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: **«**Розробка програмного забезпечення для ведення обліку розрахунків за комунальні платежі.**»**

Студента  3  курсу групи КН-323 спеціальності 122 «Комп’ютерні науки»

Ягода В.В.

(прізвище та ініціали)

Керівник: викладач   Лісовий В.М.         
Національна шкала \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

Члени комісії: \_\_\_\_\_\_\_\_\_ Володимир ЛІСОВИЙ

(підпис)

\_\_\_\_\_\_\_\_ Роман ДІЛАЙ

(підпис)

м. Тернопіль – 2023

ЗМІСТ

[Вступ 4](#_Toc152092428)

[1 Технічне завдання 5](#_Toc152092429)

[1.1 Найменування та область застосування 5](#_Toc152092430)

[1.2 Підстави для розробки 5](#_Toc152092431)

[1.3 Призначення розробки 5](#_Toc152092432)

[1.4 Вимоги до програми чи програмного виробу 5](#_Toc152092433)

[1.5 Вимоги до програмної документації 6](#_Toc152092434)

[1.6 Техніко-економічні показники 6](#_Toc152092435)

[1.7 Стадії та етапи розробки 6](#_Toc152092436)

[1.8 Порядок контролю та прийому 6](#_Toc152092437)

[2 Розробка технічного та робочого проєкту 7](#_Toc152092438)

[2.1 Розробка загальної структури і варіантів використання програми 7](#_Toc152092439)

[2.2 Розробка системи класів 7](#_Toc152092440)

[2.3 Розробка методів 7](#_Toc152092441)

[2.4 Проєктування і опис інтерфейсу користувача 7](#_Toc152092442)

[2.5 Опис файлової структури програми 7](#_Toc152092443)

[2.6 Опис структури бази даних програми 7](#_Toc152092444)

[3 Тестування програми і результати її виконання 8](#_Toc152092445)

[Висновки 9](#_Toc152092446)

[Перелік посилань 10](#_Toc152092447)

[Додаток А Лістинг файлу «myprogram.pro» 11](#_Toc152092448)

[Додаток Б Лістинг файлу «myclass.h» 12](#_Toc152092449)

[Додаток В Лістинг файлу «myclass.cpp» 13](#_Toc152092450)

[Додаток Г Для кожного програмного файлу окремий додаток 14](#_Toc152092451)

[Додаток Ґ Компакт-диск із програмним продуктом 15](#_Toc152092452)

Вступ

Сучасний світ заснований на великій кількості взаємодій і обмінів, які відбуваються кожен день. Однією з найважливіших частин щоденної життєдіяльності громадян є сплата комунальних платежів. З метою полегшення цього процесу та вдосконалення обліку розрахунків за комунальні послуги, виникає необхідність у створенні спеціалізованого програмного забезпечення.

Курсова робота "Розробка програмного забезпечення для ведення обліку розрахунків за комунальні платежі" присвячена створенню програми, яка допоможе ефективно вести облік та спрощувати процес розрахунків за комунальні послуги. У цій курсовій роботі ми розглянемо процес розробки програми з використанням мови програмування C++ та бібліотеки Qt для створення графічного інтерфейсу користувача.

Робота включатиме в себе архітектурне проектування програмного забезпечення, взаємодію з базою даних для зберігання інформації про користувачів, рахунки та платежі. Крім того, буде розглянута можливість впровадження функцій додавання та видалення користувачів, рахунків, а також функції введення платежів. Розглянемо також створення звітів та можливість генерації статистики для користувачів.

Ця курсова робота спрямована на розвиток навичок розробки програмного забезпечення з використанням сучасних технологій та інструментів, а також на вивчення принципів взаємодії з базою даних для забезпечення ефективного управління розрахунками за комунальні послуги.

# Технічне завдання

## Найменування та область застосування

Найменування програми – Програма для ведення обліку розрахунків за комунальні платежі

Стисла назва – «Taxpayer»

Область застосування програми – Програма "Розробка програмного забезпечення для ведення обліку розрахунків за комунальні платежі" знаходить свою область застосування в широкому спектрі відомостей та в усіх сферах, пов'язаних із сплатою комунальних послуг. Вона спрямована на полегшення процесів ведення особистого обліку громадянами, оптимізацію взаємодії з управляючими органами та забезпечення ефективного фінансового управління для бухгалтерій та державних установ. Ця програма також відкриває нові можливості для розробників програмного забезпечення, дозволяючи їм вивчати сучасні технології та принципи взаємодії.

## Підстави для розробки

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

Найменування теми курсової роботи – «Розробка програмного забезпечення для ведення обліку розрахунків за комунальні платежі».

Замовник – керівник курсової роботи, викладач Слободян Руслан Олесійович.

Виконавець ­– студент групи КН-323 Ягода Віталій Володимирович

## Призначення розробки

**Ціль проекту "Розробка програмного забезпечення для ведення обліку розрахунків за комунальні платежі" полягає у створенні зручного та ефективного інструменту для обліку комунальних витрат. Програма націлена на спрощення ведення особистого обліку громадянами, автоматизацію процесів введення та збереження інформації про платежі, а також на оптимізацію взаємодії з управляючими компаніями та надання можливостей для розвитку розробників ПЗ.**

## Вимоги до програми чи програмного виробу

### Вимоги до функціональних характеристик

Програмне забезпечення для ведення обліку розрахунків за комунальні платежі розроблено з урахуванням ключових функціональних характеристик, спрямованих на зручність та ефективність використання. Основні вимоги включають:

* Облік Користувачів

Система дозволяє реєструвати користувачів з унікальним ідентифікатором та персональною інформацією. Кожен користувач може вести облік своїх комунальних платежів та взаємодіяти з програмою через особистий кабінет.

* Сплата Платежів З Різних Рахунків

Користувачі можуть обирати конкретний рахунок (account), з якого вони хочуть здійснити платіж за комунальні послуги. Система дозволяє вибирати з різних рахунків під час введення інформації про платіж, щоб забезпечити гнучкість та зручність користувачів.

* **Введення та Збереження Платежів**

Користувачі можуть вводити дані про свої комунальні платежі, зазначаючи вид послуг, суму, дату оплати та інші необхідні деталі. Система автоматично зберігає цю інформацію для подальшого використання та аналізу.

### Вимоги до часових характеристик

Вимоги до часових характеристик програми мають на меті забезпечення її оптимальної продуктивності та вчасного відгуку на дії користувачів. Програма повинна демонструвати ефективність у виконанні різних операцій, зокрема, швидко обробляти та зберігати дані, генерувати звіти та відображати інформацію в реальному часі.

### Вимоги до надійності

Надійна функціональність програми повинна гарантуватися встановленням механізму перевірки вірності введених даних. Це дозволить уникнути помилок, які можуть виникнути під час виконання рутинних завдань користувачем. Крім того, необхідно забезпечити ефективний захист програми від некваліфікованих втручань користувача шляхом надання чітких та зрозумілих повідомлень під час будь-яких операцій в програмі.

### Умови до експлуатації

Розроблений програмний продукт не вимагатиме спеціального технічного обслуговування і може використовуватися звичайними навичками користувача комп'ютера.

### Вимоги до складу і параметрів технічних засобів

Для оптимального функціонування програмного забезпечення рекомендується використовувати обладнання із наступними характеристиками:

* + Монітор із діагоналлю не менше 15 дюймів.
  + Клавіатура та мишка для зручного введення та взаємодії з програмою.
  + Персональний комп'ютер, сумісний із стандартами IBM.

Мінімальні технічні вимоги для персонального комп'ютера:

* Процесор Intel Pentium 4 або еквівалентний.
* Мінімум 2 ГБ оперативної пам'яті.
* Обсяг дискової пам'яті не менше 2 ГБ.

## Вимоги до програмної документації

По закінченню розробки програмного забезпечення потрібно підготувати таку документацію:

- інструкція інсталяції програми;

- загальні відомості про можливості програми;

- інструкція з експлуатації.

## Техніко-економічні показники

Розрахунок економічної ефективності і вартості розробки програмного продукту не проводиться. Приблизне число використань розробленої програми в рік – 12-50 раз.

## Стадії та етапи розробки

Розробка програми обліку прибутково-видаткових операцій домовласника будинку буде мати такі стадії:

Етап 1: Аналіз

На цьому етапі проводиться детальний аналіз предметної області задачі. Визначаються основні об'єкти та їх взаємозв'язки, проводиться об'єктна декомпозиція програми. Створюється структурна схема програми, в якій відображаються об'єкти та події над ними.

Етап 2: Логічне проектування

Розробка структури класів, що включає в себе визначення полів для збереження складових об'єктів та алгоритмів методів. Використовуються прийоми спадкування, композиції та наповнення для створення ієрархії класів. Результатом цього етапу є опис класів.

Етап 3: Фізичне проектування

Фізичне проектування включає в себе об'єднання класів у модулі, вибір схеми їх підключення (статична або динамічна), визначення способів взаємодії з обладнанням, операційною системою та іншим програмним забезпеченням. Тут також вирішується питання синхронізації процесів для систем рівнобіжної обробки.

Етап 4: Еволюція

Починається зі створення основного інтерфейсу проекту та бібліотеки класів. Описуються типи полів класу, розробляються алгоритми методів та вносяться зміни відповідно до вимог програми. Розроблений прототип тестується та налагоджується.

Етап 5: Тестування та Оптимізація

На цьому етапі виконується тестування програми з метою виявлення та виправлення помилок. Проводяться відповідні оптимізації для підвищення продуктивності та ефективності програми.

Етап 6: Впровадження та Підтримка

Після успішного завершення тестування програма готова до впровадження. На цьому етапі здійснюється розгортання програми та надається необхідна підтримка для користувачів.

## Порядок контролю та прийому

Прийом розробленого програмного забезпечення повинен відбуватися на об’єкті Замовника в терміни, які зазначені в індивідуальному завданні.

Для прийому роботи Виконавець повинен представити:

− діючу програму, яка повністю відповідає даному технічному завданню;

− вихідний програмний код, записаний разом із програмою на оптичний 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 з програмою Taxpayer можуть працювати кілька користувачів. Таким чином, один користувач вводить інформацію, здійснює розрахунки і переглядає їх у різних видах. Наступне, що потрібно було з'ясувати, це групу дій, які ініціюватиме діючий суб'єкт. Виходячи із технічного завдання, діючий суб'єкт – домовласник буде виконувати такі дії:

* Запустити програму
* Увійти
* Зареєструватись та увійти
* Створити рахунок
* Обрати існуючий рахунок
* Поповнити рахунок
* Додати платіж
* Переглянути інформацію про існуючий платіж

Таким чином, діаграму варіантів використання, яка отримується внаслідок наведеного переліку дій, можна подати так, як це показано на рисунку 2.1.

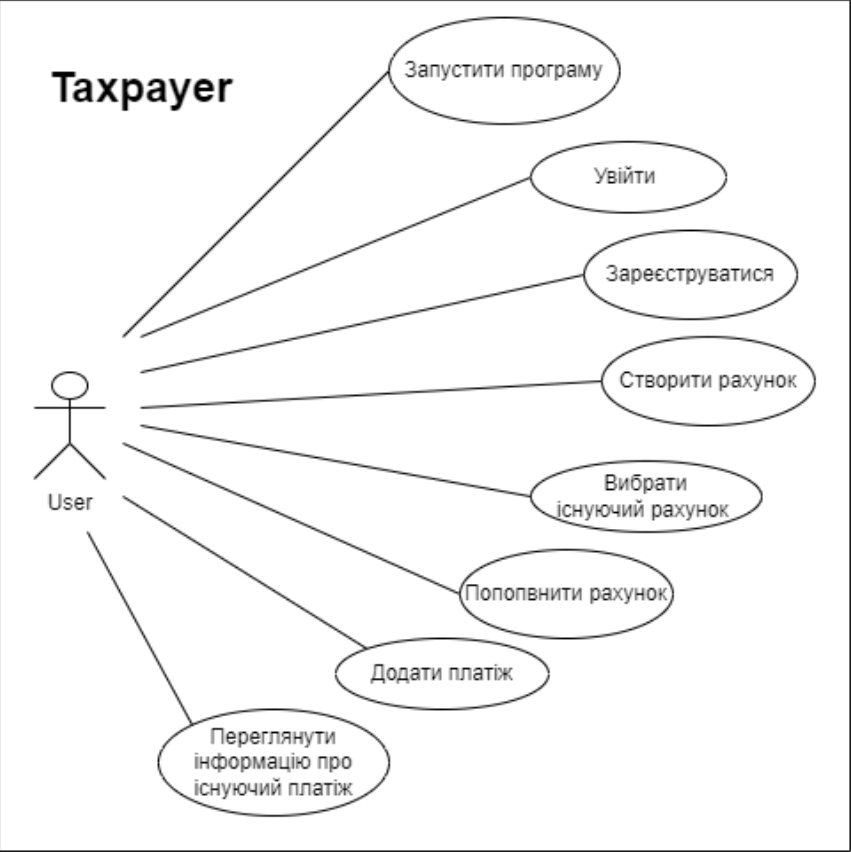


Рисунок 2.1 – Діаграма варіантів використання для програми Taxpayer

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

**Запустити програму.** Коли запускається програма, на екран повинно виводитися вікно, з якого користувач може вибрати потрібну дію. Це може називатися екраном інтерфейсу користувача.

**Увійти:** сценарій 1.Користувач вибирає опцію входу, вводить ім'я користувача та пароль, та увійшовши в систему, потрапляє в особистий кабінет.

**Зареєструватись та увійти:** сценарій 1.Користувач може зареєструватися як новий користувач, введе необхідні особисті дані, натисне кнопку зареєструватися обліковий запис і тоді здійснює вхід.

**Створити рахунок:** сценарій 1. У вкладці Account користувач може створити новий рахунок, вводячи ім'я

**Обрати існуючий рахунок:** сценарій 1. Користувач може вибрати раніше створений рахунок для подальшої роботи.

**Поповнити рахунок:** сценарій 1. Користувач обирає опцію "Поповнити рахунок" для внесення коштів на рахунок обраного об'єкта нерухомості. Вводить необхідну суму для поповнення рахунку. Система оновлює залишок на обраному рахунку та відображає оновлені дані.

**Додати платіж:** сценарій 1. Користувач може ввести інформацію про новий платіж, вибираючи вид послуги, суму, та інші необхідні дані.

**Переглянути інформацію про існуючий платіж:** сценарій 1.Користувач може переглядати інформацію про раніше внесені платежі, отримувати деталі та історію кожного платежу.

**Увійти:** сценарій 2.Користувач вибирає опцію входу, вводить неправильне ім'я користувача або пароль. Програма виводить повідомлення про помилку.

**Зареєструватись:**  сценарій 2. Користувач може зареєструватися як новий користувач, не введе необхідні особисті дані, і натисне кнопку зареєструватися. Тоді виведеться вікно помилки про те що треба ввести всі данні.

**Створити рахунок:** сценарій 2. У вкладці Account користувач може створити новий рахунок, вводячи ім'я. Якщо при створенні ім'я не введено, то виведеться вікно з помилкою.

**Поповнити рахунок:** сценарій 2. Користувач обирає опцію "Поповнити рахунок" для внесення коштів на рахунок обраного об'єкта нерухомості. Вводить необхідну суму для поповнення рахунку. Система оновлює залишок на обраному рахунку та відображає оновлені дані. Якщо при поповнені не введено коштів, виводиться повідомлення з помилкою.

**Додати платіж:** сценарій 2. Користувач може ввести інформацію про новий платіж, вибираючи вид послуги, суму, та інші необхідні дані. Якщо при оплаті не достатньо коштів виводиться повідомлення з помилкою. Або якщо не введено поле, то теж.

Також на етапі аналізу для моделювання варіантів використання, що мають декілька сценаріїв доцільно розробити UML-діаграми дій. Цей тип діаграм демонструє керівні потоки від одних дій до інших.

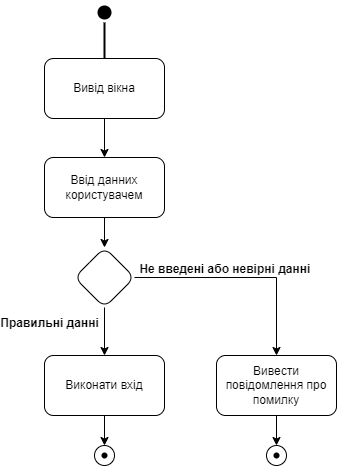


Рисунок 2.2 – Діаграма дій «Увійти»

На рисунку 2.2 показано варіант використання «Увійти», що включає обидва раніше описані сценарії. Вибір гілки діаграми залежить від того зайнята чи ні введена користувачем кімната. Якщо вона вже зайнята, то виводиться повідомлення про помилку. Також на рисунках 2.3, 2.4, 2.5, 2.6 зображено UMLдіаграму дій варіантів використання «Реєстрація», «Створення рахунку», «Поповнення рахунку», «Додати платіж».

.

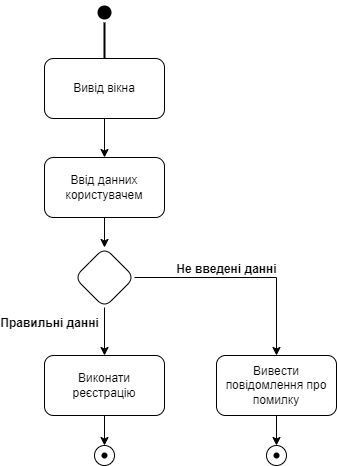


Рисунок 2.3 – Діаграма дій «Реєстрація»



Рисунок 2.4 – Діаграма дій «Створення рахунку»



Рисунок 2.5 – Діаграма дій «Поповнення рахунку»



Рисунок 2.6 – Діаграма дій «Додати платіж»

## Розробка системи класів

Етап проектування програмного забезпечення включає в себе логічне та фізичне проектування. Логічне проектування передбачає розробку структури класів, визначення полів для збереження складових об'єктів та алгоритмів методів, які описують поведінку цих об'єктів. Перш за все, необхідно визначити класи, які будуть входити до програми.

Для досягнення цього мета, ми можемо почати аналізуючи перелік іменників із опису варіантів використання. Після цього ми можемо внести деякі уточнення, визначити атрибути цих класів і перейти від дієслів до повідомлень. Зокрема, важливо врахувати основні операції, які будуть виконуватися над об'єктами.

Після цього ми можемо перейти до розробки діаграм класів та діаграм послідовностей. Діаграми класів відображають структуру класів і їх взаємодію, тоді як діаграми послідовностей дозволяють візуалізувати порядок виконання операцій між об'єктами.

Аналіз переліку іменників з опису варіантів використання. Розглянемо перелік усіх іменників, які візьмемо з опису варіантів використання і складемо перелік класів:

1. Вікно інтерфейсу користувача.
2. Вікно "Користувач"

3) Вкладка "Рахунки".

4) Вкладка "Платежі".

5) Користувач.

6) Рахунок.

7) Платіж.

Вкладка " Створити Рахунок".

Вкладка " Поповнити Рахунок".

Визначення атрибутів. Багато іменників, яким відмовлено в реєстрації як кандидатами у класи, будуть потенційними кандидатами в атрибути (компонентні дані) класів. Наприклад, в класі "User" будуть такі атрибути: ім'я , вік , логін. У класу "Рахунок": ім'я, баланс, сума. Більшість атрибутів можуть бути визначені так само.

Перехід від дієслів до повідомлень передбачає визначення методів класу, які обробляють конкретні повідомлення або виклики. Якщо розглядати описи варіантів використання, то можна здійснити цей перехід для кожного з них.

Розглянемо опис варіанта використання функції «Додати платіж»: «Програма виводить на екран 3 поля які потрібно заповнити: ім’я, сума і баланс, а також кнопка яку потрібно натиснути ».

Під словом «програма» ми насправді маємо на увазі вікно інтерфейсу користувача, отже, «Вікно інтерфейсу користувача» в момент натискання кнопки спрацьовує слот який записує данні з полів в об’єкт повідомлення об'єкту «Payment» (тобто викликає його метод).

Такий аналіз можна провести для всіх варіантів використання, щоб визначити методи кожного класу і їхні взаємодії. Це створить основу для подальшого фізичного проектування системи.

**Побудова діаграм класів**. Знаючи, які класи буде включено в розроблювану програму і як вони пов'язані між собою, можна побудувати спрощену діаграму класів, яка зображує взаємозв'язки між ними. При цьому важливо зазначити, що дану діаграму можна знайти на рисунку 2.7

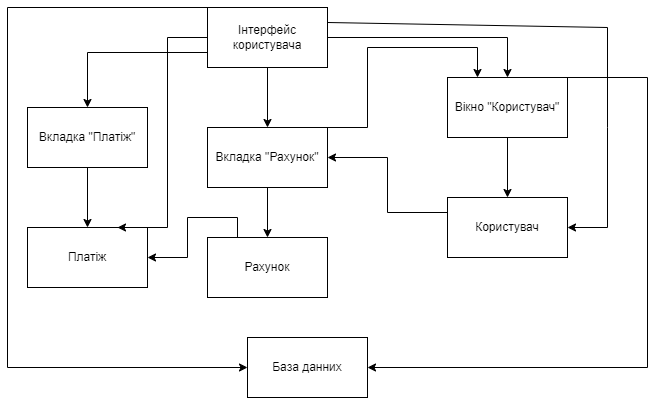


Рисунок 2.7 – Діаграма класів для програми Taxpayer

## Розробка методів

В даному етапі розробки описується еволюція програмного забезпечення, яка включає послідовну реалізацію та інтеграцію класів у проект, використовуючи бібліотеку Qt та її віджети. Починаючи зі створення основного інтерфейсу проекту та бібліотеки класів, розробник визначає типи полів класів, розробляє алгоритми методів, уточнює вимоги до програми та вносить необхідні зміни. Прототип продукту тестується та налагоджується.

У програмі використовуються віджети Qt для організації роботи з віджетами та записами таблиць програми. Деякі класи, такі як User, Account та Payment, містять віджети Qt для управління відображенням та введенням даних.

Звертаючись до взаємозв'язків між класами за допомогою віджетів Qt, були створені декілька допоміжних класів, таких як MainWindow, і UserWindow.

Атрибути класів включають віджети Qt та покажчики на об'єкти інших класів для уникнення зайвого копіювання даних у пам'яті. Наприклад, у класі Payment можуть з'явитися атрибути, такі як QString description і QDate date.

До розгортання програми було створено повну діаграму класів за допомогою програми StarUML, яка відображена на рисунку 2.8

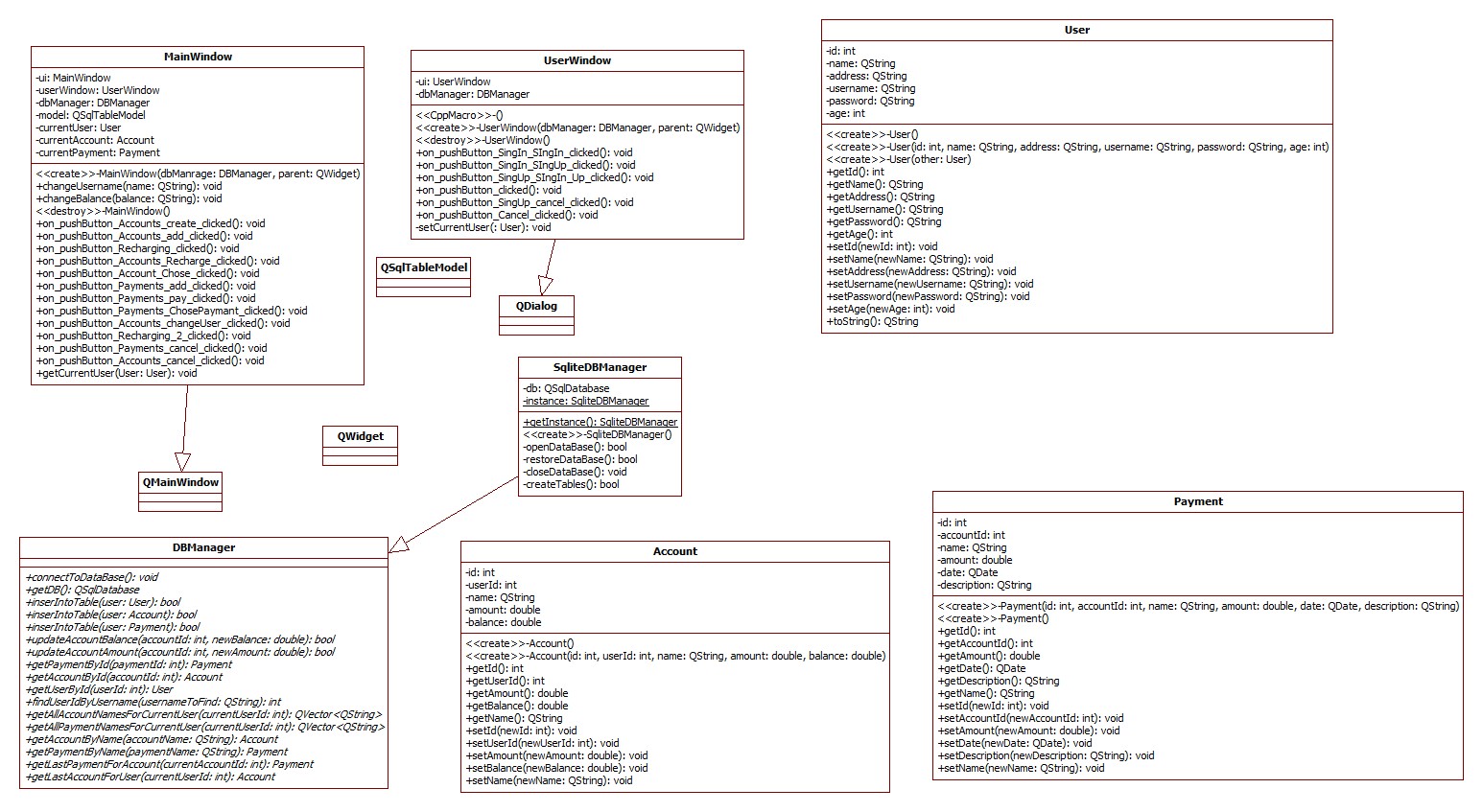


Рисунок 2.8 – UML-діаграма класів програми Taxpayer

У заголовному файлі також додано оголошення допоміжних класів та визначено інтерфейси класів для використання у Qt-віджетах.

Написання коду програми розпочинається з оголошення заголовного (\*.h) файлу, в якому визначаються інтерфейси класів, а реалізація методів розробляється у \*.cpp файлах. Перед написанням коду важливо визначити імена класів та методів, їхнє призначення та передбачити аргументи, що передаються.

Написання коду програми розглянутих методів має починатися тільки на даному етапі розроблення ПЗ і ні кроком раніше, тому що тільки зараз відомі ім'я кожного класу чи методу, їх призначення і навіть, можливо, передбачити аргументи, що їм передаються.

## Проєктування і опис інтерфейсу користувача

Проєктування і опис інтерфейсу користувача включає в себе створення та опис елементів інтерфейсу, які взаємодіють з користувачем. У цьому випадку, основною метою є забезпечення зручного та ефективного спілкування користувача з програмою.

Першим вікном є MainWindow, що включає в себе lable для відображення імені користувача та балансу. Також в MainWindow є TabWiget який має три вкладки: About the Program, Accounts, Payments. Варто зазначити що MainWindow зібраний в основному з таких віджетів, як layout, lineEdit, TextEdit, ComboBox, Vertical і Horizontal spacer

У вкладці About the Program просто зображено відомості про програму. Дану вкладку можна знайти на рисунку 2.9

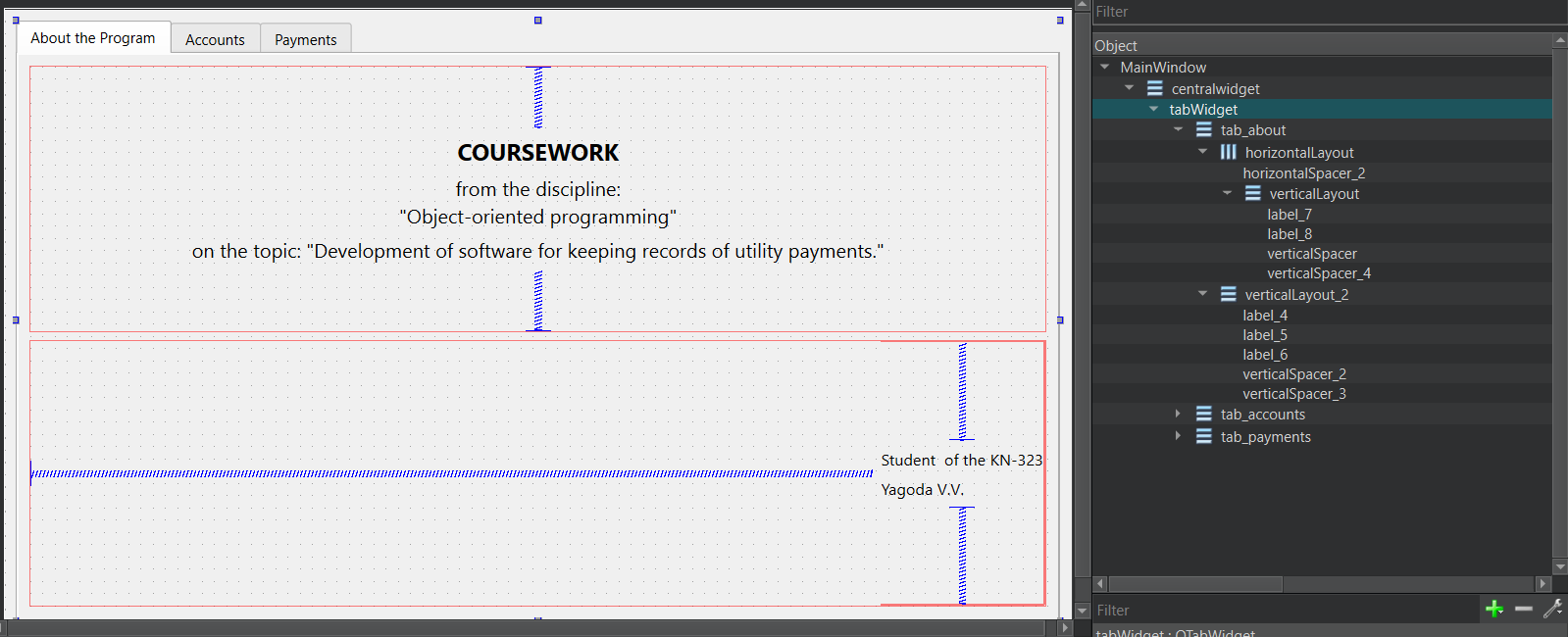


Рисунок 2.9 – Вкладка «About the program» в програмі Taxpayer

Наступною йде вкладка "Accounts" яка має в собі TabWiget з трьома вкладкми: "Створення рахунку", "Інформація про рахунок" і "поповнення рахунку". Їх можна побачити на рисунках 2.10 – 2.12.

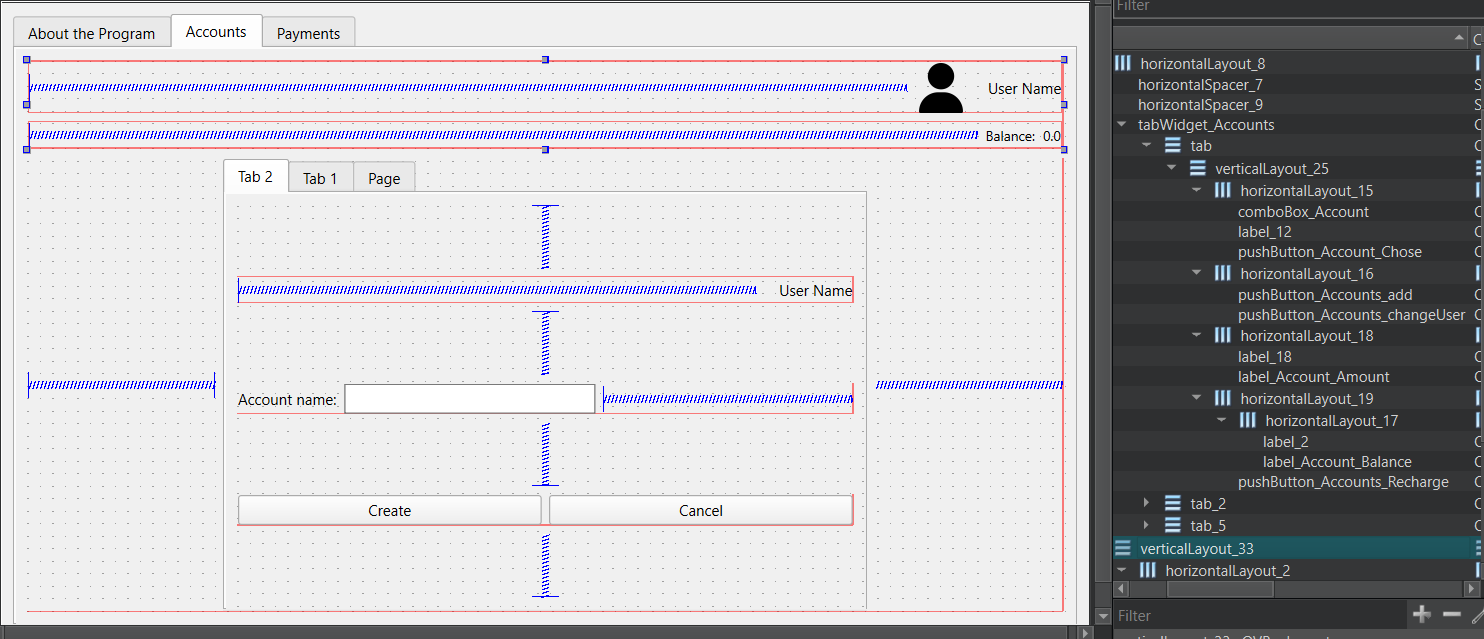


Рисунок 2.10 – Вкладка «Accounts» вкладка "Створення рахунку"

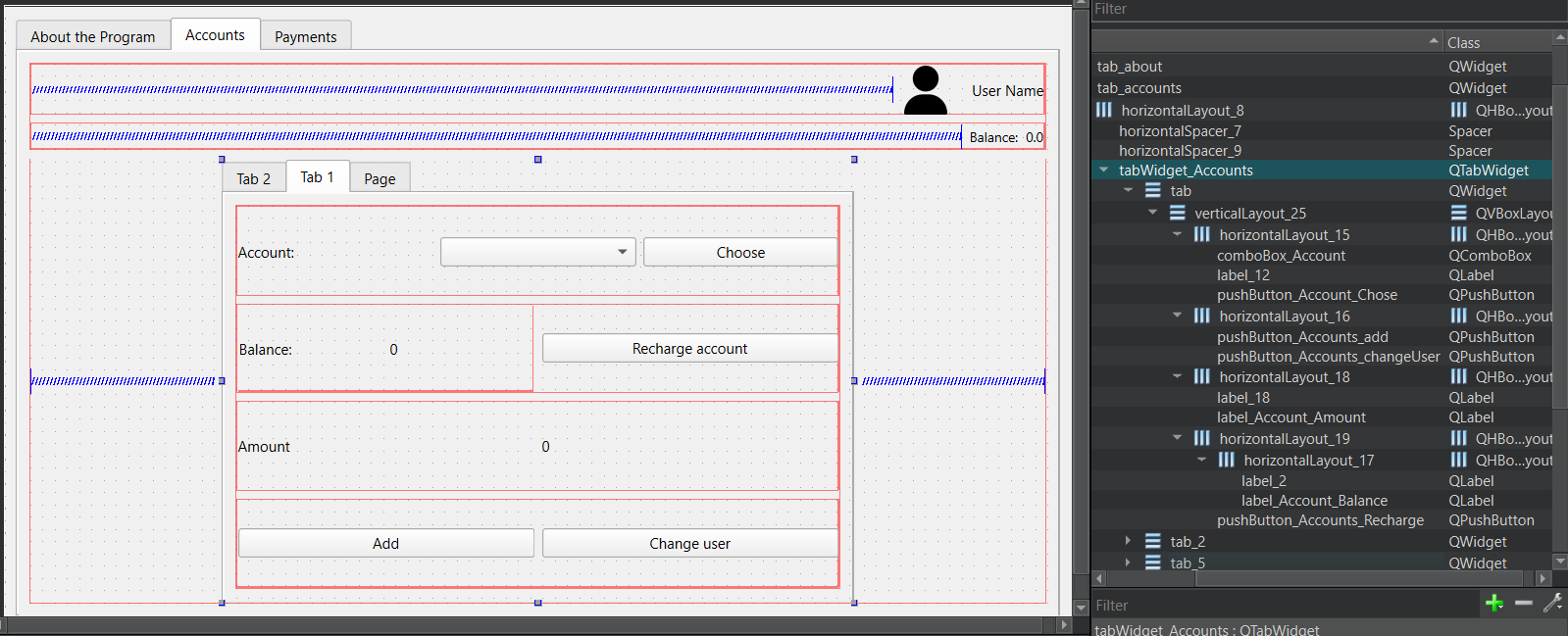


Рисунок 2.11 – Вкладка «Accounts» вкладка "Створення рахунку"

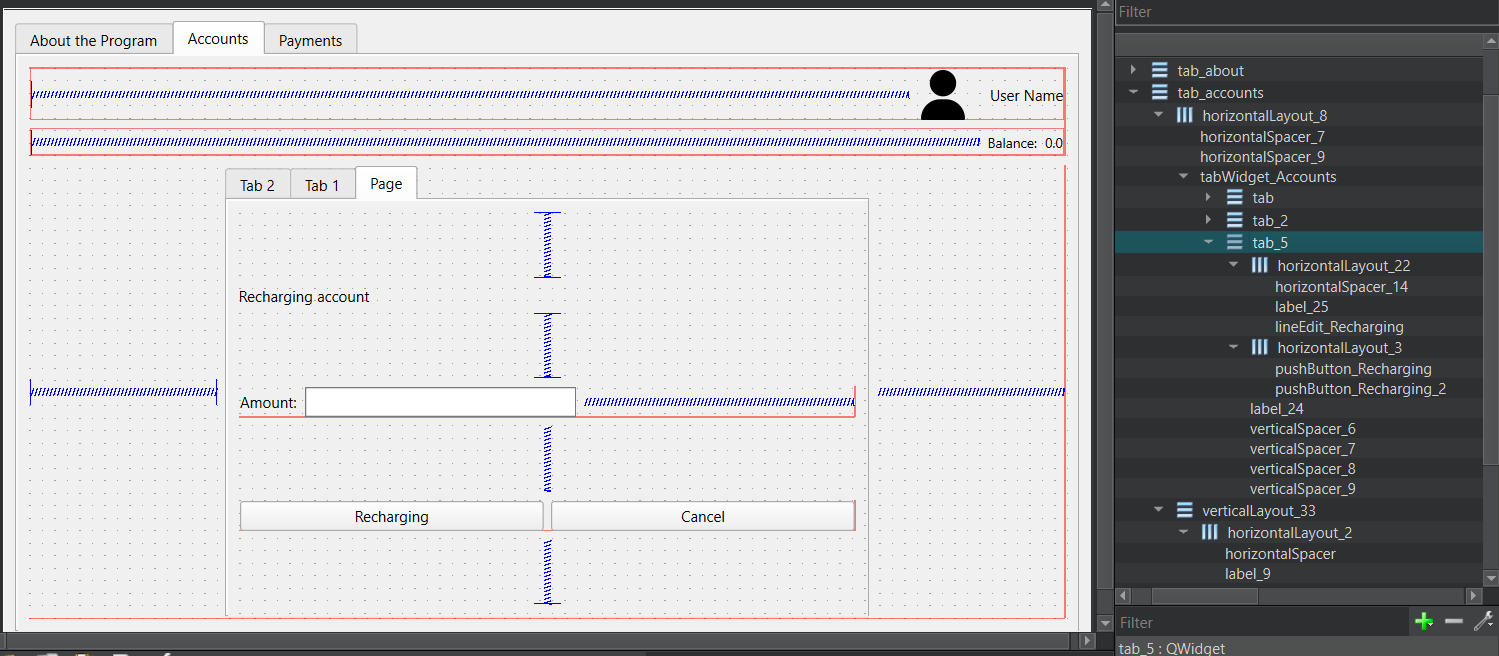


Рисунок 2.12 – Вкладка «Accounts» вкладка "Створення рахунку"

Наступною після вкладки "Accounts" є "Payments", яка має в собі TabWiget з двома вкладкми: "Внесення платежу" і "Перегляд платежу". Їх можна побачити на рисунках 2.13 – 2.14.

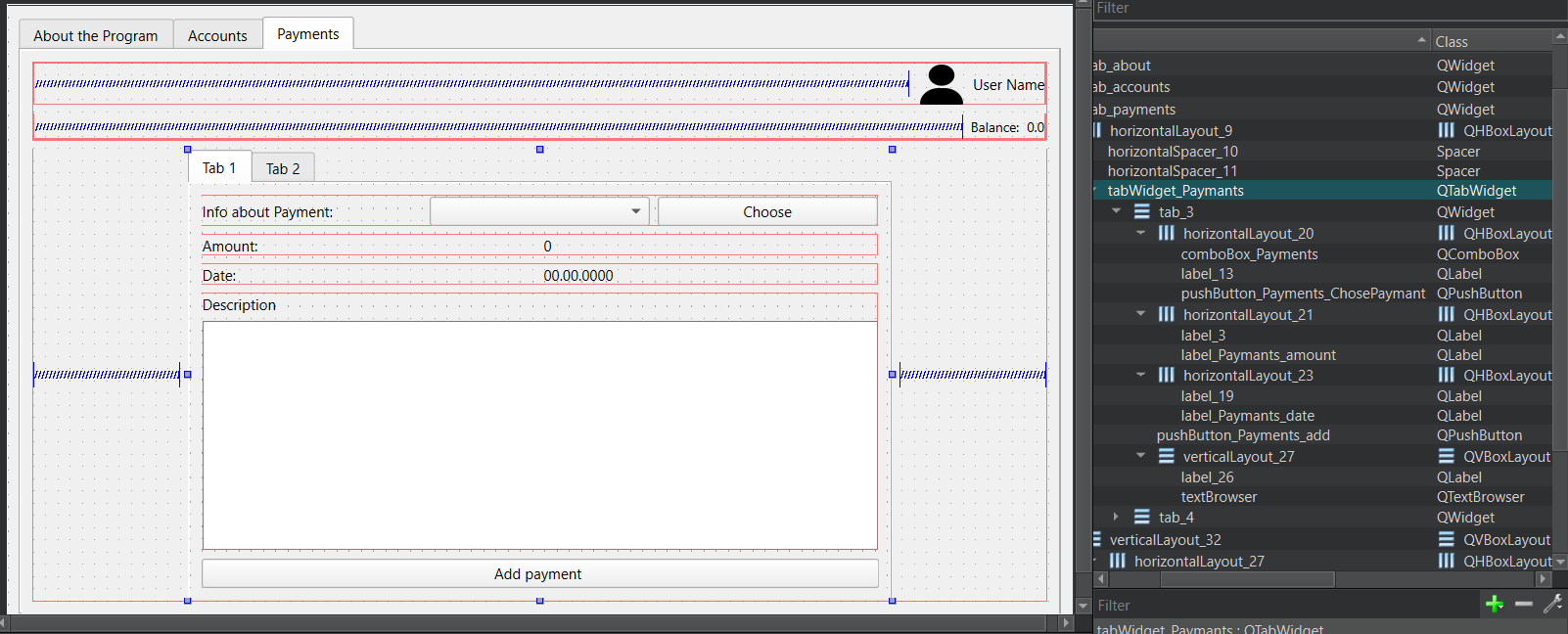


Рисунок 2.13 – Вкладка «Payments» вкладка " Внесення платежу "

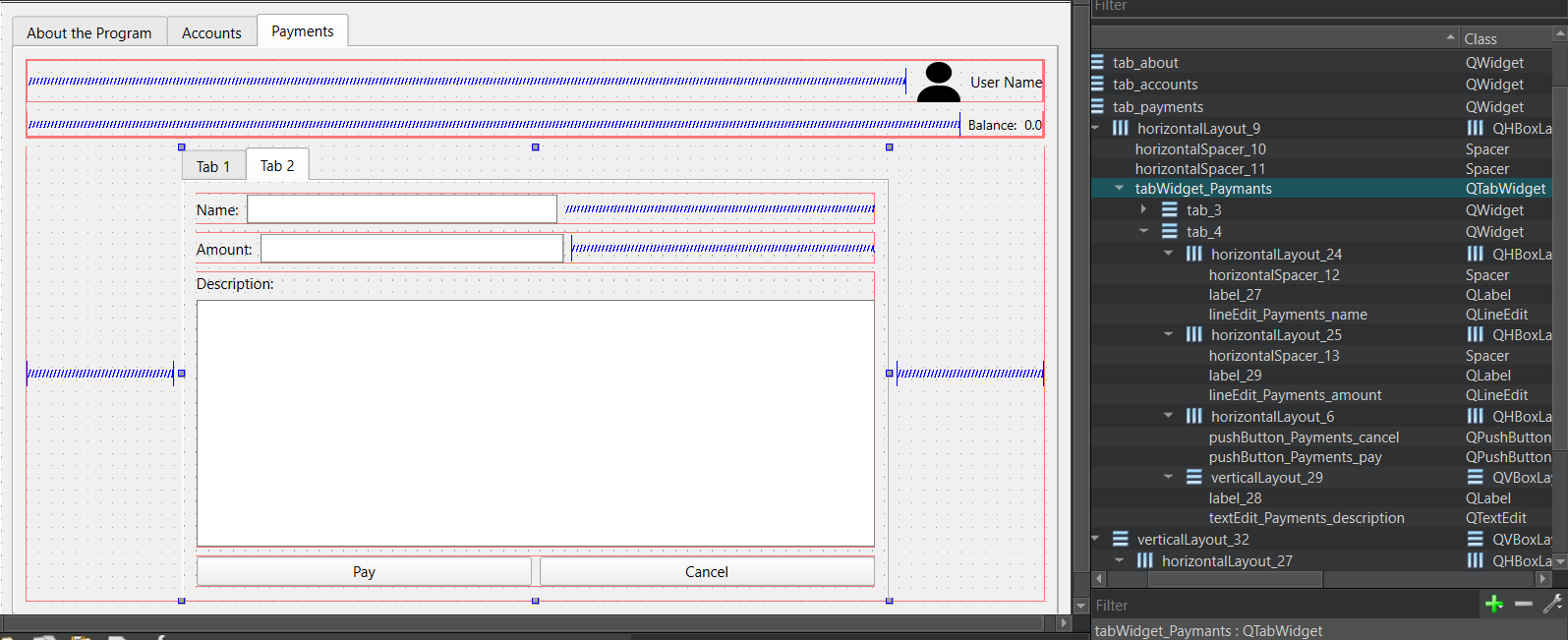


Рисунок 2.14 – Вкладка «Payments» вкладка " Перегляд платежу "

Другим вікном є UserWindow, тут також є TabWiget який має три вкладки: Sing in, Sing up, display info. Sing in – призначена для входу, Sing up –для реєстрації, а display info - для відображення інформації Їх можна побачити на рисунках 2.15 – 2.18

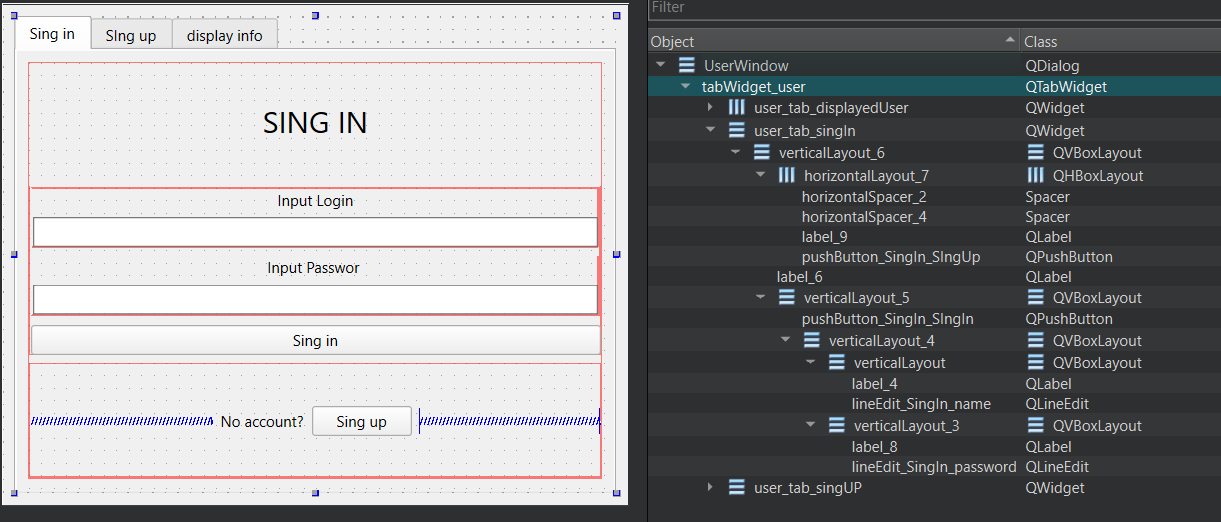


Рисунок 2.15 – Вкладка «Sing in» в вікні UserWindow

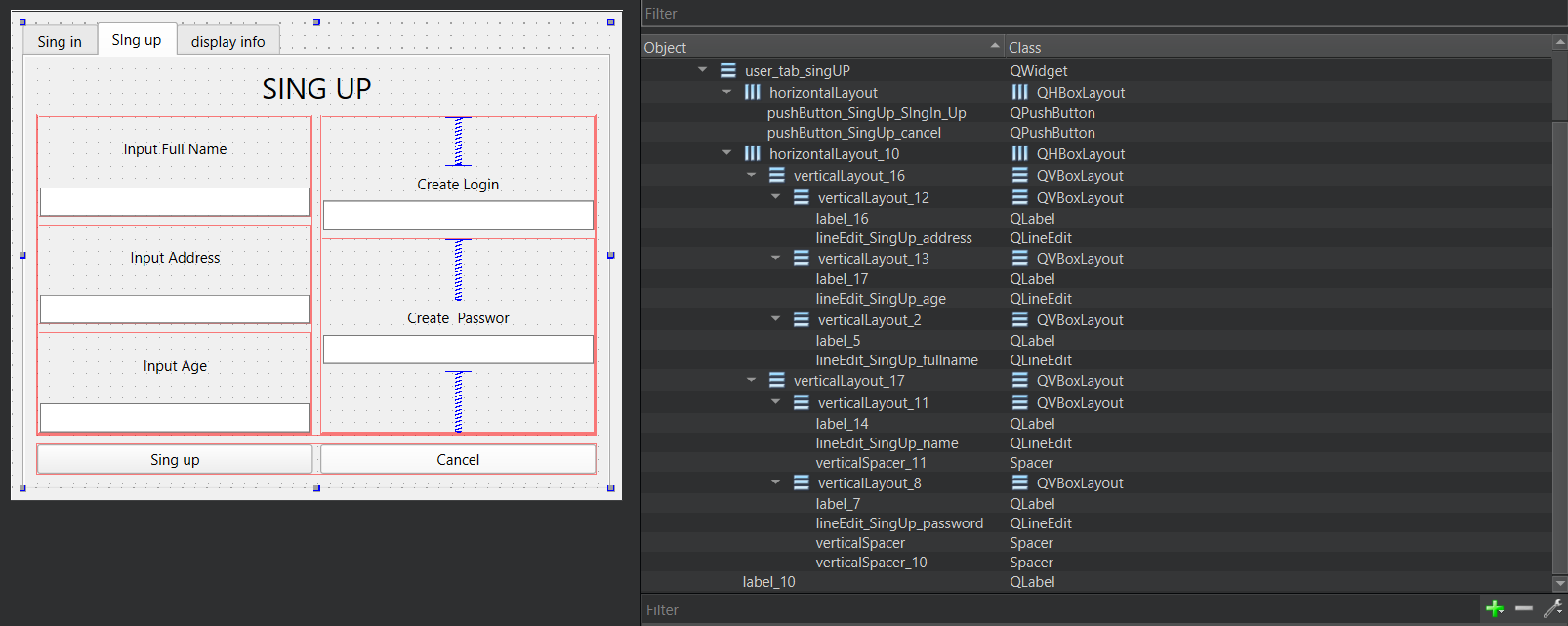


Рисунок 2.16 – Вкладка «Sing up» в вікні UserWindow

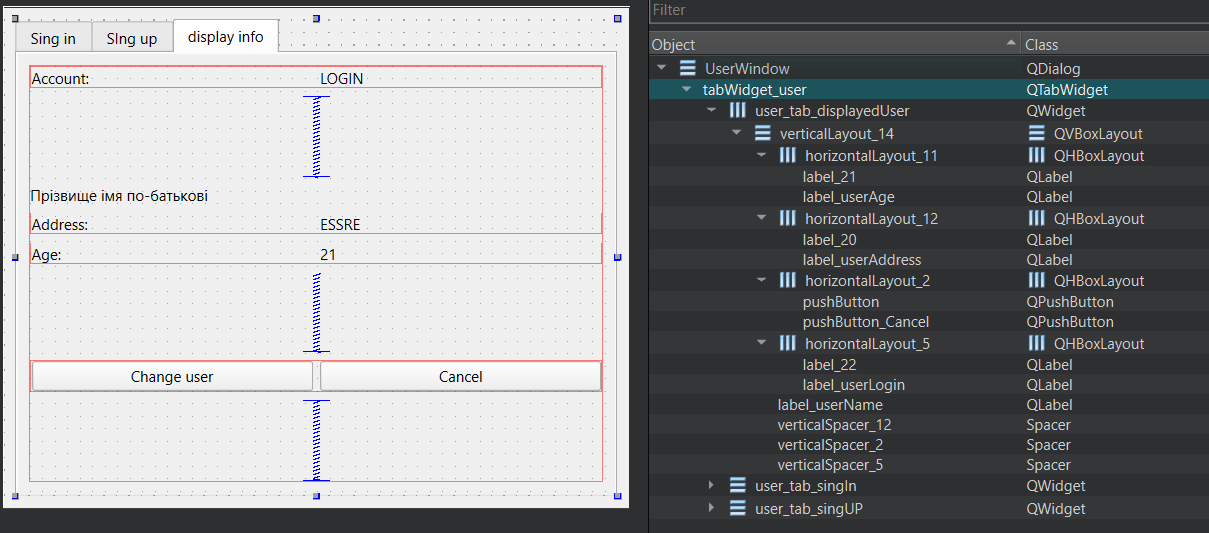


Рисунок 2.16 – Вкладка «Display info» в вікні UserWindow

## Опис файлової структури програми

Збірка програми зберігається в файлі«coursework.pro» .Лістинг даного файлу представлено у додатку А. Головна функція програми main() зберігаємо в одному коротенькому файлі «Main.cpp». Лістинг даного файлу представлено у додатку Б. У функції main() створюється об'єкт MainWindow лісинги класу цього об'єкта представлено у додатках В і Г відповідно. У MainWindow створюється об'є UserWindow лісинги класу цього об'єкта представлено у додатках Ґ і Е відповідно

У вихідному файлі «User.cpp» містяться тіла методів і функцій, які були оголошені у заголовному файлі «User.h». Лістинг даних файлів представлено у додатках Є і Ж відповідно.У вихідному файлі «Account.cpp» містяться тіла методів і функцій, які були оголошені у заголовному файлі «Account.h». Лістинг даних файлів представлено у додатках З і І відповідно.У вихідному файлі «Payment.cpp» містяться тіла методів і функцій, які були оголошені у заголовному файлі «Payment.h». Лістинг даних файлів представлено у додатках Ї і Й відповідно.

У функції main() створюється об'єкт dbManager клас якого описаний в додатку К. А його дочірній клас описаний в додатках Л і M.

## Опис структури бази даних програми

Система обліку фінансів має базу даних, яка включає три основні таблиці: User (Користувач), Account (Рахунок) та Payment (Платіж). Кожна з цих таблиць має свою унікальну структуру та взаємозв'язки для забезпечення ефективності та цілісності даних.

Таблиця User (Користувач):

id (INTEGER): Унікальний ідентифікатор користувача.

name (VARCHAR(255)): Ім'я користувача.

address (VARCHAR(255)): Адреса користувача.

username (VARCHAR(255)): Логін користувача (унікальний).

password (VARCHAR(255)): Пароль користувача.

age (INTEGER): Вік користувача.

Таблиця Account (Рахунок):

id (INTEGER): Унікальний ідентифікатор рахунку.

name (VARCHAR(255)): Назва рахунку.

amount (DECIMAL(10, 2)): Початкова сума на рахунку.

balance (DECIMAL(10, 2)): Поточний баланс рахунку.

user\_id (INTEGER): Зовнішній ключ, що посилається на id у таблиці User.

Таблиця Payment (Платіж):

id (INTEGER): Унікальний ідентифікатор платежу.

name (VARCHAR(255)): Назва платежу.

amount (DECIMAL(10, 2)): Сума платежу.

date (DATE): Дата платежу.

description (VARCHAR(255)): Опис платежу.

account\_id (INTEGER): Зовнішній ключ, що посилається на id у таблиці Account.

Кожна з цих таблиць має відповідність у вигляді зовнішніх ключів, що забезпечують зв'язок між ними. Наприклад, кожний рахунок (Account) пов'язаний з конкретним користувачем (User), використовуючи поле user\_id.

# Тестування програми і результати її виконання

В ході тестування програми було виявлено і виправлено незначні помилки.

Запустивши програму на екрані монітора відображається вкладка About The Program рисунок 3.1

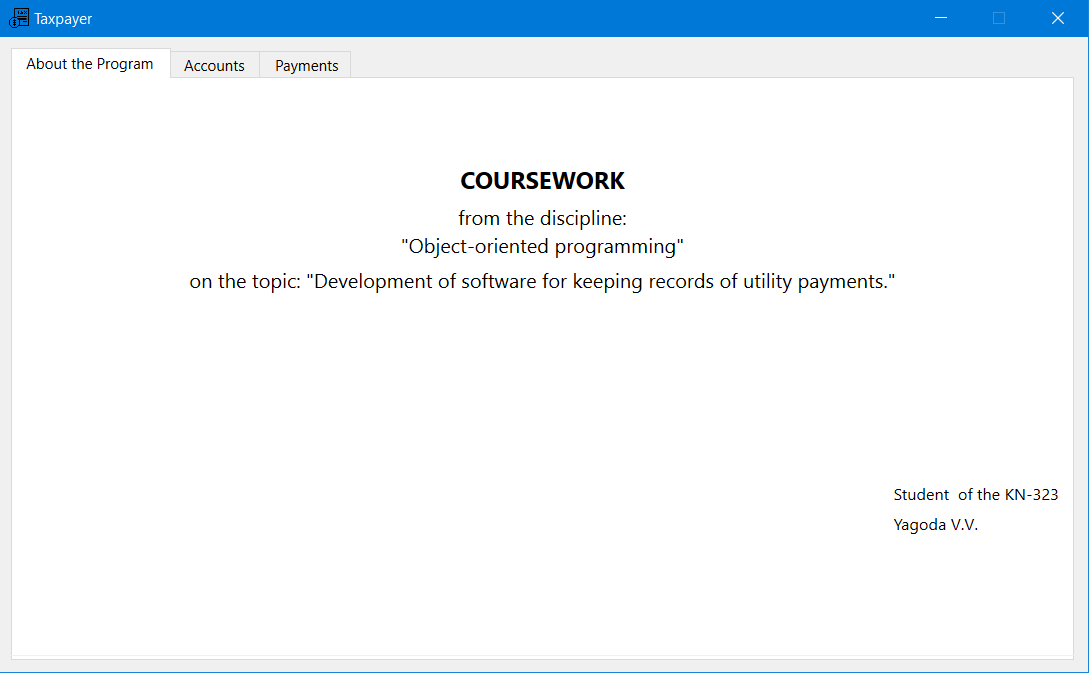


Рисунок 3.1 – вкладка About The Program

Щоб почати взаємодію з програмою потрібно перейти у вкладку Account (рисунок 3.2) і натиснути кнопку "Change User"

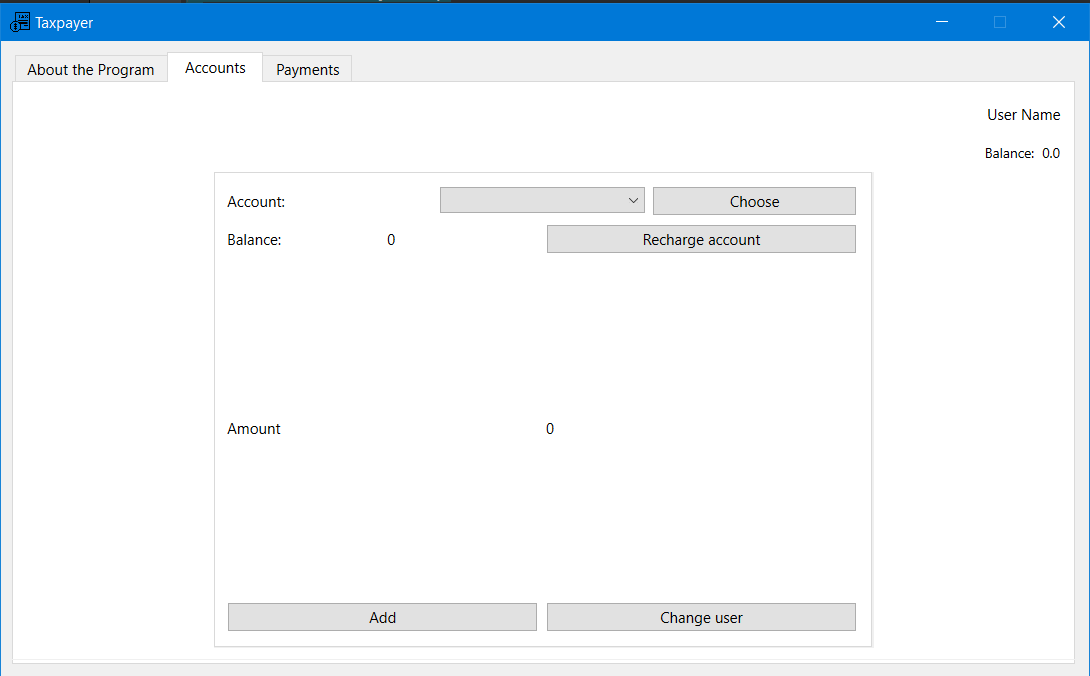


Рисунок 3.2 – вкладка Accounts

Після відкривання вікна користувача потрібно ввести пароль і логін, і натиснути кнопку "Sing in" (рисунок 3.3). Якщо профіль ще не створений, його потрібно створити, натиснувши кнопку "Sing up", тоді потрібно в вікні створення ввести всі данні і натиснути кнопку "Sing up" (рисунок 3.4).

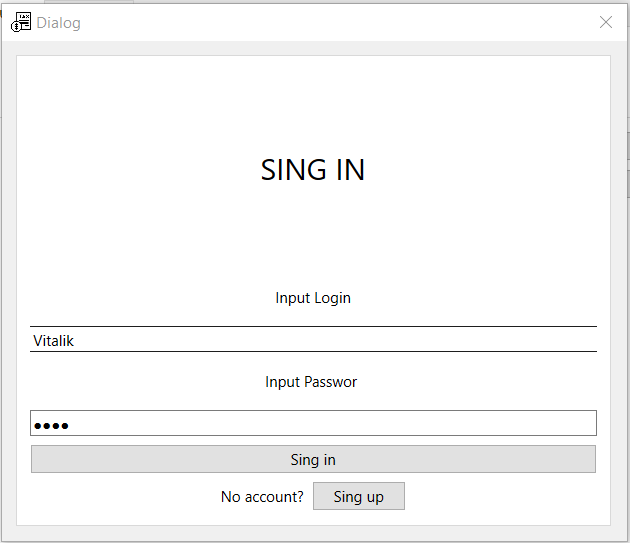


Рисунок 3.3 – вкладка Sing in вікно User

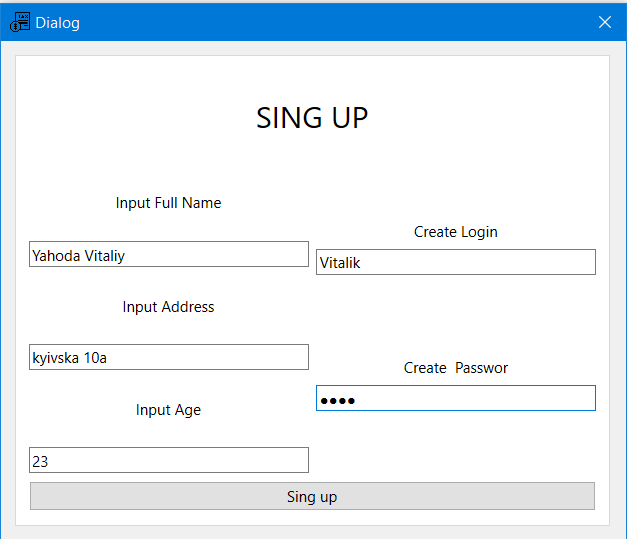


Рисунок 3.4 – вкладка Sing up вікно User

Після закриття вікна User можна починати основну роботу з програмою. Для подальшої роботи на вкладці Accounts потрібно вибрати рахунок з існуючих(рисунок 3.5), вибравши назву в вибачаючому списку і натиснувши кнопку "Choose" або cтворити новий(рисунок 3.6),натиснувши кнопку "Add", після чого вибрати його.

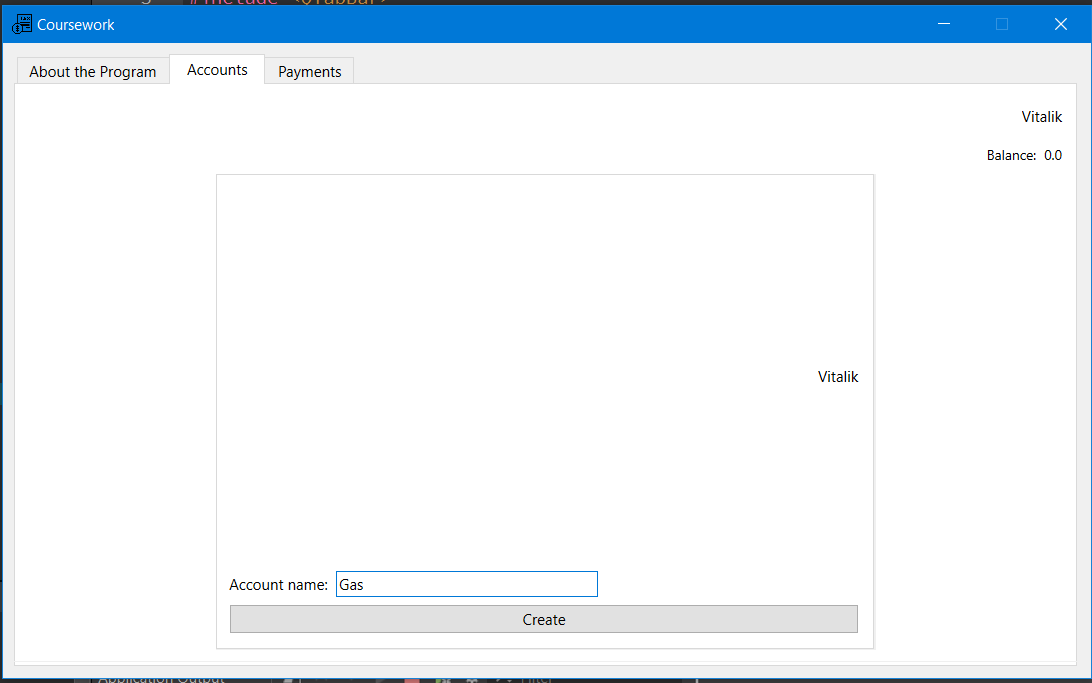


Рисунок 3.5 – вкладка Accounts створення рахунку

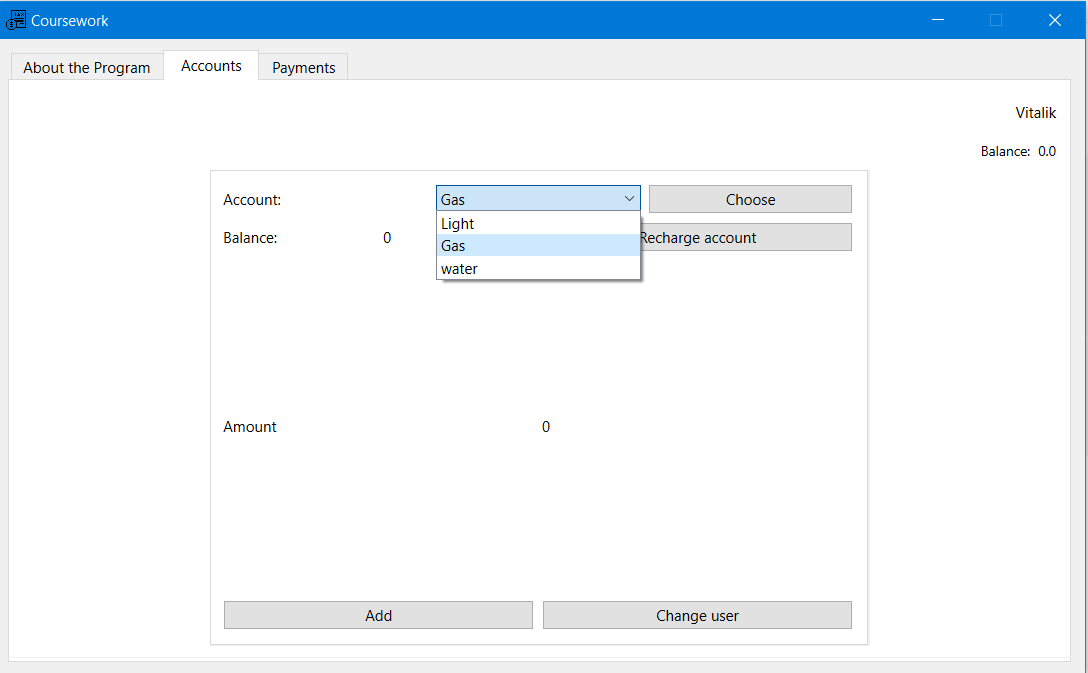


Рисунок 3.6 – вкладка Accounts вибір рахунку

Далі потрібно поповнити рахунок натиснувши на кнопку "Recharge account" (рисунок 3.7).

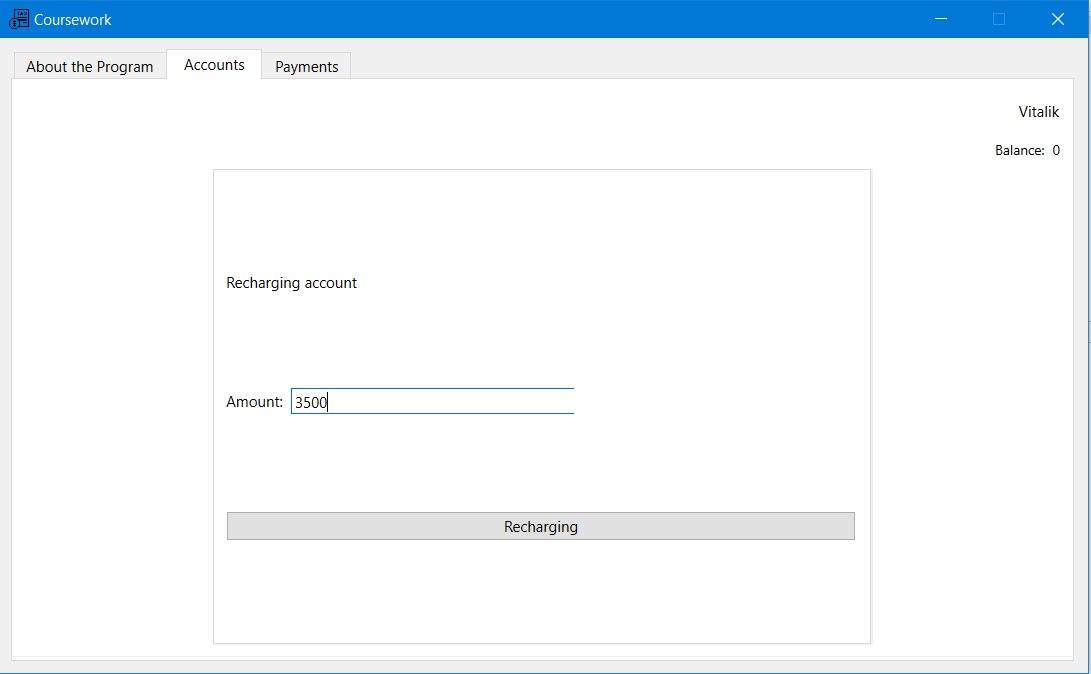


Рисунок 3.7 – вкладка Accounts поповнення рахунку

Після поповнення можна переходити в вкладку Payment та здійснювати платежі, щоб здійснити платіж потрібно натиснути кнопку "Add payment", після чого потрібно ввести данні про платіж і натиснути кнопку "Pay" (рисунок 3.8). Також можна переглянути інформацію про вже зроблений платіж, вибравши назву в вибачаючому списку і натиснувши кнопку "Choose"(рисунок 3.9).

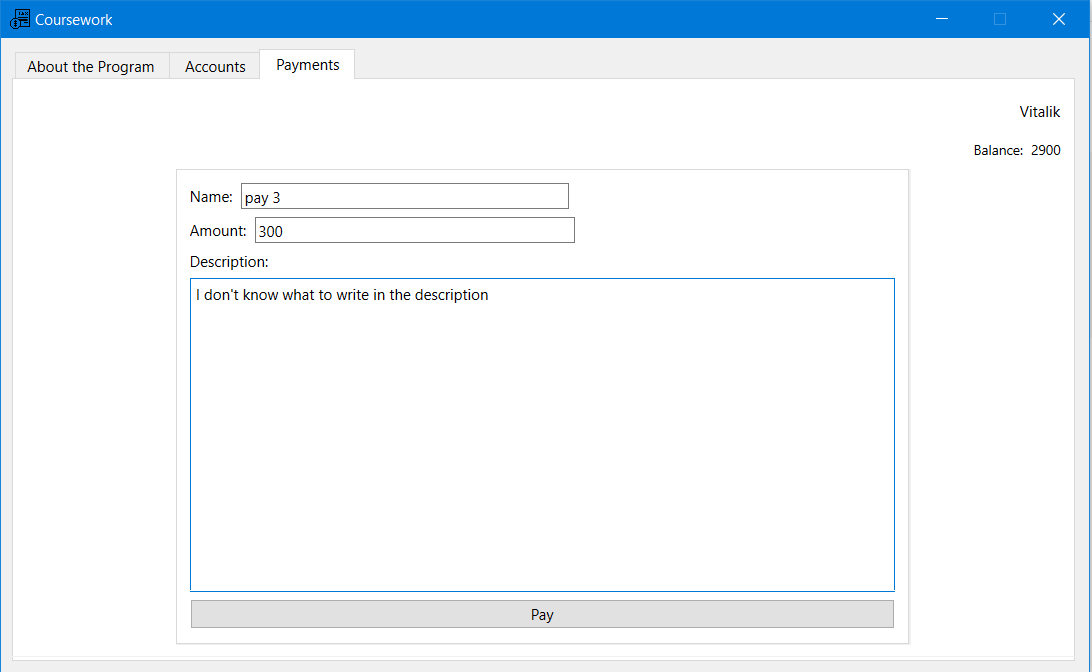


Рисунок 3.8 – вкладка Payment поповнення додавання платежу

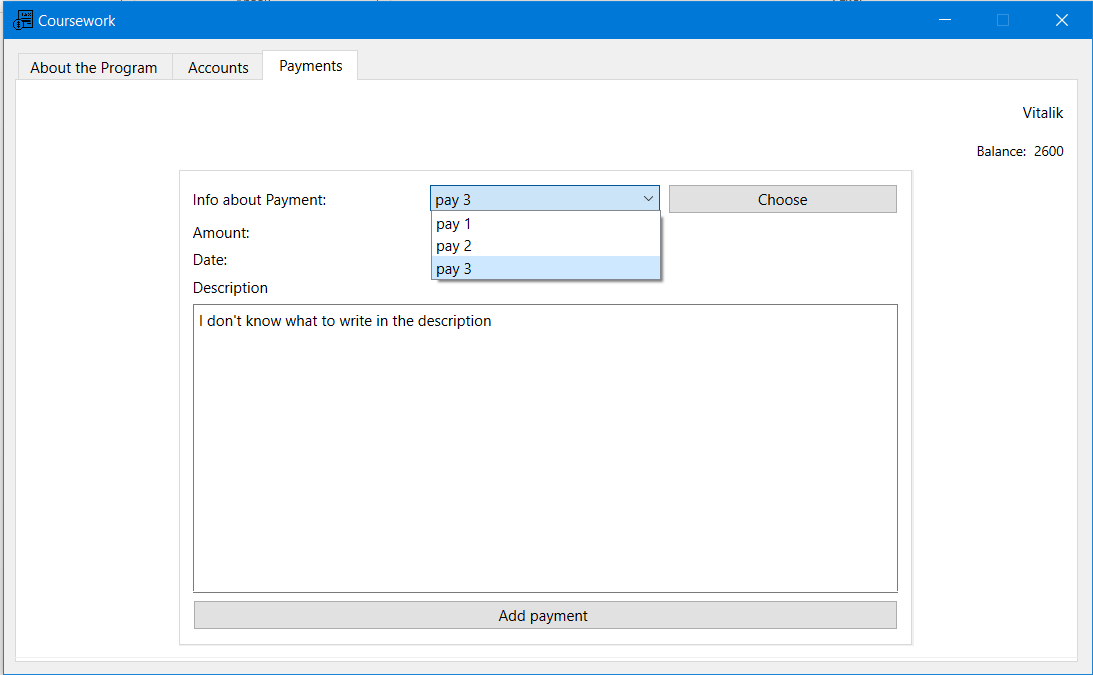


Рисунок 3.9 – вкладка Payment перегляд інформації платежу

Висновки

У рамках цієї курсової роботи було ретельно розглянуте проектування та розробка програмного продукту Taxpayer, спрямованого на облік прибутково-видаткових операцій домовласника багатоквартирного будинку. Розроблене програмне забезпечення повністю відповідає технічному завданню, визначаючи необхідність для ефективного ведення фінансового обліку та аналізу операцій.

У процесі виконання курсової роботи були закріплені та розширені знання, набуті під час вивчення курсу "Об'єктно-орієнтоване програмування", а також глибше освоєна мова програмування C++. Використання бібліотеки Qt та інструментів Qt Widgets, сприяло швидкій та ефективній розробці програмного засобу Taxpayer.

Під час розробки програми в основу були покладені принципи та концепції об'єктно-орієнтованого програмування, що дозволило створити зручний та легко розширюваний інструмент для ведення обліку. Використання стандартної бібліотеки Qt, зокрема векторів та множин, сприяло значному прискоренню розробки програми.

Отже, розробка програми Taxpayer є результативним застосуванням отриманих знань з об'єктно-орієнтованого програмування та сучасних технологій розробки програм, зокрема фреймворку Qt.

Перелік посилань

1) Слободян Р.О. Методичні вказівки до виконання курсової роботи з дисципліни «Об’єктно-орієнтоване програмування» для студентів спеціальності 122 «Комп’ютерні науки». Тернопіль : ТК ТНТУ ім. І.Пулюя, 2019. 45с

2) Вайсфельд М. Объектно-ориентированное мышление. СПб. : Питер, 2014. 304 с.

Додаток А  
Лістинг файлу «myprogram.pro»

QT += core gui

QT += sql

greaterThan(QT\_MAJOR\_VERSION, 4): QT += widgets

CONFIG += c++17

# You can make your code fail to compile if it uses deprecated APIs.

# In order to do so, uncomment the following line.

#DEFINES += QT\_DISABLE\_DEPRECATED\_BEFORE=0x060000 # disables all the APIs deprecated before Qt 6.0.0

SOURCES += \

account.cpp \

main.cpp \

mainwindow.cpp \

payment.cpp \

sqlitedbmanager.cpp \

user.cpp \

userwindow.cpp

HEADERS += \

QDialog.h \

QMainWindow.h \

QWidget.h \

account.h \

dbmanager.h \

mainwindow.h \

payment.h \

sqlitedbmanager.h \

user.h \

userwindow.h

FORMS += \

mainwindow.ui \

userwindow.ui

# Default rules for deployment.

qnx: target.path = /tmp/$${TARGET}/bin

else: unix:!android: target.path = /opt/$${TARGET}/bin

!isEmpty(target.path): INSTALLS += target

RC\_ICONS = $$PWD/Image/mainIcon.ico

#Share all project output files by directories

MOC\_DIR = moc

RCC\_DIR = rcc

UI\_DIR = ui

unix:OBJECTS\_DIR = unix

win32:OBJECTS\_DIR = win32

macx:OBJECTS\_DIR = mac

#If release-buid -> run windeploy applications, that will collect all the dlls

CONFIG(release, debug|release) {

win32:QMAKE\_POST\_LINK = $$(QTDIR)/bin/windeployqt $$OUT\_PWD/release

}

Додаток Б  
Лістинг файлу «main.cpp»

#include "mainwindow.h"

#include <QApplication>

#include "sqlitedbmanager.h"

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

{

QApplication a(argc, argv);

DBManager\* dbManager = SqliteDBManager::getInstance();

dbManager->connectToDataBase();

MainWindow w(dbManager);

w.show();

return a.exec();

}

Додаток В  
Лістинг файлу «mainwindow.h»

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QWidget>

#include <QMainWindow>

#include <QDateTime>

#include <QSqlTableModel>

#include <QSqlQuery>

#include "user.h"

#include "dbmanager.h"

#include "sqlitedbmanager.h"

#include "userwindow.h"

#include <QDebug>

class QSqlTableModel;

class DBManager;

QT\_BEGIN\_NAMESPACE

namespace Ui { class MainWindow; }

QT\_END\_NAMESPACE

class MainWindow : public QMainWindow

{

Q\_OBJECT

public:

explicit MainWindow(DBManager\* dbManrage, QWidget \*parent = nullptr);

void changeUsername(QString name);

void changeBalance(QString balance);

~MainWindow();

private slots:

void on\_pushButton\_Accounts\_create\_clicked();

void on\_pushButton\_Accounts\_add\_clicked();

void on\_pushButton\_Recharging\_clicked();

void on\_pushButton\_Accounts\_Recharge\_clicked();

void on\_pushButton\_Account\_Chose\_clicked();

void on\_pushButton\_Payments\_add\_clicked();

void on\_pushButton\_Payments\_pay\_clicked();

void on\_pushButton\_Payments\_ChosePaymant\_clicked();

void on\_pushButton\_Accounts\_changeUser\_clicked();

void on\_pushButton\_Recharging\_2\_clicked();

void on\_pushButton\_Payments\_cancel\_clicked();

void on\_pushButton\_Accounts\_cancel\_clicked();

public slots:

void getCurrentUser(User \*User);

private:

Ui::MainWindow \*ui;

UserWindow\* userWindow;

DBManager\* dbManager;

QSqlTableModel\* model;

User\* currentUser = nullptr;

Account\* currentAccount = nullptr;

Payment\* currentPayment = nullptr;};

Додаток Г

**Лістинг файлу «mainwindow.cpp»**

#include "mainwindow.h"

#include "ui\_mainwindow.h"

#include <QTabBar>

#include <QMessageBox>

MainWindow::MainWindow(DBManager\* dbManager,QWidget \*parent)

: QMainWindow(parent)

, ui(new Ui::MainWindow), dbManager(dbManager)

{

ui->setupUi(this);

userWindow = new UserWindow(this->dbManager,this);

userWindow ->setModal(false);

connect(userWindow, &UserWindow::setCurrentUser, this, &MainWindow::getCurrentUser);

ui->tabWidget->setCurrentIndex(0);

ui->tabWidget\_Accounts->tabBar()->hide();

ui->tabWidget\_Accounts->setCurrentIndex(1);

ui->tabWidget\_Paymants->tabBar()->hide();

ui->tabWidget\_Paymants->setCurrentIndex(0);

}

MainWindow::~MainWindow()

{

delete ui;

delete userWindow;

if (currentUser)

delete currentUser;

if (currentAccount)

delete currentAccount;

if (currentPayment)

delete currentPayment;

if (model)

delete model;

}

//User

void MainWindow::getCurrentUser(User \*user){

MainWindow::changeUsername(user->getUsername());

currentUser = new User(\*user);

ui->comboBox\_Account->clear();

QVector<QString> accountNames = dbManager->getAllAccountNamesForCurrentUser(currentUser->getId());

for (const QString& name : accountNames) {

ui->comboBox\_Account->addItem(name);

}

}

void MainWindow::on\_pushButton\_Accounts\_create\_clicked()

{

if(currentUser == nullptr){

QMessageBox::information(nullptr, "Current user exist", "Change user");

}

else{

if ( ui->lineEdit\_Account\_name->text().isEmpty())

{

QMessageBox::information(nullptr, "Empty field", "Enter all fields");

}

else{

Account\* account = new Account(1,currentUser->getId(),ui->lineEdit\_Account\_name->text(),0,0);

ui->comboBox\_Account->addItem(account->getName());

if (!dbManager->inserIntoTable(\*account)) {

qDebug() << "Error inserting data into the database.";}

ui->tabWidget\_Accounts->setCurrentIndex(1);

currentAccount = new Account(\*dbManager->getLastAccountForUser(currentUser->getId()));

}

}

}

void MainWindow::on\_pushButton\_Accounts\_add\_clicked()

{

ui->tabWidget\_Accounts->setCurrentIndex(0);

}

void MainWindow::on\_pushButton\_Recharging\_clicked()

{ if(currentUser == nullptr){

QMessageBox::information(nullptr, "Current user exist", "Change user");

}

else{

ui->tabWidget\_Accounts->setCurrentIndex(1);

currentAccount->setBalance(currentAccount->getBalance() + ui->lineEdit\_Recharging->text().toDouble());

ui->label\_Account\_Balance->setText(QString::number( currentAccount->getBalance() ));

if( !dbManager->updateAccountBalance(currentAccount->getId(), currentAccount->getBalance())){

qDebug() << "Error updating data into the database.";}

MainWindow::changeBalance(QString::number( currentAccount->getBalance() ));

}

}

void MainWindow::on\_pushButton\_Accounts\_Recharge\_clicked()

{

ui->tabWidget\_Accounts->setCurrentIndex(2);

}

void MainWindow::on\_pushButton\_Account\_Chose\_clicked()

{

if(currentUser == nullptr){

QMessageBox::information(nullptr, "Current user exist", "Change user");

}

else{

currentAccount = new Account(dbManager->getAccountByName(ui->comboBox\_Account->currentText()));

ui->label\_Account\_Balance->setText(QString::number(currentAccount->getBalance()));

ui->label\_Account\_Amount->setText(QString::number(currentAccount->getAmount()));

ui->comboBox\_Payments->clear();

QVector<QString> paymentsNames = dbManager->getAllPaymentNamesForCurrentUser(currentAccount->getId());

for (const QString& name : paymentsNames) {

ui->comboBox\_Payments->addItem(name);

}

MainWindow::changeBalance(QString::number( currentAccount->getBalance() ));

}

}

void MainWindow::on\_pushButton\_Accounts\_changeUser\_clicked()

{

userWindow->show();

}

void MainWindow::on\_pushButton\_Recharging\_2\_clicked()

{

ui->tabWidget\_Accounts->setCurrentIndex(1);

}

void MainWindow::on\_pushButton\_Accounts\_cancel\_clicked()

{

ui->tabWidget\_Accounts->setCurrentIndex(1);

}

//Paymants

void MainWindow::on\_pushButton\_Payments\_add\_clicked()

{

ui->tabWidget\_Paymants->setCurrentIndex(1);

}

void MainWindow::on\_pushButton\_Payments\_pay\_clicked()

{

if (ui->lineEdit\_Payments\_name->text().isEmpty() ||

ui->lineEdit\_Payments\_amount->text().isEmpty()||

ui->textEdit\_Payments\_description->toPlainText().isEmpty())

{

QMessageBox::information(nullptr, "Empty field", "Enter all fields");

}

if (currentAccount->getBalance() < ui->lineEdit\_Payments\_amount->text().toDouble())

{

QMessageBox::information(nullptr, "Not enough funds", "Top up your account");

}

else{

Payment\* payment = new Payment(0,currentAccount->getId(),ui->lineEdit\_Payments\_name->text()

,ui->lineEdit\_Payments\_amount->text().toDouble() ,QDate::currentDate(),ui->textEdit\_Payments\_description->toPlainText());

if (!dbManager->inserIntoTable(\*payment)) {

qDebug() << "Error inserting data into the database.";}

currentPayment = new Payment( \*dbManager->getLastPaymentForAccount(currentAccount->getId()));

ui->comboBox\_Payments->addItem(payment->getName());

qDebug()<< currentAccount->getBalance();

currentAccount->setBalance( currentAccount->getBalance() - currentPayment->getAmount());

qDebug()<< currentAccount->getBalance();

currentAccount->setAmount( currentAccount->getAmount() + currentPayment->getAmount());

if(!dbManager->updateAccountBalance(currentAccount->getId(), currentAccount->getBalance())){

qDebug() << "Error updating data into the database.";}

if(!dbManager->updateAccountAmount(currentAccount->getId(), currentAccount->getAmount())){

qDebug() << "Error updating data into the database.";}

ui->label\_Account\_Balance->setText(QString::number( currentAccount->getBalance() ));

ui->label\_Account\_Amount->setText(QString::number( currentAccount->getAmount() ));

ui->tabWidget\_Paymants->setCurrentIndex(0);

MainWindow::changeBalance(QString::number( currentAccount->getBalance() ));

}

}

void MainWindow::on\_pushButton\_Payments\_ChosePaymant\_clicked()

{

currentPayment = new Payment(dbManager->getPaymentByName(ui->comboBox\_Payments->currentText()));

ui->label\_Paymants\_amount->setText(QString::number(currentPayment->getAmount()));

ui->label\_Paymants\_date->setText(currentPayment->getDate().toString("yyyy-MM-dd"));

ui->textBrowser->setText(currentPayment->getDescription());

}

void MainWindow::changeUsername(QString name){

ui->label\_userData\_account\_login->setText(name);

ui->label\_userData\_payment\_login->setText(name);

ui->label\_userLogin5->setText(name);

}

void MainWindow::changeBalance(QString balance){

ui->label\_userData\_account\_balance->setText(balance);

ui->label\_userData\_paiment\_balance->setText(balance);}

void MainWindow::on\_pushButton\_Payments\_cancel\_clicked()

{ui->tabWidget\_Paymants->setCurrentIndex(0);}

Додаток Ґ

**Лістинг файлу «userwindow.h»**

#ifndef USERWINDOW\_H

#define USERWINDOW\_H

#include <QDialog>

#include <QWidget>

#include <QDateTime>

#include <QSqlQuery>

#include "user.h"

#include "dbmanager.h"

#include "sqlitedbmanager.h"

#include "userwindow.h"

#include <QDebug>

#include <QMessageBox>

namespace Ui {class UserWindow;}

class UserWindow : public QDialog

{

Q\_OBJECT

public:

explicit UserWindow(DBManager\* dbManager,QWidget \*parent = nullptr);

~UserWindow();

private slots:

void on\_pushButton\_SingIn\_SIngIn\_clicked();

void on\_pushButton\_SingIn\_SIngUp\_clicked();

void on\_pushButton\_SingUp\_SIngIn\_Up\_clicked();

void on\_pushButton\_clicked();

void on\_pushButton\_SingUp\_cancel\_clicked();

void on\_pushButton\_Cancel\_clicked();

private:

Ui::UserWindow \*ui;

DBManager\* dbManager;

signals:

void setCurrentUser(User\*);};

Додаток Е

**Лістинг файлу «userwindow.cpp»**

#include "userwindow.h"

#include "ui\_userwindow.h"

#include <QTabBar>

#include <QLineEdit>

UserWindow::UserWindow(DBManager\* dbManager,QWidget \*parent) :

QDialog(parent),

ui(new Ui::UserWindow), dbManager(dbManager)

{

ui->setupUi(this);

ui->tabWidget\_user->tabBar()->hide();

ui->tabWidget\_user->setCurrentIndex(0);

ui->lineEdit\_SingIn\_password->setEchoMode(QLineEdit::Password);

ui->lineEdit\_SingUp\_password->setEchoMode(QLineEdit::Password);

}

UserWindow::~UserWindow()

{

delete ui;

}

void UserWindow::on\_pushButton\_SingIn\_SIngIn\_clicked()

{

User\* user;

if (

ui->lineEdit\_SingIn\_password->text().isEmpty() &&

ui->lineEdit\_SingIn\_name->text().isEmpty())

{

QMessageBox::information(nullptr, "Empty field", "Enter all fields");

}

else{

bool userFound = false;

user = new User(dbManager->getUserById(dbManager->findUserIdByUsername(ui->lineEdit\_SingIn\_name->text())));

if(user->getPassword() == ui->lineEdit\_SingIn\_password->text()) {

ui->label\_userLogin->setText(user->getUsername());

// ui->label\_userData\_main\_login->setText(user->getUsername());

ui->label\_userName->setText(user->getName());

ui->label\_userAddress->setText(user->getAddress());

ui->label\_userAge->setText(QString::number(user->getAge()));

ui->tabWidget\_user->setCurrentIndex(2);

userFound = true;

}

if (!userFound) {

// Виведення повідомлення про невірний логін чи пароль

QMessageBox::warning(this, "Error", "Invalid login or password", QMessageBox::Ok);

}

emit setCurrentUser(user);

close();

}

delete user;

}

void UserWindow::on\_pushButton\_SingIn\_SIngUp\_clicked()

{

ui->tabWidget\_user->setCurrentIndex(1);

}

void UserWindow::on\_pushButton\_SingUp\_SIngIn\_Up\_clicked()

{

if ( ui->lineEdit\_SingUp\_fullname->text().isEmpty() &&

ui->lineEdit\_SingUp\_address->text().isEmpty() &&

ui->lineEdit\_SingUp\_age->text().isEmpty() &&

ui->lineEdit\_SingUp\_password->text().isEmpty() &&

ui->lineEdit\_SingUp\_name->text().isEmpty())

{

QMessageBox::information(nullptr, "Empty field", "Enter all fields");

}

else{

User\* user = new User(3,ui->lineEdit\_SingUp\_fullname->text(),

ui->lineEdit\_SingUp\_address->text(),

ui->lineEdit\_SingUp\_name->text(),

ui->lineEdit\_SingUp\_password->text(),

ui->lineEdit\_SingUp\_age->text().toInt());

ui->tabWidget\_user->setCurrentIndex(0);

if (!dbManager->inserIntoTable(\*user)) {

qDebug() << "Error inserting data into the database.";}

}}

void UserWindow::on\_pushButton\_clicked()

{

ui->tabWidget\_user->setCurrentIndex(0);

}

void UserWindow::on\_pushButton\_SingUp\_cancel\_clicked()

{

ui->tabWidget\_user->setCurrentIndex(0);

}

void UserWindow::on\_pushButton\_Cancel\_clicked()

{

close();}

Додаток Є

**Лістинг файлу «user.h»**

#ifndef USER\_H

#define USER\_H

#include <QString>

#include <QVector>

#include "account.h"

class User

{

int id;

QString name;

QString address;

QString username;

QString password;

int age;

public:

User(){

}

User(int id, QString name, QString address, QString username, QString password, int age);

User(const User &other);

int getId() const;

QString getName() const;

QString getAddress() const;

QString getUsername() const;

QString getPassword() const;

int getAge() const;

void setId(int newId);

void setName(const QString &newName);

void setAddress(const QString &newAddress);

void setUsername(const QString &newUsername);

void setPassword(const QString &newPassword);

void setAge(int newAge);

QString toString() const;

};

#endif

Додаток Ж

**Лістинг файлу «user.cpp»**

#include "user.h"

User::User(int id, QString name, QString address, QString username, QString password, int age)

: id(id), name(name), address(address), username(username), password(password), age(age) {}

User::User(const User &other)

: id(other.id), name(other.name), address(other.address),

username(other.username), password(other.password), age(other.age) {

}

int User::getId() const {

return id;}

QString User::getName() const {

return name;}

QString User::getAddress() const {

return address;}

QString User::getUsername() const {

return username;}

QString User::getPassword() const {

return password;}

int User::getAge() const {

return age;}

void User::setId(int newId) {

id = newId;}

void User::setName(const QString &newName) {

name = newName;}

void User::setAddress(const QString &newAddress) {

address = newAddress;}

void User::setUsername(const QString &newUsername) {

username = newUsername;}

void User::setPassword(const QString &newPassword) {

password = newPassword;}

void User::setAge(int newAge) {

age = newAge;}

QString User::toString() const {

QString userString;

QTextStream stream(&userString);

stream << "ID: " << getId() << "\n";

stream << "Name: " << getName() << "\n";

stream << "Address: " << getAddress() << "\n";

stream << "Username: " << getUsername() << "\n";

stream << "Password: " << getPassword() << "\n";

stream << "Age: " << getAge();

return userString;}

Додаток З

**Лістинг файлу «account.h»**

#ifndef ACCOUNT\_H

#define ACCOUNT\_H

#include <QString>

#include "payment.h"

class Account

{

int id;

int userId;

QString name;

double amount;

double balance;

public:

Account(){}

Account(int id, int userId, QString name, double amount, double balance);

// Методи для отримання значень властивостей

int getId() const;

int getUserId() const;

double getAmount() const;

double getBalance() const;

QString getName() const;

// Методи для зміни значень властивостей

void setId(int newId);

void setUserId(int newUserId);

void setAmount(double newAmount);

void setBalance(double newBalance);

void setName(const QString &newName);

};

#endif // ACCOUNT\_H

Додаток І

**Лістинг файлу «account.cpp»**

#include "account.h"

Account::Account(int id, int userId,QString name, double amount, double balance)

: id(id), userId(userId), name(name), amount(amount), balance(balance) {}

int Account::getId() const {

return id;

}

int Account::getUserId() const {

return userId;

}

QString Account::getName() const{

return name;

}

double Account::getAmount() const {

return amount;

}

double Account::getBalance() const {

return balance;

}

void Account::setAmount(double newAmount) {

amount = newAmount;

}

void Account::setBalance(double newBalance) {

balance = newBalance;

}

void Account::setId(int newId) {

id = newId;

}

void Account::setUserId(int newUserId) {

userId = newUserId;

}

void Account::setName(const QString &newName) {

name = newName;

}

Додаток Ї

**Лістинг файлу «payment.h»**

#ifndef PAYMENT\_H

#define PAYMENT\_H

#include <QDate>

#include <QString>

class Payment {

public:

// Конструктор

Payment(int id,int accountId,QString name, double amount, const QDate &date, const QString &description);

Payment(){}

// Методи для отримання значень

int getId() const;

int getAccountId() const;

double getAmount() const;

QDate getDate() const;

QString getDescription() const;

QString getName() const;

void setId(int newId);

void setAccountId(int newAccountId);

void setAmount(double newAmount);

void setDate(const QDate &newDate);

void setDescription(const QString &newDescription);

void setName(const QString &newName);

private:

int id;

int accountId; // ID рахунку

QString name;

double amount; // Сума платежу

QDate date; // Дата платежу

QString description; // Опис платежу

};

#endif // PAYMENT\_H

Додаток Й

**Лістинг файлу «payment.cpp»**

#include "payment.h"

Payment::Payment(int id,int accountId,QString name, double amount, const QDate &date, const QString &description)

:id(id), accountId(accountId),name(name), amount(amount), date(date), description(description) {}

// Реалізація методів для отримання значень

int Payment::getAccountId() const {

return accountId;

}

double Payment::getAmount() const {

return amount;

}

QDate Payment::getDate() const {

return date;

}

QString Payment::getDescription() const {

return description;

}

QString Payment::getName() const{

return name;

}

int Payment::getId() const {

return id;

}

void Payment::setId(int newId) {

id = newId;

}

void Payment::setAccountId(int newAccountId) {

accountId = newAccountId;

}

void Payment::setAmount(double newAmount) {

amount = newAmount;

}

void Payment::setDate(const QDate &newDate) {

date = newDate;

}

void Payment::setDescription(const QString &newDescription) {

description = newDescription;

}

void Payment::setName(const QString &newName) {

name = newName;

}

Додаток К

**Лістинг файлу «dbmanager.h»**

#ifndef DBMANAGER\_H

#define DBMANAGER\_H

#include "payment.h"

#include "user.h"

#include "account.h"

#include <QSqlDatabase>

class DBManager {

public:

virtual void connectToDataBase() = 0;

virtual QSqlDatabase getDB() = 0;

// Суто віртуальний метод для вставки записів у таблицю

virtual bool inserIntoTable(const User& user) = 0;

virtual bool inserIntoTable(const Account& user) = 0;

virtual bool inserIntoTable(const Payment& user) = 0;

virtual bool updateAccountBalance(int accountId, double newBalance) =0;

virtual bool updateAccountAmount(int accountId, double newAmount) = 0;

virtual Payment getPaymentById(int paymentId)= 0;

virtual Account getAccountById(int accountId) =0;

virtual User getUserById(int userId) = 0;

virtual int findUserIdByUsername(const QString &usernameToFind) = 0;

virtual QVector<QString> getAllAccountNamesForCurrentUser(int currentUserId) = 0;

virtual QVector<QString> getAllPaymentNamesForCurrentUser(int currentUserId) = 0;

virtual Account getAccountByName(const QString &accountName) = 0;

virtual Payment getPaymentByName(const QString &paymentName) = 0;

virtual Payment\* getLastPaymentForAccount(int currentAccountId) = 0;

virtual Account\* getLastAccountForUser(int currentUserId) =0;

};

#endif // DBMANAGER\_H

Додаток Л

**Лістинг файлу «sqlitedbmanager.h»**

#ifndef SQLITEDBMANAGER\_H

#define SQLITEDBMANAGER\_H

#include <QSqlDatabase>

#include "dbmanager.h"

#include <QObject>

#include <QSqlQuery>

#include <QSqlError>

#include <QFile>

#include <QDate>

#include <QDebug>

#include <QVector>

#define DATABASE\_HOST\_NAME "ExampleDataBase"

#define DATABASE\_FILE\_NAME "DataBase.sqlite"

#define TABLE\_USER "Users"

#define TABLE\_USER\_ID "id"

#define TABLE\_USER\_NAME "name"

#define TABLE\_USER\_ADDRESS "address"

#define TABLE\_USER\_USERNAME "username"

#define TABLE\_USER\_PASSWORD "password"

#define TABLE\_USER\_AGE "age"

// Константи для класу Account

#define TABLE\_ACCOUNT "Accounts"

#define TABLE\_ACCOUNT\_USER\_ID "user\_id"

#define TABLE\_ACCOUNT\_NAME "name"

#define TABLE\_ACCOUNT\_AMOUNT "amount"

#define TABLE\_ACCOUNT\_BALANCE "balance"

// Константи для класу Payment

#define TABLE\_PAYMENT "Payments"

#define TABLE\_PAYMENT\_ID "id"

#define TABLE\_PAYMENT\_ACCOUNT\_ID "account\_id"

#define TABLE\_PAYMENT\_NAME "name"

#define TABLE\_PAYMENT\_AMOUNT "amount"

#define TABLE\_PAYMENT\_DATE "date"

#define TABLE\_PAYMENT\_DESCRIPTION "description"

class SqliteDBManager : public DBManager {

public:

static SqliteDBManager\* getInstance();

void connectToDataBase() override;

QSqlDatabase getDB() override;

bool inserIntoTable(const User& user) override;

bool inserIntoTable(const Account& user) override;

bool inserIntoTable(const Payment& user) override;

bool updateAccountBalance(int accountId, double newBalance) override;

bool updateAccountAmount(int accountId, double newAmount) override;

Payment getPaymentById(int paymentId) override;

Account getAccountById(int accountId) override;

User getUserById(int userId) override;

int findUserIdByUsername(const QString &usernameToFind) override;

QVector<QString> getAllAccountNamesForCurrentUser(int currentUserId) override;

QVector<QString> getAllPaymentNamesForCurrentUser(int currentUserId) override;

Account getAccountByName(const QString &accountName) override;

Payment getPaymentByName(const QString &paymentName) override;

Payment\* getLastPaymentForAccount(int currentAccountId) override;

Account\* getLastAccountForUser(int currentUserId) override;

private:

QSqlDatabase db;

static SqliteDBManager\* instance;

SqliteDBManager();

bool openDataBase();

bool restoreDataBase();

void closeDataBase();

bool createTables();

};

#endif // SQLITEDBMANAGER\_H

Додаток M

**Лістинг файлу «sqlitedbmanager.cpp»**

#include "sqlitedbmanager.h"

#include <QObject>

#include <QSqlQuery>

#include <QSqlError>

#include <QFile>

#include <QDate>

#include <QDebug>

SqliteDBManager\* SqliteDBManager::instance = nullptr;

SqliteDBManager::SqliteDBManager() {

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

db.setHostName(DATABASE\_HOST\_NAME);

db.setDatabaseName(DATABASE\_FILE\_NAME);

}

// Метод для отримання екземпляру даного класу (патерн Singleton)

SqliteDBManager\* SqliteDBManager::getInstance() {

if (instance == nullptr) {

instance = new SqliteDBManager();

}

return instance;

}

// Метод для підключення до бази даних

void SqliteDBManager::connectToDataBase() {

/\* Перед підключенням до бази даних виконуємо перевірку на її існування.

\* В залежності від результату виконуємо відкриття бази даних або її відновлення

\* \*/

if (QFile(DATABASE\_FILE\_NAME).exists()) {

this->openDataBase();

} else {

this->restoreDataBase();

}

}

// Метод для отримання обробника підключення до БД

QSqlDatabase SqliteDBManager::getDB() {

return db;

}

// Метод відновлення бази даних

bool SqliteDBManager::restoreDataBase() {

if (!this->openDataBase()) {

qDebug() << "Не вдалось відкрити базу даних";

return false;

}

qDebug() << "Базу даних відкрито успішно";

if (!this->createTables()) {

qDebug() << "Не вдалось створити таблиці";

return false;

} else {

qDebug() << "Таблиці створено успішно";

return true;

}

}

// Метод для відкриття бази даних

bool SqliteDBManager::openDataBase() {

/\* База даних відкривається по вказаному шляху

\* та імені бази даних, якщо вона існує

\* \*/

if (db.open()) {

return true;

} else

return false;

}

// Метод закриття бази даних

void SqliteDBManager::closeDataBase() {

db.close();

}

// Метод для створення таблиці в базі даних

bool SqliteDBManager::createTables() {

QSqlQuery query;

if (!query.exec("CREATE TABLE " TABLE\_USER " ("

"id INTEGER PRIMARY KEY AUTOINCREMENT, "

TABLE\_USER\_NAME " VARCHAR(255) NOT NULL,"

TABLE\_USER\_ADDRESS " VARCHAR(255) NOT NULL,"

TABLE\_USER\_USERNAME " VARCHAR(255) NOT NULL,"

TABLE\_USER\_PASSWORD " VARCHAR(255) NOT NULL,"

TABLE\_USER\_AGE " INTEGER NOT NULL"

" )"

)) {

qDebug() << "DataBase: error of create " << TABLE\_USER;

qDebug() << query.lastError().text();

return false;

}

else if (!query.exec("CREATE TABLE " TABLE\_ACCOUNT " ("

"id INTEGER PRIMARY KEY AUTOINCREMENT, "

TABLE\_ACCOUNT\_NAME " VARCHAR(255) NOT NULL,"

TABLE\_ACCOUNT\_AMOUNT " DECIMAL(10, 2) NOT NULL,"

TABLE\_ACCOUNT\_BALANCE " DECIMAL(10, 2) NOT NULL,"

TABLE\_ACCOUNT\_USER\_ID " INTEGER NOT NULL,"

"FOREIGN KEY ( " TABLE\_ACCOUNT\_USER\_ID ") REFERENCES " TABLE\_USER " ( id ) )")) {

qDebug() << "DataBase: error of create " << TABLE\_ACCOUNT;

qDebug() << query.lastError().text();

return false;

} else if (!query.exec("CREATE TABLE " TABLE\_PAYMENT " ("

"id INTEGER PRIMARY KEY AUTOINCREMENT, "

TABLE\_PAYMENT\_NAME " VARCHAR(255) NOT NULL,"

TABLE\_PAYMENT\_AMOUNT " DECIMAL(10, 2) NOT NULL,"

TABLE\_PAYMENT\_DATE " DATE NOT NULL,"

TABLE\_PAYMENT\_DESCRIPTION " VARCHAR(255) NOT NULL,"

TABLE\_PAYMENT\_ACCOUNT\_ID " INTEGER NOT NULL,"

"FOREIGN KEY ( " TABLE\_PAYMENT\_ACCOUNT\_ID ") REFERENCES " TABLE\_ACCOUNT " ( id ) )")) {

qDebug() << "DataBase: error of create " << TABLE\_ACCOUNT;

qDebug() << query.lastError().text();

return false;

}else

return true;

}

// Метод для вставки записів у таблицю messages

bool SqliteDBManager::inserIntoTable(const User& user) {

// SQL-запит формується із об'єкта класу Messageu

// qDebug() << user.toString();

QSqlQuery query;

/\*

\* Спочатку SQL-запит формується з ключами, які потім зв'язуються методом bindValue

\* для підставки даних із об'єкта класу Message

\* \*/

query.prepare("INSERT INTO " TABLE\_USER " ( "

TABLE\_USER\_NAME ", "

TABLE\_USER\_ADDRESS ", "

TABLE\_USER\_USERNAME ", "

TABLE\_USER\_PASSWORD ", "

TABLE\_USER\_AGE " ) "

"VALUES (:Name, :Address, :Username, :Password, :Age )");

query.bindValue(":Name", user.getName());

query.bindValue(":Address", user.getAddress());

query.bindValue(":Username", user.getUsername());

query.bindValue(":Password", user.getPassword());

query.bindValue(":Age", user.getAge());

// Після чого виконується запит методом exec()

if (!query.exec()) {

qDebug() << "error insert into " << TABLE\_USER;

qDebug() << query.lastError().text();

qDebug() << query.lastQuery();

return false;

} else

return true;

}

bool SqliteDBManager::inserIntoTable(const Account& account) {

// SQL-запит формується із об'єкта класу Messageu

// qDebug() << user.toString();

QSqlQuery query;

/\*

\* Спочатку SQL-запит формується з ключами, які потім зв'язуються методом bindValue

\* для підставки даних із об'єкта класу Message

\* \*/

query.prepare("INSERT INTO " TABLE\_ACCOUNT " ( "

TABLE\_ACCOUNT\_NAME ", "

TABLE\_ACCOUNT\_AMOUNT ", "

TABLE\_ACCOUNT\_BALANCE ", "

TABLE\_ACCOUNT\_USER\_ID " ) "

"VALUES (:Name, :Amount, :Balance, :UserId )");

query.bindValue(":Name", account.getName());

query.bindValue(":Amount", account.getAmount());

query.bindValue(":Balance", account.getBalance());

query.bindValue(":UserId", account.getUserId());

// Після чого виконується запит методом exec()

if (!query.exec()) {

qDebug() << "error insert into " << TABLE\_ACCOUNT;

qDebug() << query.lastError().text();

qDebug() << query.lastQuery();

return false;

} else

return true;

}

bool SqliteDBManager::inserIntoTable(const Payment& payment) {

// SQL-запит формується із об'єкта класу Messageu

// qDebug() << user.toString();

QSqlQuery query;

/\*

\* Спочатку SQL-запит формується з ключами, які потім зв'язуються методом bindValue

\* для підставки даних із об'єкта класу Message

\* \*/

query.prepare("INSERT INTO " TABLE\_PAYMENT " ( "

TABLE\_PAYMENT\_NAME ", "

TABLE\_PAYMENT\_AMOUNT ", "

TABLE\_PAYMENT\_DATE ", "

TABLE\_PAYMENT\_DESCRIPTION ", "

TABLE\_PAYMENT\_ACCOUNT\_ID " ) "

"VALUES (:Name, :Amount, :Date, :Description, :AccountId)");

query.bindValue(":Name", payment.getName());

query.bindValue(":Amount", payment.getAmount());

query.bindValue(":Date", payment.getDate());

query.bindValue(":Description", payment.getDescription());

query.bindValue(":AccountId", payment.getAccountId());

// Після чого виконується запит методом exec()

if (!query.exec()) {

qDebug() << "error insert into " << TABLE\_PAYMENT;

qDebug() << query.lastError().text();

qDebug() << query.lastQuery();

return false;

} else

return true;

}

bool SqliteDBManager::updateAccountBalance(int accountId, double newBalance) {

QSqlQuery query;

// SQL query to update the balance for the specified account

query.prepare("UPDATE " TABLE\_ACCOUNT " SET "

TABLE\_ACCOUNT\_BALANCE " = :NewBalance "

"WHERE id = :AccountId");

query.bindValue(":NewBalance", newBalance);

query.bindValue(":AccountId", accountId);

// Execute the query

if (!query.exec()) {

qDebug() << "Error updating account balance";

qDebug() << query.lastError().text();

qDebug() << query.lastQuery();

return false;

} else {

qDebug() << "Account balance updated successfully";

return true;

}

}

bool SqliteDBManager::updateAccountAmount(int accountId, double newAmount) {

QSqlQuery query;

// SQL query to update the amount for the specified account

query.prepare("UPDATE " TABLE\_ACCOUNT " SET "

TABLE\_ACCOUNT\_AMOUNT " = :NewAmount "

"WHERE id = :AccountId");

query.bindValue(":NewAmount", newAmount);

query.bindValue(":AccountId", accountId);

// Execute the query

if (!query.exec()) {

qDebug() << "Error updating account amount";

qDebug() << query.lastError().text();

qDebug() << query.lastQuery();

return false;

} else {

qDebug() << "Account amount updated successfully";

return true;

}

}

User SqliteDBManager::getUserById(int userId) {

QSqlQuery query;

User user;

// SQL query to retrieve user data based on the user ID

query.prepare("SELECT \* FROM " TABLE\_USER " WHERE id = :UserId");

query.bindValue(":UserId", userId);

// Execute the query

if (query.exec() && query.next()) {

// Retrieve data from the query result and populate the User object

user.setId(query.value("id").toInt());

user.setName(query.value(TABLE\_USER\_NAME).toString());

user.setAddress(query.value(TABLE\_USER\_ADDRESS).toString());

user.setUsername(query.value(TABLE\_USER\_USERNAME).toString());

user.setPassword(query.value(TABLE\_USER\_PASSWORD).toString());

user.setAge(query.value(TABLE\_USER\_AGE).toInt());

qDebug() << "User data retrieved successfully";

} else {

qDebug() << "Error retrieving user data";

qDebug() << query.lastError().text();

qDebug() << query.lastQuery();

}

return user;

}

Account SqliteDBManager::getAccountById(int accountId) {

QSqlQuery query;

Account account;

query.prepare("SELECT \* FROM " TABLE\_ACCOUNT " WHERE id = :AccountId");

query.bindValue(":AccountId", accountId);

if (query.exec() && query.next()) {

account.setId(query.value("id").toInt());

account.setName(query.value(TABLE\_ACCOUNT\_NAME).toString());

account.setAmount(query.value(TABLE\_ACCOUNT\_AMOUNT).toDouble());

account.setBalance(query.value(TABLE\_ACCOUNT\_BALANCE).toDouble());

account.setUserId(query.value(TABLE\_ACCOUNT\_USER\_ID).toInt());

qDebug() << "Account data retrieved successfully";

} else {

qDebug() << "Error retrieving account data";

qDebug() << query.lastError().text();

qDebug() << query.lastQuery();

}

return account;

}

Payment SqliteDBManager::getPaymentById(int paymentId) {

QSqlQuery query;

Payment payment;

query.prepare("SELECT \* FROM " TABLE\_PAYMENT " WHERE id = :PaymentId");

query.bindValue(":PaymentId", paymentId);

if (query.exec() && query.next()) {

payment.setId(query.value("id").toInt());

payment.setName(query.value(TABLE\_PAYMENT\_NAME).toString());

payment.setAmount(query.value(TABLE\_PAYMENT\_AMOUNT).toDouble());

payment.setDate(query.value(TABLE\_PAYMENT\_DATE).toDate());

payment.setDescription(query.value(TABLE\_PAYMENT\_DESCRIPTION).toString());

payment.setAccountId(query.value(TABLE\_PAYMENT\_ACCOUNT\_ID).toInt());

qDebug() << "Payment data retrieved successfully";

} else {

qDebug() << "Error retrieving payment data";

qDebug() << query.lastError().text();

qDebug() << query.lastQuery();

}

return payment;

}

int SqliteDBManager::findUserIdByUsername(const QString &usernameToFind) {

QSqlQuery query;

int userId = -1; // Assuming -1 is used to indicate that the user was not found

// SQL query to retrieve the ID of the user with the given username

query.prepare("SELECT id FROM " TABLE\_USER " WHERE " TABLE\_USER\_USERNAME " = :Username");

query.bindValue(":Username", usernameToFind);

// Execute the query

if (query.exec() && query.next()) {

// Retrieve the user ID from the query result

userId = query.value("id").toInt();

qDebug() << "User ID found for username" << usernameToFind << ": " << userId;

} else {

qDebug() << "Error finding user ID for username" << usernameToFind;

qDebug() << query.lastError().text();

qDebug() << query.lastQuery();

}

return userId;

}

QVector<QString> SqliteDBManager::getAllAccountNamesForCurrentUser(int currentUserId) {

QSqlQuery query;

QVector<QString> accountNames;

// SQL query to retrieve all account names for the current user

query.prepare("SELECT " TABLE\_ACCOUNT\_NAME ", " TABLE\_ACCOUNT\_USER\_ID " FROM " TABLE\_ACCOUNT);

// Execute the query

if (query.exec()) {

// Fetch account names for the current user and add each one to the QVector

while (query.next()) {

QString accountName = query.value(TABLE\_ACCOUNT\_NAME).toString();

int userId = query.value(TABLE\_ACCOUNT\_USER\_ID).toInt();

// Check if the user\_id matches the currentUser->id

if (userId == currentUserId) {

accountNames.append(accountName);

}

}

qDebug() << "All account names for the current user retrieved successfully";

} else {

qDebug() << "Error retrieving account names for the current user";

qDebug() << query.lastError().text();

qDebug() << query.lastQuery();

}

return accountNames;

}

Account SqliteDBManager::getAccountByName(const QString &accountName) {

QSqlQuery query;

Account foundAccount;

// SQL query to retrieve the account by name

query.prepare("SELECT \* FROM " TABLE\_ACCOUNT " WHERE " TABLE\_ACCOUNT\_NAME " = :AccountName");

query.bindValue(":AccountName", accountName);

// Execute the query

if (query.exec() && query.next()) {

foundAccount.setId(query.value("id").toInt());

foundAccount.setName(query.value(TABLE\_ACCOUNT\_NAME).toString());

foundAccount.setAmount(query.value(TABLE\_ACCOUNT\_AMOUNT).toDouble());

foundAccount.setBalance(query.value(TABLE\_ACCOUNT\_BALANCE).toDouble());

foundAccount.setUserId(query.value(TABLE\_ACCOUNT\_USER\_ID).toInt());

qDebug() << "Account data retrieved successfully";

} else {

qDebug() << "Error retrieving account data";

qDebug() << query.lastError().text();

qDebug() << query.lastQuery();

}

return foundAccount;

}

QVector<QString> SqliteDBManager::getAllPaymentNamesForCurrentUser(int currentUserId) {

QSqlQuery query;

QVector<QString> paymentNames;

// SQL query to retrieve all payment names for the current user

query.prepare("SELECT " TABLE\_PAYMENT\_NAME ", " TABLE\_PAYMENT\_ACCOUNT\_ID " FROM " TABLE\_PAYMENT);

// Execute the query

if (query.exec()) {

// Fetch payment names for the current user and add each one to the QVector

while (query.next()) {

QString paymentName = query.value(TABLE\_PAYMENT\_NAME).toString();

int accountId = query.value(TABLE\_PAYMENT\_ACCOUNT\_ID).toInt();

// Check if the account\_id matches the currentUserId

if (accountId == currentUserId) {

paymentNames.append(paymentName);

}

}

qDebug() << "All payment names for the current user retrieved successfully";

} else {

qDebug() << "Error retrieving payment names for the current user";

qDebug() << query.lastError().text();

qDebug() << query.lastQuery();

}

return paymentNames;

}

Payment SqliteDBManager::getPaymentByName(const QString &paymentName) {

QSqlQuery query;

Payment foundPayment;

// SQL query to retrieve the payment by name

query.prepare("SELECT \* FROM " TABLE\_PAYMENT " WHERE " TABLE\_PAYMENT\_NAME " = :PaymentName");

query.bindValue(":PaymentName", paymentName);

if (query.exec() && query.next()) {

foundPayment.setId(query.value("id").toInt());

foundPayment.setName(query.value(TABLE\_PAYMENT\_NAME).toString());

foundPayment.setAmount(query.value(TABLE\_PAYMENT\_AMOUNT).toDouble());

foundPayment.setDate(query.value(TABLE\_PAYMENT\_DATE).toDate());

foundPayment.setDescription(query.value(TABLE\_PAYMENT\_DESCRIPTION).toString());

foundPayment.setAccountId(query.value(TABLE\_PAYMENT\_ACCOUNT\_ID).toInt());

qDebug() << "Payment data retrieved successfully";

} else {

qDebug() << "Error retrieving payment data";

qDebug() << query.lastError().text();

qDebug() << query.lastQuery();

}

// Execute the query

return foundPayment;

}

Payment\* SqliteDBManager::getLastPaymentForAccount(int currentAccountId) {

QSqlQuery query;

Payment \*lastPayment = nullptr;

// SQL query to retrieve the last payment for the specified account ID

query.prepare("SELECT \* FROM " TABLE\_PAYMENT " WHERE " TABLE\_PAYMENT\_ACCOUNT\_ID " = :AccountId ORDER BY id DESC LIMIT 1");

query.bindValue(":AccountId", currentAccountId);

// Execute the query

if (query.exec() && query.next()) {

// Retrieve payment information from the query result and create a Payment object

int paymentId = query.value(TABLE\_PAYMENT\_ID).toInt();

QString paymentName = query.value(TABLE\_PAYMENT\_NAME).toString();

double paymentAmount = query.value(TABLE\_PAYMENT\_AMOUNT).toDouble();

QDate paymentDate = query.value(TABLE\_PAYMENT\_DATE).toDate();

QString paymentDescription = query.value(TABLE\_PAYMENT\_DESCRIPTION).toString();

lastPayment = new Payment(paymentId, currentAccountId,paymentName, paymentAmount, paymentDate, paymentDescription);

qDebug() << "Last payment retrieved successfully for account ID" << currentAccountId;

} else {

qDebug() << "Error retrieving last payment for account ID" << currentAccountId;

qDebug() << query.lastError().text();

qDebug() << query.lastQuery();

}

return lastPayment;

}

Account\* SqliteDBManager::getLastAccountForUser(int currentUserId) {

QSqlQuery query;

Account \*lastAccount = nullptr;

// SQL query to retrieve the last account for the specified user ID

query.prepare("SELECT \* FROM " TABLE\_ACCOUNT " WHERE " TABLE\_ACCOUNT\_USER\_ID " = :UserId ORDER BY id DESC LIMIT 1");

query.bindValue(":UserId", currentUserId);

// Execute the query

if (query.exec() && query.next()) {

// Retrieve account information from the query result and create an Account object

int accountId = query.value("id").toInt();

QString accountName = query.value(TABLE\_ACCOUNT\_NAME).toString();

double amount = query.value(TABLE\_ACCOUNT\_AMOUNT).toDouble();

double balance = query.value(TABLE\_ACCOUNT\_BALANCE).toDouble();

lastAccount = new Account(accountId, currentUserId, accountName, amount, balance);

qDebug() << "Last account retrieved successfully for user ID" << currentUserId;

} else {

qDebug() << "Error retrieving last account for user ID" << currentUserId;

qDebug() << query.lastError().text();

qDebug() << query.lastQuery();

}

return lastAccount;

}

Компакт-диск із програмним продуктом