phone").toString());

order.setOrderDate(query.value("orderDate").toString());

order.setStatus(query.value("status").toString());

ordersList.append(order);

}

return true;

}

bool DatabaseManager::takeOrderForInspector(int orderId, const QString &inspectorEmail) {

QSqlQuery query;

query.prepare("SELECT id FROM customerorder WHERE id = :id");

query.bindValue(":id", orderId);

if (!query.exec() || !query.next()) {

qDebug() << "Order not found in the database.";

return false;

}

query.prepare("UPDATE customerorder SET status = :status, inspectorEmail = :inspectorEmail WHERE id = :id");

query.bindValue(":status", "In Progress");

query.bindValue(":inspectorEmail", inspectorEmail);

query.bindValue(":id", orderId);

if (!query.exec()) {

qDebug() << "Failed to update order status: " << query.lastError().text();

return false;

}

qDebug() << "Order ID " << orderId << " status updated to 'In Progress' by inspector: " << inspectorEmail;

return true;

}

bool DatabaseManager::updateOrderStatusForInspector(int orderId, const QString &status, const QString &inspectorEmail) {

QSqlQuery query;

query.prepare("SELECT COUNT(\*) FROM customerorder WHERE id = :id");

query.bindValue(":id", orderId);

if (!query.exec() || !query.next() || query.value(0).toInt() == 0) {

QMessageBox::warning(nullptr, "Error", "Order with the specified ID does not exist.");

qDebug() << "Order with ID" << orderId << "not found in the database.";

return false;

}

query.prepare("UPDATE customerorder SET status = :status, inspectorEmail = :inspectorEmail WHERE id = :id");

query.bindValue(":status", status);

query.bindValue(":inspectorEmail", inspectorEmail);

query.bindValue(":id", orderId);

if (!query.exec()) {

QMessageBox::critical(nullptr, "Error", "Failed to update order status. " + query.lastError().text());

qDebug() << "Database error:" << query.lastError().text();

return false;

}

qDebug() << "Order ID:" << orderId << "status updated to:" << status << "by inspector:" << inspectorEmail;

return true;

}

bool DatabaseManager::saveDetailsForInspector(int customerOrderId, const QString &mediaLink, const QString &workDescription, const QString &requiredMaterials, const QString &inspectorEmail) {

QSqlQuery query;

query.prepare("UPDATE customerorder SET status = :status, inspectorEmail = :inspectorEmail WHERE id = :id");

query.bindValue(":status", "Sent to workers");

query.bindValue(":inspectorEmail", inspectorEmail);

query.bindValue(":id", customerOrderId);

if (!query.exec()) {

QMessageBox::critical(nullptr, "Error", "Failed to update customer order: " + query.lastError().text());

return false;

}

query.prepare("INSERT INTO inspectororders (customerOrderId, inspectorEmail, mediaLink, workDescription, requiredMaterials) "

"VALUES (:customerOrderId, :inspectorEmail, :mediaLink, :workDescription, :requiredMaterials)");

query.bindValue(":customerOrderId", customerOrderId);

query.bindValue(":inspectorEmail", inspectorEmail);

query.bindValue(":mediaLink", mediaLink);

query.bindValue(":workDescription", workDescription);

query.bindValue(":requiredMaterials", requiredMaterials);

if (!query.exec()) {

QMessageBox::critical(nullptr, "Error", "Failed to insert inspector order: " + query.lastError().text());

return false;

}

return true;

}

bool DatabaseManager::loadInspectorOrdersForInspector(const QString &inspectorEmail, QVector<OrderDetails> &orders) {

QSqlQuery query;

query.prepare("SELECT DISTINCT c.id, c.user\_email, c.userGreeting, c.status, c.address, c.phone, "

"i.workDescription, i.mediaLink, c.orderDate "

"FROM customerorder c "

"LEFT JOIN inspectororders i ON c.id = i.customerOrderId "

"WHERE c.status = 'In Progress' "

"AND c.inspectorEmail = :inspectorEmail");

query.bindValue(":inspectorEmail", inspectorEmail);

if (!query.exec()) {

QMessageBox::warning(nullptr, "Error", "Failed to load orders: " + query.lastError().text());

return false;

}

if (query.size() == 0) {

QMessageBox::information(nullptr, "No Orders", "No orders are currently in progress.");

return true;

}

orders.clear();

while (query.next()) {

OrderDetails order;

order.setOrderId(query.value("id").toInt());

order.setUserEmail(query.value("user\_email").toString());

order.setUserName(query.value("userGreeting").toString());

order.setStatus(query.value("status").toString());

order.setAddress(query.value("address").toString());

order.setPhone(query.value("phone").toString());

order.setWorkDescription(query.value("workDescription").toString());

order.setMediaLink(query.value("mediaLink").toString());

QDateTime createdAt = query.value("orderDate").toDateTime();

order.setOrderDate(createdAt.toString("yyyy-MM-dd hh:mm:ss"));

orders.append(order);

}

return true;

}

bool DatabaseManager::loadCompletedOrdersForInspector(const QString &inspectorEmail, QVector<OrderDetails> &orders) {

QSqlQuery query;

query.prepare(

"SELECT DISTINCT c.id, c.user\_email, c.userGreeting, c.status, c.address, c.phone, "

"i.mediaLink, i.workDescription, i.requiredMaterials, c.orderDate "

"FROM customerorder c "

"LEFT JOIN inspectororders i ON c.id = i.customerOrderId "

"WHERE (c.status = 'Done' OR c.status = 'Sent to workers') AND c.inspectorEmail = :inspectorEmail"

);

query.bindValue(":inspectorEmail", inspectorEmail);

if (!query.exec()) {

QMessageBox::warning(nullptr, "Error", "Failed to load completed orders: " + query.lastError().text());

return false;

}

if (query.size() == 0) {

QMessageBox::information(nullptr, "No Orders", "No completed orders found.");

return true;

}

while (query.next()) {

OrderDetails order;

order.setOrderId(query.value("id").toInt());

order.setUserEmail(query.value("user\_email").toString());

order.setUserName(query.value("userGreeting").toString());

order.setStatus(query.value("status").toString());

order.setAddress(query.value("address").toString());

order.setPhone(query.value("phone").toString());

order.setMediaLink(query.value("mediaLink").toString());

order.setWorkDescription(query.value("workDescription").toString());

order.setRequiredMaterials(query.value("requiredMaterials").toString());

order.setOrderDate(query.value("orderDate").toString());

orders.append(order);

}

return true;

}

bool DatabaseManager::loadAvaibleOrdersForWorkers(QList<OrderDetails>& ordersList) {

QSqlQuery query;

query.prepare(

"SELECT DISTINCT co.id, co.user\_email, co.address, co.phone, co.orderDate, "

"co.status, co.userGreeting, co.inspectorEmail, "

"i.workDescription, i.mediaLink "

"FROM customerorder co "

"LEFT JOIN inspectororders i ON co.id = i.customerOrderId "

"WHERE co.status = 'Sent to workers'"

);

if (!query.exec()) {

qDebug() << "Failed to load available orders:" << query.lastError().text();

return false;

}

ordersList.clear();

while (query.next()) {

OrderDetails order;

order.setUserEmail(query.value("user\_email").toString());

order.setAddress(query.value("address").toString());

order.setPhone(query.value("phone").toString());

order.setOrderDate(query.value("orderDate").toString());

order.setUserName(query.value("userGreeting").toString());

order.setStatus(query.value("status").toString());

order.setInspectorEmail(query.value("inspectorEmail").toString());

order.setWorkDescription(query.value("workDescription").toString());

order.setMediaLink(query.value("mediaLink").toString());

order.setOrderId(query.value("id").toInt());

ordersList.append(order);

}

return true;

}

bool DatabaseManager::loadAcceptedOrdersForWorkers(const QString& workerEmail, QList<OrderDetails>& ordersList)

{

QSqlQuery query;

query.prepare(

"SELECT DISTINCT c.id, c.user\_email, c.address, c.phone, i.workDescription, i.mediaLink, c.status, "

"c.orderDate, i.requiredMaterials, i.inspectorEmail "

"FROM customerorder c "

"LEFT JOIN inspectororders i ON c.id = i.customerOrderId "

"WHERE c.status = 'Accepted by workers' AND c.workerEmail = :workerEmail"

);

query.bindValue(":workerEmail", workerEmail);

if (!query.exec()) {

QMessageBox::warning(nullptr, "Error", "Failed to load accepted orders: " + query.lastError().text());

return false;

}

while (query.next()) {

OrderDetails orderDetails;

orderDetails.setOrderId(query.value("id").toInt());

orderDetails.setCustomerEmail(query.value("user\_email").toString());

orderDetails.setAddress(query.value("address").toString());

orderDetails.setPhone(query.value("phone").toString());

orderDetails.setWorkDescription(query.value("workDescription").toString());

orderDetails.setMediaLink(query.value("mediaLink").toString());

orderDetails.setStatus(query.value("status").toString());

orderDetails.setOrderDate(query.value("orderDate").toString());

orderDetails.setRequiredMaterials(query.value("requiredMaterials").toString());

orderDetails.setInspectorEmail(query.value("inspectorEmail").toString());

ordersList.append(orderDetails);

}

return true;

}

bool DatabaseManager::acceptOrderForWorkers(int orderId, const QString& workerEmail)

{

QSqlQuery query;

query.prepare("UPDATE customerorder SET status = :status, workerEmail = :workerEmail WHERE id = :id");

query.bindValue(":status", "Accepted by workers");

query.bindValue(":workerEmail", workerEmail);

query.bindValue(":id", orderId);

if (!query.exec()) {

qDebug() << "Error accepting order:" << query.lastError().text();

return false;

}

return true;

}

bool DatabaseManager::saveWorkDetailsForWorkers(int orderId, const QString& workDescription, const QString& mediaLink, const QString& workerEmail)

{

QSqlQuery query;

query.prepare("SELECT status FROM customerorder WHERE id = :id");

query.bindValue(":id", orderId);

if (!query.exec()) {

qDebug() << "Error checking order status:" << query.lastError().text();

return false;

}

if (query.next()) {

QString status = query.value("status").toString();

if (status != "Accepted by workers") {

qDebug() << "Order is not in 'Accepted by workers' status.";

return false;

}

}

query.prepare("INSERT INTO workerorders (customerOrderId, workerEmail, workDescription, mediaLink) "

"VALUES (:customerOrderId, :workerEmail, :workDescription, :mediaLink)");

query.bindValue(":customerOrderId", orderId);

query.bindValue(":workerEmail", workerEmail);

query.bindValue(":workDescription", workDescription);

query.bindValue(":mediaLink", mediaLink);

if (!query.exec()) {

qDebug() << "Error saving work details:" << query.lastError().text();

return false;

}

query.prepare("UPDATE customerorder SET status = 'Done' WHERE id = :id");

query.bindValue(":id", orderId);

if (!query.exec()) {

qDebug() << "Error updating order status:" << query.lastError().text();

return false;

}

return true;

}

bool DatabaseManager::getWorkDetailsForWorkers(int orderId, OrderDetails& orderDetails)

{

QSqlQuery query;

query.prepare("SELECT c.id, c.user\_email, c.address, c.phone, c.orderDate, c.status, "

"i.workDescription, i.mediaLink, i.inspectorEmail "

"FROM customerorder c "

"LEFT JOIN inspectororders i ON c.id = i.customerOrderId "

"WHERE c.id = :orderId");

query.bindValue(":orderId", orderId);

if (!query.exec() || !query.next()) {

qDebug() << "Error retrieving work details:" << query.lastError().text();

return false;

}

orderDetails.setOrderId(query.value("id").toInt());

orderDetails.setCustomerEmail(query.value("user\_email").toString());

orderDetails.setAddress(query.value("address").toString());

orderDetails.setPhone(query.value("phone").toString());

orderDetails.setOrderDate(query.value("orderDate").toString());

orderDetails.setStatus(query.value("status").toString());

orderDetails.setWorkDescription(query.value("workDescription").toString());

orderDetails.setMediaLink(query.value("mediaLink").toString());

orderDetails.setInspectorEmail(query.value("inspectorEmail").toString());

return true;

}

bool DatabaseManager::loadCompletedOrdersForWorkers(const QString& workerEmail, QList<OrderDetails>& completedOrders)

{

QSqlQuery query;

query.prepare(

"SELECT DISTINCT c.id, c.user\_email, c.address, i.inspectorEmail, i.workDescription, i.mediaLink "

"FROM customerorder c "

"LEFT JOIN inspectororders i ON c.id = i.customerOrderId "

"WHERE c.status = 'Done' AND c.workerEmail = :workerEmail"

);

query.bindValue(":workerEmail", workerEmail);

if (!query.exec()) {

qDebug() << "Error loading completed orders:" << query.lastError().text();

return false;

}

while (query.next()) {

OrderDetails orderDetails;

orderDetails.setOrderId(query.value("id").toInt());

orderDetails.setCustomerEmail(query.value("user\_email").toString());

orderDetails.setAddress(query.value("address").toString());

orderDetails.setInspectorEmail(query.value("inspectorEmail").toString());

orderDetails.setWorkDescription(query.value("workDescription").toString());

orderDetails.setMediaLink(query.value("mediaLink").toString());

completedOrders.append(orderDetails);

}

return true;

}

bool DatabaseManager::searchOrdersForInspector(const QString &searchText, QVector<OrderDetails> &orders) {

QSqlQuery query(db);

query.prepare(

"SELECT c.id, c.user\_email, c.userGreeting, c.status, c.address, c.phone, c.orderDate, "

"CASE "

" WHEN c.status = 'Sent to workers' THEN i.workDescription "

" WHEN c.status = 'Done' THEN w.workDescription "

" ELSE NULL "

"END AS workDescription, "

"CASE "

" WHEN c.status = 'Sent to workers' THEN i.mediaLink "

" WHEN c.status = 'Done' THEN w.mediaLink "

" ELSE NULL "

"END AS mediaLink "

"FROM customerorder c "

"LEFT JOIN inspectororders i ON c.id = i.customerOrderId "

"LEFT JOIN workerorders w ON c.id = w.customerOrderId "

"WHERE (LOWER(c.user\_email) LIKE LOWER(:searchText) "

"OR LOWER(c.address) LIKE LOWER(:searchText) "

"OR LOWER(c.status) LIKE LOWER(:searchText)) "

"AND (c.status = 'Done' OR c.status = 'Sent to workers')"

);

QString searchPattern = "%" + searchText + "%";

query.bindValue(":searchText", searchPattern);

qDebug() << "Executing query:" << query.executedQuery();

qDebug() << "Bound values:" << query.boundValues();

if (!query.exec()) {

qDebug() << "Error executing search query:" << query.lastError().text();

return false;

}

orders.clear();

while (query.next()) {

OrderDetails order;

order.setOrderId(query.value("id").toInt());

order.setUserEmail(query.value("user\_email").toString());

order.setUserName(query.value("userGreeting").toString());

order.setStatus(query.value("status").toString());

order.setAddress(query.value("address").toString());

order.setPhone(query.value("phone").toString());

order.setWorkDescription(query.value("workDescription").toString());

order.setMediaLink(query.value("mediaLink").toString());

order.setOrderDate(query.value("orderDate").toString());

qDebug() << "Loaded order:" << order.getOrderId() << order.getUserEmail() << order.getStatus();

orders.append(order);

}

qDebug() << "Total orders found:" << orders.size();

return true;

}

bool DatabaseManager::searchAvailableOrdersForWorker(const QString &searchText, QList<OrderDetails> &orders)

{

QSqlQuery query;

query.prepare(

"SELECT c.id, c.user\_email, c.userGreeting, c.status, c.address, c.phone, c.orderDate, "

"CASE "

" WHEN c.status = 'Done' THEN w.workDescription "

" ELSE NULL "

"END AS workDescription, "

"CASE "

" WHEN c.status = 'Done' THEN w.mediaLink "

" ELSE NULL "

"END AS mediaLink "

"FROM customerorder c "

"LEFT JOIN workerorders w ON c.id = w.customerOrderId "

"WHERE (LOWER(c.user\_email) LIKE LOWER(:searchText) "

"OR LOWER(c.address) LIKE LOWER(:searchText) "

"OR LOWER(c.status) LIKE LOWER(:searchText)) "

"AND c.status = 'Done'"

);

query.bindValue(":searchText", "%" + searchText + "%");

if (!query.exec()) {

qDebug() << "Database error: " << query.lastError().text();

return false;

}

orders.clear();

while (query.next()) {

OrderDetails order;

order.setOrderId(query.value("id").toInt());

order.setUserEmail(query.value("user\_email").toString());

order.setUserName(query.value("userGreeting").toString());

order.setStatus(query.value("status").toString());

order.setAddress(query.value("address").toString());

order.setPhone(query.value("phone").toString());

order.setWorkDescription(query.value("workDescription").toString());

order.setMediaLink(query.value("mediaLink").toString());

order.setOrderDate(query.value("orderDate").toString());

orders.append(order);

}

return true;

}