

**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ КИЇВСЬКИЙ  
ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. І. Сікорського  
Кафедра інформаційних систем та технологій**

**КУРСОВА РОБОТА**

з дисципліни «Основи програмування»

на тему:

**ПЛАНУВАЛЬНИК ЗАВДАНЬ. РОЗПОДІЛ ТА КОНТРОЛЬ ЗА  
ВИКОНАННЯМ ЗАВДАНЬ ЧЛЕНАМИ КОМАНДИ ПРОЕКТУ.**

Виконала студентка  
II курсу групи ЗПІ-зп-21  
Спеціальність: 121 Інженерія  
програмного забезпечення  
Луценко Л. В.  
Керівник старший викладач  
Проскура С.Л.

Кількість балів: \_\_\_\_\_

Національна оцінка \_\_\_\_\_

**Члени комісії**

\_\_\_\_\_  
(підпис)

\_\_\_\_\_  
(вчене звання, науковий ступінь, прізвище та ініціали)

\_\_\_\_\_  
(підпис)

\_\_\_\_\_  
(вчене звання, науковий ступінь, прізвище та ініціали)

Київ – 2023

**ЛИСТ ЗАВДАННЯ**  
**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ КИЇВСЬКИЙ**  
**ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. І. Сікорського**  
**Кафедра інформаційних систем та технологій**

Дисципліна Основи програмування  
Спеціальність 121 «Інженерія програмного забезпечення»  
Курс II Група ЗПІ-зп-21 Семестр II

**ЗАВДАННЯ на курсову роботу студентки**  
**Луценко Людмили Володимирівни**

1. Тема проекту(роботи): Планувальник завдань. Розподіл та контроль за виконанням завдань членами команди проекту.
2. Строк здачі студентом закінченого проекту (роботи): 29 травня 2023 року
3. Вихідні дані до проекту (роботи):

При виконанні проекту постає низка завдань, які потребують їх виконання. Завдання має опис його сутності, час, необхідний для його виконання, та пріоритет. Завдання перебуває в одному зі статусів: не розпочато, на виконанні, виконано. Команда проекту складається з робітників, яким розподіляються завдання на виконання відповідно до їхньої зайнятості.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які підлягають розробці):

Аналіз предметної області. Опис програмного забезпечення. Кодування (програмування). Результати тестування.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень): Діаграма класів.
6. Дата видачі завдання: 10 березня 2023 року.

## КАЛЕНДАРНИЙ ПЛАН

<b>№ з/п</b>	<b>Назва етапів курсової роботи</b>	<b>Термін виконання етапів роботи</b>	<b>Підписи керівника, студента</b>
1.	Отримання теми курсової роботи	24.03.2023 р.	
2.	Пошук та вивчення літератури з питань курсової роботи	27.03.2023 р.	
3.	Об'єктно-орієнтований аналіз предметної області. Функціональні і нефункціональні вимоги до програми	11.04.2023 р.	
4.	Розробка сценарію роботи програми	19.04.2023 р.	
5.	Проектування архітектури програми	26.04.2023 р.	
6.	Узгодження з керівником інтерфейсу користувача	27.04.2023 р.	
7.	Розробка програмного забезпечення	10.05.2023 р.	
8.	Узгодження з керівником плану тестування	11.05.2023 р.	
9.	Тестування програми	19.05.2023 р.	
10.	Підготовка пояснювальної записки	26.05.2023 р.	
11.	Здача курсової роботи на перевірку	29.05.2023 р.	
12.	Захист курсової роботи		

## ЗМІСТ

Зміст.....	4
Вступ.....	6
1 Аналіз предметної галузі.....	7
2 Опис програмного забезпечення.....	12
2.1 Технічне завдання.....	12
2.2 Розробка програмного забезпечення.....	12
2.3 Сценарій роботи додатку.....	21
3 Програмування додатку.....	24
3.1 Лістинг програмного коду користувацького інтерфейсу EmployeeTasksManagingApp .....	24
3.1.1 Клас Program: .....	24
3.1.2 Клас LoginForm: .....	24
3.1.3 Клас TaskManagingForm: .....	26
3.2 Лістинг програмного коду ModelLibrary.....	28
3.2.1 Клас Employee: .....	28
3.2.2 Клас EmployeeTask:.....	28
3.2.3 Enum TaskState:.....	29
3.3 Лістинг програмного коду DataAccessLibrary .....	29
3.3.1 Клас Config:.....	29
3.3.2 Інтерфейс IDbContext:.....	30
3.3.3 Клас DbContext: .....	31
3.4 Лістинг програмного коду Business Library.....	31
3.4.1 Клас EmployeeOperations:.....	31
3.4.2 Клас EmployeeTaskOperations: .....	32

4	Тестування програмного забезпечення.....	35
4.1	Лістинг програмного коду BusinessLibraryTests .....	37
4.1.1	Клас EmployeeOperationsTest:.....	37
4.1.2	Клас EmployeeTaskOPTests:.....	38
4.2	Результати тестів .....	42
	Висновки .....	43
	Висновок до розділу 1.....	43
	Висновок до розділу 2.....	43
	Висновок до розділу 3.....	44
	Висновок до розділу 4.....	44
	Бібліографія.....	<b>Помилка! Закладку не визначено.</b>

## **ВСТУП.**

Управління командою є однією з важливих речей, якими повинен володіти менеджер. Це сприяє підвищенню задоволеності співробітників, а також формує довіру. Крім того, це також зменшує ймовірність непотрібних конфліктів між членами команди. Однак керувати командою не так просто, як може здатися. Особливо, якщо організаційна структура складна, а команда величезна [1]. Ось тут і з'являється програмне забезпечення для управління командою.

Додатки контролю виконання співробітниками завдань – це програми, які дозволяють керівникам призначати, відстежувати й оцінювати роботу своїх підлеглих з різних проектів і завдань. Такі програми можуть покращити продуктивність, комунікацію та співпрацю в команді, а також допомогти уникнути прострочень, помилок та конфліктів.

Курсова робота присвячена проблемі організації робочого часу співробітників, які утворюють команду, що працює над розробкою програмних продуктів.

Метою роботи є створення додатку, який дозволяє керувати ходом розробки програмного продукту, розподіляти завдання між членами команди та відслідковувати хід їх виконання.

Результатом роботи є додаток, який містить форму авторизації та форму управління завданнями і дозволяє організувати робочий час команди співробітників проекту шляхом розподілу завдань між ними і контролю за ходом їх виконання.

## 1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ.

Перш ніж приступити до реалізації поставленого завдання, потрібно проаналізувати предметну галузь даної роботи.

Існує безліч додатків для планування та контролю завдань для команди співробітників, але з найпопулярніших у 2023 році можна назвати:

- Todoist – простий і зручний планувальник справ, який одночасно синхронізується з кількома пристроями. Дозволяє створювати проекти, підзадачі, чек-листи, мітки, фільтри, коментарі та нагадування. Підтримує спільну роботу в команді та інтеграцію з іншими програмами;
- Trello – популярний таск-менеджер, що базується на методології канбан. Представляє робочий процес у вигляді дощок, списків та карток, які можна перетягувати мишею. Підходить для керування різноманітними проектами у команді. Має безліч налаштувань, розширень та інтеграцій;
- Asana – потужний інструмент для планування та контролю завдань у команді. Дозволяє створювати проекти, завдання, підзавдання, розділи, віхи, залежності тощо. Має різні режими відображення: списком, календарем, дошкою чи діаграмою Ганта. Має просунуті функції для співпраці, звітності та автоматизації;
- Timely – сервіс, який поєднує функції планування та моніторингу часу. Допомогає оцінювати витрати часу на завдання та проекти, а також відстежувати фактичний час роботи у команді. Автоматично записує всі активності на комп'ютері та мобільному;
- Any.do – популярний і нескладний планувальник справ, який одночасно синхронізується з кількома пристроями. Дозволяє створювати списки завдань, нагадування, чек-листи, календарі та ділитися ними з іншими користувачами;

- Google Tasks – простий та безкоштовний task-менеджер від Google, який інтегрується з Gmail, Google Calendar та Google Keep. Дозволяє створювати та редагувати завдання, додавати підзавдання та дедлайни, переміщувати завдання між списками;
- EverNote – потужний інструмент для створення нотаток, які можна організовувати в блокноти, теги та стеки. Дозволяє додавати до нотаток різні типи контенту: текст, аудіо, відео, зображення, скріншоти тощо. Підтримує спільну роботу над нотатками та інтеграцію з іншими сервісами;
- WEEEEK – сучасний task-менеджер, призначений для віддаленої роботи. Дозволяє створювати проекти, завдання, підзавдання, розділи та віхи. Має різні режими відображення: списком, календарем, дошкою чи діаграмою Ганта. Має просунуті функції для співпраці, звітності та автоматизації.

Цілі та завдання, які вирішують такі додатки:

- підвищення ефективності та продуктивності роботи працівників за рахунок оптимального розподілу ресурсів, часу та обов'язків між учасниками проекту;
- зниження ризиків та збитків через прострочення, помилки, конфлікти або несумлінність співробітників за рахунок своєчасного виявлення та усунення проблем у робочому процесі;
- поліпшення комунікації та співробітництва в команді за рахунок забезпечення прозорості, передбачуваності та об'єктивності контролю, а також можливості обміну інформацією, файлами, коментарями та фідбеком по задачам;
- підвищення мотивації та залучення співробітників за рахунок формування сприятливої обстановки в колективі, обліку



індивідуальних особливостей та потреб кожного виконавця, а також винагороди за досягнення цілей та результатів;

- стимулювання розвитку та зростання компанії за рахунок покращення якості роботи, підвищення конкурентоспроможності та репутації на ринку, а також отримання достовірних даних для аналізу та прийняття стратегічних рішень.

Спробую порівняти ці додатки за кількома критеріями:

#### *Функціонал.*

Всі ці програми дозволяють створювати та редагувати завдання, додавати підзавдання, терміни, мітки та коментарі. Однак деякі з них мають додаткові функції, які можуть бути корисними для різних цілей. Наприклад, EverNote хороший для створення нотаток з різними типами контенту, WEEEEK підтримує діаграму Ганта та автоматизацію процесів, Google Tasks інтегрується з іншими сервісами Google, Any.do має голосове введення та список покупок, Todoist пропонує просунуту систему фільтрів та звітів.

#### *Інтерфейс.*

Всі ці програми мають сучасний та інтуїтивно зрозумілий інтерфейс, який адаптується під різні пристрої та платформи. Однак деякі з них можуть бути зручнішими або привабливішими для різних користувачів. Наприклад, Trello має візуальний інтерфейс з картками та дошками, WEEEEK має яскравий та мінімалістичний дизайн з тижневим календарем, EverNote має багато налаштувань та опцій для оформлення нотаток, Any.do має простий та елегантний інтерфейс з анімацією, Todoist має класичний інтерфейс зі списками та кольорами.

#### *Вартість.*

Всі ці програми мають безкоштовні версії з обмеженим функціоналом та платні тарифи з розширеними можливостями. Проте вартість та умови платних тарифів можуть відрізнятись. Наприклад, Trello коштує 5 доларів на місяць за

одного користувача на стандартному тарифі та 10 доларів на преміум-тарифі, WEEEK коштує 5 доларів на місяць за одного користувача на командному тарифі та 10 доларів на професійному тарифі, EverNote коштує 4 долари на місяць за одного користувача на базовому тарифі та 8 доларів на преміум-тарифі, Any.do коштує 3 долари на місяць за одного користувача на преміум-тарифі та 5 доларів на командному тарифі, Todoist коштує 4 долари на місяць за одного користувача на преміум-тарифі та 6 доларів на бізнес-тарифі.

Думки щодо корисності таких програм можуть бути різними, залежно від особистих уподобань та цілей користувачів. Однак для планування та контролю завдань загалом можна виділити кілька переваг використання додатків:

- а) вони допомагають підвищити продуктивність та ефективність роботи: за допомогою цих додатків можна легко організувати свої завдання, розставити пріоритети, відстежувати прогрес та терміни виконання, отримувати нагадування та повідомлення, а також аналізувати свою роботу за допомогою статистики та звітів;
- б) вони полегшують командну роботу та співпрацю: за допомогою цих програм можна легко ділитися завданнями та проектами з іншими користувачами, обмінюватися інформацією та коментарями, координувати дії та контролювати результати;
- в) вони спрощують управління різними аспектами життя: за допомогою цих додатків можна легко планувати не лише робочі, але й особисті справи, такі як список покупок, здоров'я, хобі, мандрівки тощо і їх можна інтегрувати з іншими сервісами, такими як календарі, пошта, хмари, голосові помічники і т.п.

Деякі недоліки використання таких програм можуть бути такими:

- а) вони можуть бути складними або незрозумілими у використанні, а деякі з цих програм мають:

- 1) багато функцій та налаштувань, які можуть бути надмірними або заплутаними для деяких користувачів;
  - 2) незручний або незвичний інтерфейс;
- б) вони можуть бути дорогими чи обмеженими у функціоналі, а деякі з цих програм мають:
- 1) платні тарифи або підписки, які можуть бути недоступними або не вигідними для деяких користувачів;
  - 2) безкоштовні версії з обмеженим функціоналом або рекламою.
- в) вони можуть бути ненадійними чи небезпечними, а деякі з цих програм мають:
- 1) проблеми із синхронізацією, збереженням або відновленням даних;
  - 2) проблеми із захистом персональної інформації або конфіденційних даних.

Думки співробітників компаній, які використовують такі програми, також можуть бути різними. Деякі з них можуть вважати, що такі програми допомагають їм краще організувати свою роботу, підвищити свою мотивацію та відповідальність, а також покращити комунікацію та співпрацю з колегами. Інші ж можуть вважати, що такі програми створюють для них додатковий стрес, порушують їхній особистий простір і свободу, а також підривають їхню довіру до керівництва.

## **2 ОПИС ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.**

### **2.1 Технічне завдання.**

Додаток повинен здійснювати розподіл та контроль за виконанням завдань членами команди проекту. При виконанні будь якого проекту постає низка завдань, які потребують їх виконання, бажано у найкоротші терміни. Кожне з цих завдань повинно мати опис його сутності, час, необхідний для його виконання, та пріоритет. Завдання може перебувати в одному зі статусів: нове, на виконанні, виконано.

Автором завдання зазвичай виступає менеджер проекту, який як правило є керівником команди розробників. Саме він має розподіляти завдання між членами своєї команди і слідкувати за їх виконанням.

Команда проекту має складатися з співробітників, яким розподіляються завдання з вказанням часу, необхідного для виконання.

### **2.2 Розробка програмного забезпечення.**

Під час створення додатку розроблені інтерфейс користувача, база даних та, окремим проектом, логіка роботи додатку. При розробці додатку використана тришарова архітектура, яка складається з таких частин:

- шар взаємодії з користувачем, який представлено користувацьким інтерфейсом;
- шар бізнес логіки, який визначає правила, алгоритми реакції програми на дії користувача або на внутрішні події, правила обробки даних;
- шар доступу до даних - зберігання, вибірка, модифікація і видалення даних, пов'язаних з розв'язуваною додатком прикладною задачею.

Перший шар – користувацький інтерфейс, який реалізовано за допомогою шаблону Windows Forms App.

Windows Forms – інтерфейс програмування додатків (API), відповідальний за графічний інтерфейс користувача і є частиною Microsoft .NET Framework [2]. Бібліотека Windows Forms була розроблена як частина .NET Framework для спрощення розробки компонентів графічного інтерфейсу користувача. Windows Forms надає можливість розробки кросплатформного графічного користувацького інтерфейсу.

Другий шар – логіка роботи додатку, яку реалізовано за допомогою шаблону Class Library.

Бібліотека класів – це набір ресурсів, що використовуються під час розробки або виконання програм. Конкретний набір ресурсів залежить від того, для чого саме вживають бібліотеку. На стадії розробки користуються бібліотеками, які надають класи, об'єкти, функції, іноді також зміни та інші дані. Такі бібліотеки надаються у вигляді вихідного коду або скомпільованого коду. Різні програми можуть використовувати ці ресурси статично (код бібліотечних функцій та інших ресурсів включається в один файл, що виконується) або динамічно (спеціальний файл з кодом повинен бути присутнім під час виконання програми). В останньому випадку одна й та ж бібліотека може бути використана одночасно в декількох застосунках. Платформа .NET підтримує механізм динамічних бібліотек. Стандартні класи згруповані у складання та скомпільовані у бібліотеки динамічного компонування (Dynamic-link library). Такі файли зазвичай мають розширення dll.

Під час створення нового додатку в середовищі MS Visual Studio до проекту автоматично підключаються найбільш вживані стандартні бібліотеки .NET.

Іноді виникає необхідність у створенні власних бібліотек класів. Такі бібліотеки групуватимуть простори імен та класи, які потім можуть бути

застосовані в інших проектах. У найпростішому випадку ці бібліотеки будуть застосовані у проектах поточного рішення. Щоб вживати класи з нової бібліотеки, її підключення слід здійснити вручну для кожного проекту, де вона необхідна.

Третій шар – база даних, який реалізований за допомогою Microsoft SQL Server.

Microsoft SQL Server – система управління базами даних, яка розробляється корпорацією Microsoft [3]. Як сервер даних виконує головну функцію по збереженню та наданню даних у відповідь на запити інших застосунків, які можуть виконуватися як на тому ж самому сервері, так і у мережі.

Мова, що використовується для запитів – Transact-SQL, створена спільно Microsoft та Sybase [2]. Transact-SQL є реалізацією стандарту ANSI / ISO щодо структурованої мови запитів SQL із розширеннями. Використовується як для невеликих і середніх за розміром баз даних, так і для великих баз даних масштабу підприємства. Багато років вдало конкурує з іншими системами керування базами даних.

Користувацький інтерфейс та бібліотека класів створені у середовищі розробки Microsoft Visual Studio Community [2]. База даних додатку створена за допомогою СУБД Microsoft SQL Server Management Studio [2].

Оскільки одним із завдань курсової роботи було створення об'єктно-орієнтованої моделі предметної галузі, то для розробки додатку, за погодженням з викладачем, використано мову програмування C#.

Мова C# визнана однією з безумовних лідерів серед об'єктно-орієнтованих мов програмування, таких як PHP, C++, Python, Java та багато інших [4].

Народження ООП відкрило програмістам можливість працювати з програмами більшого обсягу, оскільки полегшувала промислове програмування, і також створення різних додатків.

ООП, або об'єктно-орієнтоване програмування – це одна із основних парадигм програмування, котра бачить програму, як множину об'єктів, що взаємодіють між собою [5]. ООП містить у собі чотири основних механізми-принципи:

- абстракція – це суттєві характеристики якогось об'єкту, які відділяють його від решти видів об'єктів, і таким же чином визначають особливості цього об'єкту з точки зору подальшого розгляду та аналізу. Вона зосереджується на суттєвих з погляду спостерігача характеристиках об'єкта;
- інкапсуляція – це властивість системи, яка дозволяє об'єднати дані і методи, які працюють з ними в класі, і приховати деталі реалізації від користувачів. Напрямую доступ до стану об'єкту буде заборонено, і ззовні з ним можна буде взаємодіяти тільки за допомогою інтерфейсу. Програмісти зможуть як завгодно редагувати всі закриті елементи, не переживаючи, що хтось їх буде використовувати у своїх програмах;
- наслідування – це одна із найважливіших абстракцій, яка дає здібність об'єкту або класу успадковувати ті чи інші методи або властивості від іншого класу. Новий клас(дочірній), який створений на основі існуючого класу, може отримати властивості та поведінку від батьківського класу(існуючого), а також доповнити їх своїми ж власними. Множинне успадкування можна втілити у життя за допомогою інтерфейсів;
- поліморфізм – реалізація завдань, які мають ідентичну ідею, різними способами. Його великим привілеєм є те, що він допомагає зменшувати складність програми, дозволяючи при цьому використання того ж інтерфейсу для завдання єдиного класу дій. Вибір же конкретної дії, в залежності від ситуації, покладається на компілятор.

Користувачський інтерфейс складається з наступних класів:

Частковий клас `LoginForm`, який успадковується від стандартного класу `Form`, або, як в цьому застосунку, від його вдосконаленої модифікації `MetroFramework.Forms.MetroForm`. Клас `LoginForm` містить поля, які описують об'єкти інших класів, а саме: часткового класу `TaskManagingForm`, який також є частиною інтерфейсу користувача, класу `DbContext`, який імплементує інтерфейс `IDbContext`, класу `Employee` та класу `EmployeeOperations`, які є частинами шару логіки додатку. Також клас `LoginForm` містить конструктор, який виконує ініціалізацію компонентів, та метод обробки дії користувача – натискання на кнопку `Login`.

Частковий клас `TaskManagingForm`, який також успадковується від вдосконаленої модифікації стандартного класу `Form` – `MetroFramework.Forms.MetroForm`. Клас `TaskManagingForm` містить поля, які описують об'єкти інших класів, а саме: класу `DbContext`, який імплементує інтерфейс `IDbContext`, класу `EmployeeOperations`, класу `TaskState` та класу `EmployeeTask`, які є частинами шару логіки додатку. Також клас `TaskManagingForm` містить конструктор, який виконує ініціалізацію компонентів, метод обробки події – завантаження форми та методи обробки дій користувача – натискання на кнопки `Add`, `Edit`, `Delete`, `Save`.

Всі події в класах користувачького інтерфейсу оброблені з застосуванням блоку `try catch` для запобігання виникненню необроблених виняткових ситуацій. Логіка додатку реалізована трьома проектами типу `Class Library`, кожен з яких виконує окремі функції.

Перший з них – `ModelLibrary`, що містить класи `Employee` та `EmployeeTask`, які є моделями таблиць бази даних, та клас `TaskState`, який визначає поточні стани завдань.

Клас `Employee` містить поля, які відповідають стовпцям таблиці `Employees` а саме: `Id`, `Name`, `Role`, `Password`.



Клас `EmployeeTask` містить поля, які відповідають стовпцям таблиці `TasksTable`, а саме: `Id`, `Title`, `Status`, `Priority`, `Author`, `Executor`, `Estimate`, `Description`.

Класи `Employee` та `EmployeeTask` успадковуються від стандартного класу `String`, який містить методи роботи з типом даних `string`, а клас `TaskState`, який містить тип даних `enum`, – від стандартного класу `Enum`, який є базовим класом всіх типів перерахування.

Другий – `BusinessLibrary`, що містить класи `EmployeeOperations` та `EmployeeTaskOperations`, які містять методи роботи з моделями, описаними у `ModelLibrary`.

Клас `EmployeeOperations` містить об'єкт інтерфейсу `IDbContext`, конструктор, який ініціює цей об'єкт, та метод `Login`, який приймає об'єкт класу `Employee` і повертає дане типу `string`, яке є назвою ролі співробітника в проекті.

Клас `EmployeeTaskOperations` також містить об'єкт інтерфейсу `IDbContext`, конструктор, який ініціює цей об'єкт, та методи `Delete`, `GetAll`, `Insert`, `Update`, `Save`. Метод `Delete` приймає значення типу `int`, яке являє собою `Id` завдання, яке потрібно видалити, і повертає значення типу `bool`, яке означає вдале або невдале видалення. Метод `GetAll` повертає значення типу `List<EmployeeTask>`, яке є списком завдань, що містяться в базі даних. Метод `Insert` приймає об'єкт класу `EmployeeTask` та повертає об'єкт типу `EmployeeTask`, який був доданий до бази даних. Метод `Update` також приймає об'єкт класу `EmployeeTask` та повертає об'єкт типу `EmployeeTask`, який був відредагований у базі даних. Метод `Save` приймає об'єкти класів `EmployeeTask` та `TaskState` і повертає об'єкт типу `EmployeeTask`, який був збережений в базі даних.

Третій – `DataAccessLibrary`, який містить інтерфейс `IDbContext`, клас `DbContext`, який реалізує цей інтерфейс, та клас `Config`, який при виклику передає `ConnectionString` для доступу до бази даних. Інтерфейс `IDbContext` створений як доповнення інтерфейсу `IDbConnection`, який зазвичай використовується в подібних застосунках, і містить методи, які відсутні у

інтерфейсі IDbConnection, але потрібні для реалізації та подальшого тестування функціоналу додатку. Клас DbContext містить об'єкт інтерфейсу IDbConnection, конструктор, який ініціює цей об'єкт і надає йому значення об'єкту класу SqlConnection, який приймає ConnectionString від класу Config. Клас DbContext реалізує методи інтерфейсу IDbContext: ExecuteQuery, ExecuteProcedure, ExecuteScalar, кожен з яких приймає запит до бази даних типу string, список параметрів запиту типу object та повертає результат виконання запису.

Нижче наведена діаграма класів додатку (Рисунок 2.1).

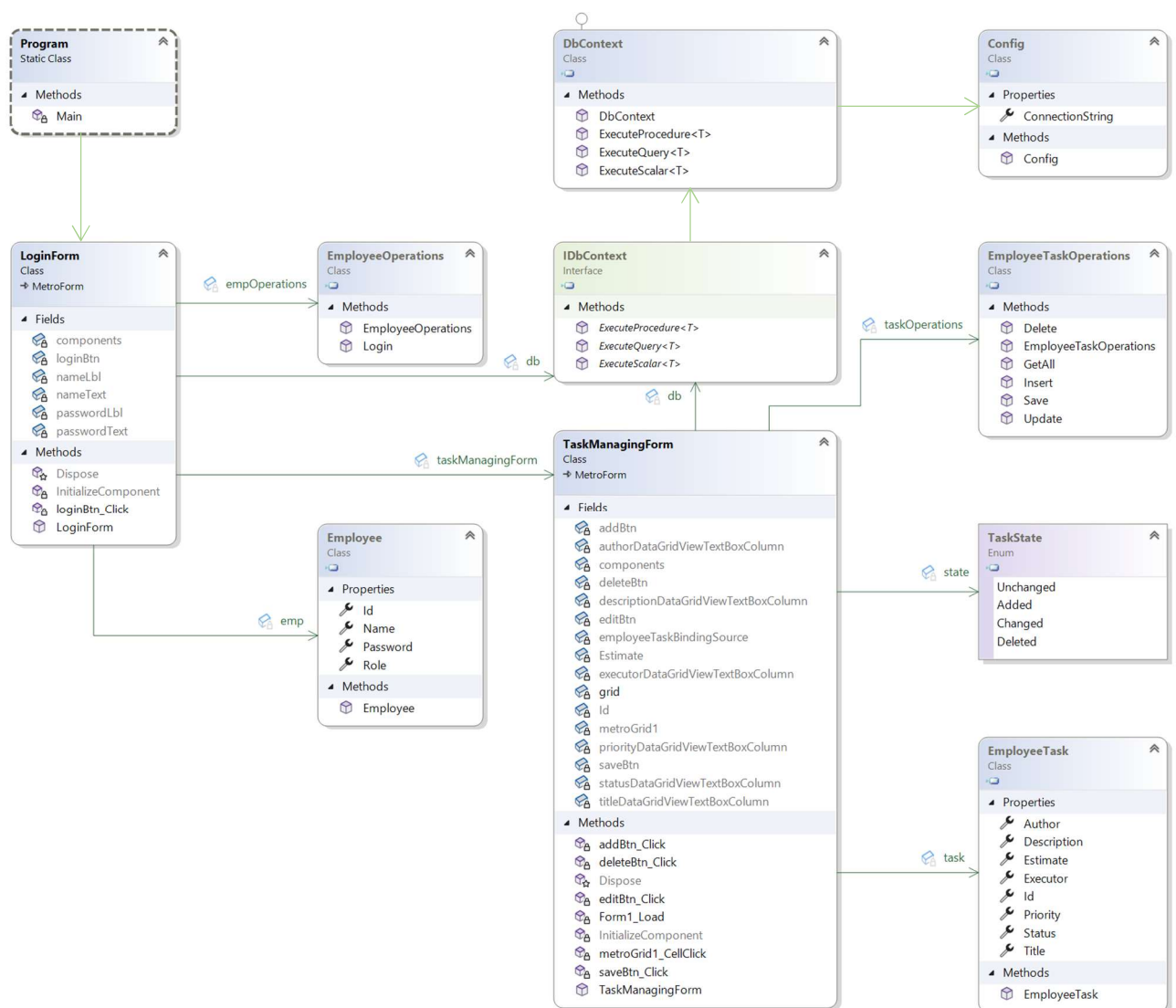


Рисунок 2.1

UML (Unified Modeling Language) — уніфікована мова моделювання, що використовується розробниками програмного забезпечення для візуалізації процесів та роботи систем [6].

Це не мова програмування, скоріше набір правил та стандартів для створення діаграм. Вони дозволяють розробникам програмного забезпечення та інженерам «говорити однією мовою», не заглиблюючись у фактичний код свого продукту. Складання діаграм за допомогою UML — це чудовий спосіб допомогти іншим швидко зрозуміти складну ідею чи структуру. База даних додатку має назву TaskTrackerDB і містить таблиці Employees, створену на основі класу Employee та TasksTable, створену на основі класу EmployeeTask (Рисунок 2.2).

Employees	
	Name
	Role
🔑	Id
	Password

TasksTable	
🔑	Id
	Title
	Status
	Priority
	Executor
	Author
	Description
	Estimate

Рисунок 2.2

Також для роботи з додатком були створені збережені процедури `dbo.TasksTable_AddTask` для додавання нових завдань до бази даних та `dbo.TasksTable_UpdateTask` для редагування існуючих процедур в базі даних.

Процедура `dbo.TasksTable_AddTask` виглядає наступним чином (Рисунок 2.3):

```

USE [TaskTrackerDB]
GO

DECLARE @RC int
DECLARE @Title nvarchar(50)
DECLARE @Status nvarchar(50)
DECLARE @Priority nvarchar(50)
DECLARE @Author nvarchar(50)
DECLARE @Executor nvarchar(50)
DECLARE @Estimate nvarchar(50)
DECLARE @Description nvarchar(50)

-- TODO: Set parameter values here.

EXECUTE @RC = [dbo].[TasksTable_AddTask]
    @Title
    ,@Status
    ,@Priority
    ,@Author
    ,@Executor
    ,@Estimate
    ,@Description
GO

```

Рисунок 2.3

Процедура `dbo.TasksTable_UpdateTask` виглядає наступним чином (Рисунок 2.4):

```

USE [TaskTrackerDB]
GO

DECLARE @RC int
DECLARE @Title nvarchar(50)
DECLARE @Status nvarchar(50)
DECLARE @Priority nvarchar(50)
DECLARE @Author nvarchar(50)
DECLARE @Executor nvarchar(50)
DECLARE @Estimate nvarchar(50)
DECLARE @Description nvarchar(50)

-- TODO: Set parameter values here.

EXECUTE @RC = [dbo].[TasksTable_UpdateTask]
    @Title
    ,@Status
    ,@Priority
    ,@Author
    ,@Executor
    ,@Estimate
    ,@Description
GO

```

Рисунок 2.4

Збережені процедури доцільно використовувати при роботі з даними для уникнення багаторазових повторів одних і тих самих запитів [7].

За допомогою форми управління додатком, яка має назву TaskManagingForm, завдання можна створювати та додавати до бази даних, отримувати з бази даних, редагувати та видаляти із збереженням відповідних змін у базі даних. Такий набір операцій має назву CRUD – це скорочення від англійських слів, що позначають ці чотири операції:

- create (створювати);
- read (читати);
- update (редагувати);
- delete (видаляти).

Операції CRUD становлять основу для багатьох застосунків та систем управління інформацією, даючи змогу користувачам взаємодіяти з даними та змінювати їх відповідно до своїх потреб [8].

Операції CRUD використовуються в системах постійного зберігання, тобто дані не зникають навіть після вимкнення системи. Вони відрізняються від операцій з даними, що зберігаються в енергозалежному сховищі, як-от оперативна пам'ять або кеш-файли.

Користувачі можуть викликати чотири основні функції CRUD для виконання різних типів операцій над вибраними даними в базі даних. Це можна зробити за допомогою коду або через графічний інтерфейс користувача.

При запуску програми користувач бачить перед собою форму для авторизації в програмі, в яку він повинен ввести свій логін та пароль.

### **2.3 Сценарій роботи додатку.**

При запуску програми користувач побачить перед собою форму входу, до якої він повинен ввести своє ім'я та пароль (Рисунок 2.5):

LoginForm

Name

Password

Login

Рисунок 2.5

При успішній реєстрації перед користувачем з'являється діалогове вікно, яке показує його роль в проекті (Рисунок 2.6).

LoginForm

Name

Password

Developer

OK

Login

Рисунок 2.6

Якщо такий користувач не зареєстрований або у випадку введення некоректних даних користувач побачить діалогове вікно з повідомленням про некоректні дані (Рисунок 2.7).

LoginForm

Name

Password

Incorrect name or password

OK

Рисунок 2.7

Після успішної реєстрації перед користувачем з'явиться форма управління додатком, в якій показані всі наявні завдання для поточного проекту. Завдання можна додавати, редагувати та видаляти (Рисунок 2.8).

Task Managing Form

Id	Title	Status	Priority	Author	Executor	Description	Estimate
1	UML diagram	Completed	Normal	Mary	Jack	Make UML Diagram for new app	20.05.2023
2	Database	Completed	Normal	Mary	Jane	Create Database for new application	20.05.2023
3	ScreenForms	InProgress	Normal	Mary	Bob	Create ScreenForms for new application	25.05.2023
5	Connection	InProgress	Normal	Mary	Jane	Connect Database with ScreenForms	30.05.2023
6	Test	InProgress	Normal	Mary	Ann	Test New Screen Forms With Database	10.06.2023
9	Update	InProgress	Normal	Mary	Bob	Update User Interface	15.06.2023
16	Preview	Completed	High	Mary	Mary	Review application with customer	25.06.2023
17	Updates	InProgress	Normal	Mary	Bob	Implement customer's edits	30.06.2023
24	Update Database	In Progress	Normal	Mary	Jane	Add new records to database	30.06.2023
26	Test DB	New	Normal	Mary	Ann	Test database updates	10.07.2023
28	Test UI	New	Normal	Mary	Ann	Test UI updates	15.07.2023

Add Edit Delete Save

Рисунок 2.8

Внесені зміни зберігаються в базі даних (Рисунок 2.9).

Results		Messages						
	Id	Title	Status	Priority	Executor	Author	Description	Estimate
1	1	UML diagram	Completed	Normal	Jack	Mary	Make UML Diagram for new app	20.05.2023
2	2	Database	Completed	Normal	Jane	Mary	Create Database for new application	20.05.2023
3	3	ScreenForms	InProgress	Normal	Bob	Mary	Create ScreenForms for new application	25.05.2023
4	5	Connection	InProgress	Normal	Jane	Mary	Connect Database with ScreenForms	30.05.2023
5	6	Test	InProgress	Normal	Ann	Mary	Test New Screen Forms With Database	10.06.2023
6	9	Update	InProgress	Normal	Bob	Mary	Update User Interface	15.06.2023
7	16	Preview	Completed	High	Mary	Mary	Review application with customer	25.06.2023
8	17	Updates	InProgress	Normal	Bob	Mary	Implement customer's edits	30.06.2023
9	24	Update Database	In Progress	Normal	Jane	Mary	Add new records to database	30.06.2023
10	26	Test DB	New	Normal	Ann	Mary	Test database updates	10.07.2023
11	28	Test UI	New	Normal	Ann	Mary	Test UI updates	15.07.2023

Рисунок 2.9

### 3 ПРОГРАМУВАННЯ ДОДАТКУ.

В цьому розділі наведені лістинги програмного коду додатку. Код містить коментарі, які роблять його зрозумілим і читабельним.

#### 3.1 Лістинг програмного коду користувацького інтерфейсу

##### EmployeeTasksManagingApp

###### 3.1.1 Клас Program:

```
using EmployeeTasksManagingApp;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace ModelLibrary
{
    internal static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new LoginForm());
        }
    }
}
```

###### 3.1.2 Клас LoginForm:

```
using System.ComponentModel;
using System.Data;
using System.Data.SqlClient;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace EmployeeTasksManagingApp
{
    public partial class LoginForm : MetroFramework.Forms.MetroForm
    {
        // Create an instance of the database context
        IDbContext db = new DbContext();
    }
}
```



```

// Create an instance of the Employee class
Employee emp = new Employee();

// Create an instance of the EmployeeOperations class
EmployeeOperations empOperations;

// Create an instance of the TaskManagingForm class
TaskManagingForm taskManagingForm = new TaskManagingForm();

public LoginForm()
{
    InitializeComponent();

    // Initialize the EmployeeOperations instance with the database context
    empOperations = new EmployeeOperations(db);
}

private void loginBtn_Click(object sender, EventArgs e)
{
    try
    {
        // Set the employee's name and password based on the input fields
        emp.Name = nameText.Text;
        emp.Password = passwordText.Text;

        // Call the Login method of the EmployeeOperations instance
        var result = empOperations.Login(emp);

        // Check the result to determine the role of the employee
        if (result == "ProjectManager")
        {
            // Show a message box indicating the role
            MessageBox.Show("ProjectManager");

            // Hide the login form and show the task managing form
            this.Hide();
            taskManagingForm.Show();
        }
        else if (result == "Developer")
        {
            MessageBox.Show("Developer");
            this.Hide();
            taskManagingForm.Show();
        }
        else if (result == "QA")
        {
            MessageBox.Show("QA");
            this.Hide();
            taskManagingForm.Show();
        }
        else
        {
            // Show an error message for incorrect name or password
            MessageBox.Show("Incorrect name or password");
        }
    }
}

```

```

        catch (Exception ex)
        {
            // Display a MetroMessageBox with the error message
            MetroFramework.MetroMessageBox.Show(this, ex.Message, "Message", M
essageBoxButtons.OK, MessageBoxIcon.Error);
        }
    }
}

```

### 3.1.3 Клас TaskManagingForm:

```

using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;
using System.Windows.Forms;
using BusinessLibrary;
using DataAccessLibrary;
using MetroFramework.Controls;
using ModelLibrary;

namespace EmployeeTasksManagingApp
{
    public partial class TaskManagingForm : MetroFramework.Forms.MetroForm
    {
        IDbContext db = new DbContext();
        EmployeeTaskOperations taskOperations;
        TaskState state = TaskState.Unchanged;
        EmployeeTask task = new EmployeeTask();

        MetroGrid grid = new MetroGrid();

        public TaskManagingForm()
        {
            InitializeComponent();
            taskOperations = new EmployeeTaskOperations(db);
        }

        // Form load event handler
        private void Form1_Load(object sender, EventArgs e)
        {
            try
            {
                // Load all employee tasks into the binding source
                employeeTaskBindingSource.DataSource = taskOperations.GetAll();
                task = employeeTaskBindingSource.Current as EmployeeTask;
            }
            catch (Exception ex)
            {
                MetroFramework.MetroMessageBox.Show(this, ex.Message, "Message", M
essageBoxButtons.OK, MessageBoxIcon.Error);
            }
        }

        // Add button click event handler
    }
}

```

```

private void addBtn_Click(object sender, EventArgs e)
{
    state = TaskState.Added;
    employeeTaskBindingSource.Add(task);
}

// Edit button click event handler
private void editBtn_Click(object sender, EventArgs e)
{
    state = TaskState.Changed;
}

// MetroGrid cell click event handler
private void metroGrid1_CellClick(object sender, DataGridViewCellEventArgs
e)
{
    try
    {
        // Get the currently selected employee task
        task = employeeTaskBindingSource.Current as EmployeeTask;
    }
    catch (Exception ex)
    {
        MetroFramework.MetroMessageBox.Show(this, ex.Message, "Message", M
essageBoxButtons.OK, MessageBoxIcon.Error);
    }

    // Delete button click event handler
    private void deleteBtn_Click(object sender, EventArgs e)
    {
        state = TaskState.Deleted;
        if(MetroFramework.MetroMessageBox.Show(this, "Delete task?", "Message"
, MessageBoxButtons.YesNo, MessageBoxIcon.Question) == DialogResult.Yes)
        {
            try
            {
                // Delete the selected employee task
                task = employeeTaskBindingSource.Current as EmployeeTask;
                if(task != null)
                {
                    bool result = taskOperations.Delete(task.Id);
                    if(result)
                    {
                        // Remove the task from the binding source and refresh
the grid
                        employeeTaskBindingSource.RemoveCurrent();
                        grid.Refresh();
                        state = TaskState.Unchanged;
                    }
                }
            }
            catch (Exception ex)
            {
                MetroFramework.MetroMessageBox.Show(this, ex.Message, "Message
", MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
        }
    }
}

```

```

// Save button click event handler
private void saveBtn_Click(object sender, EventArgs e)
{
    try
    {
        // End editing on the binding source and save the changes
        employeeTaskBindingSource.EndEdit();
        task = employeeTaskBindingSource.Current as EmployeeTask;
        if (task != null)
        {
            taskOperations.Save(task, state);
            grid.Refresh();
            state = TaskState.Unchanged;
        }
    }
    catch (Exception ex)
    {
        MetroFramework.MetroMessageBox.Show(this, ex.Message, "Message", M
essageBoxButtons.OK, MessageBoxIcon.Error);
    }
}
}
}

```

## 3.2 Лістинг програмного коду ModelLibrary

### 3.2.1 Клас Employee:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ModelLibrary
{
    public class Employee
    {
        public int Id { get; set; } // Unique identifier for the employee
        public string Name { get; set; } // Name of the employee
        public string Role { get; set; } // Role or position of the employee
        public string Password { get; set; } // Password associated with the emplo
        yee
    }
}

```

### 3.2.2 Клас EmployeeTask:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

```

```

namespace ModelLibrary
{
    public class EmployeeTask
    {
        public int Id { get; set; } // Unique identifier for the task
        public string Title { get; set; } // Title or name of the task
        public string Status { get; set; } // Status of the task (e.g., new, in progress, completed)
        public string Priority { get; set; } // Priority level of the task (e.g., high, medium, low)
        public string Author { get; set; } // Author or creator of the task
        public string Executor { get; set; } // Employee assigned to execute or work on the task
        public string Estimate { get; set; } // Estimated time required for the task
        public string Description { get; set; } // Detailed description or notes about the task
    }
}

```

### 3.2.3 Enum TaskState:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ModelLibrary
{
    public enum TaskState
    {
        Unchanged, // The task is in its original state, without any changes
        Added,     // The task is newly added and pending further action
        Changed,   // The task has been modified
        Deleted    // The task has been marked for deletion
    }
}

```

## 3.3 Лістинг програмного коду DataAccessLibrary

### 3.3.1 Клас Config:

```

using System;
using System.Collections.Generic;
using System.Configuration;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DataAccessLibrary
{
    public class Config

```

```

{
    /// <summary>
    /// Retrieves the connection string from the configuration file.
    /// </summary>
    public static string ConnectionString
    {
        get
        {
            return ConfigurationManager.ConnectionStrings
                ["TaskTrackerDBConnectionString"].ConnectionString;
        }
    }
}

```

### 3.3.2 Интерфейс IDbContext:

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Threading.Tasks;

namespace DataAccessLibrary
{
    public interface IDbContext
    {
        /// <summary>
        /// Executes the SQL query and returns a single value of the specified type.
        /// </summary>
        /// <typeparam name="T">The type of the value to return.</typeparam>
        /// <param name="sql">The SQL query to execute.</param>
        /// <param name="parameters">Optional parameters for the SQL query.</param>
        >
        /// <returns>The single value of the specified type.</returns>
        T ExecuteScalar<T>(string sql, object parameters = null);

        /// <summary>
        /// Executes the stored procedure and returns a single value of the specified type.
        /// </summary>
        /// <typeparam name="T">The type of the value to return.</typeparam>
        /// <param name="sql">The stored procedure name to execute.</param>
        /// <param name="parameters">Optional parameters for the stored procedure.
        </param>
        /// <returns>The single value of the specified type.</returns>
        T ExecuteProcedure<T>(string sql, object parameters = null);

        /// <summary>
        /// Executes the SQL query and returns a list of results of the specified type.
        /// </summary>
        /// <typeparam name="T">The type of the results to return.</typeparam>
        /// <param name="sql">The SQL query to execute.</param>
        /// <param name="parameters">Optional parameters for the SQL query.</param>
        >
        /// <returns>A list of results of the specified type.</returns>
        List<T> ExecuteQuery<T>(string sql, object parameters = null);
    }
}

```

```
}
```

### 3.3.3 Клас *DbContext*:

```
using Dapper;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Linq;

namespace DataAccessLibrary
{
    public class DbContext : IDbContext //Implements interface
    {
        private readonly IDbConnection connection;

        public DbContext()
        {
            connection = new SqlConnection(Config.ConnectionString);
        }

        public List<T> ExecuteQuery<T>(string sql, object parameters)
        {
            if (connection.State != ConnectionState.Open)
                connection.Open();
            return connection.Query<T>(sql, parameters, commandType: CommandType.T
ext).ToList();
        }

        public T ExecuteProcedure<T>(string sql, object parameters = null)
        {
            if (connection.State != ConnectionState.Open)
                connection.Open();
            return connection.ExecuteScalar<T>(sql, parameters, commandType: Comma
ndType.StoredProcedure);
        }

        public T ExecuteScalar<T>(string sql, object parameters)
        {
            if (connection.State != ConnectionState.Open)
                connection.Open();
            return connection.ExecuteScalar<T>(sql, parameters, commandType: Comma
ndType.Text);
        }
    }
}
```

## 3.4 Лістинг програмного коду Business Library

### 3.4.1 Клас *EmployeeOperations*:

```
using System.Data.SqlClient;
using System.Linq;
using System.Runtime.Remoting.Contexts;
```

```

using System.Text;
using System.Threading.Tasks;

namespace BusinessLibrary
{
    public class EmployeeOperations
    {
        IDbContext db;

        /// <summary>
        /// Initializes a new instance of the EmployeeOperations class.
        /// </summary>
        /// <param name="db">The database context.</param>
        public EmployeeOperations(IDbContext db)
        {
            this.db = db;
        }

        /// <summary>
        /// Performs a login operation for the given employee.
        /// </summary>
        /// <param name="emp">The employee object containing the name and password
        .</param>
        /// <returns>The role of the employee if the login is successful; otherwis
        e, null.</returns>
        public string Login(Employee emp)
        {
            var cmd = $"SELECT Role FROM dbo.Employees WHERE Name = '{emp.Name}' A
ND Password = '{emp.Password}'";
            var result = db.ExecuteScalar<string>(cmd);
            return result;
        }
    }
}

```

### 3.4.2 Класс EmployeeTaskOperations:

```

using DataAccessLibrary;
using System;
using System.Collections.Generic;
using System.Data.SqlClient;
using System.Data;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Dapper;
using ModelLibrary;
using System.Configuration;

namespace BusinessLibrary
{
    public class EmployeeTaskOperations
    {
        IDbContext db;

        /// <summary>
        /// Initializes a new instance of the EmployeeTaskOperations class.

```



```

    /// </summary>
    /// <param name="db">The database context.</param>
    public EmployeeTaskOperations(IDbContext db)
    {
        this.db = db;
    }

    /// <summary>
    /// Deletes a task with the specified ID from the TasksTable.
    /// </summary>
    /// <param name="id">The ID of the task to delete.</param>
    /// <returns>True if the task is deleted successfully; otherwise, false.</
returns>
    public bool Delete(int id)
    {
        return db.ExecuteScalar<int>("delete from dbo.TasksTable where Id = @i
d", new { id }) == 1;
    }

    /// <summary>
    /// Retrieves all employee tasks from the TasksTable.
    /// </summary>
    /// <returns>A list of EmployeeTask objects representing the tasks.</retur
ns>
    public List<EmployeeTask> GetAll()
    {
        return db.ExecuteQuery<EmployeeTask>("select Id, Title, Status, Priori
ty, Author, Executor, Estimate, Description from dbo.TasksTable");
    }

    /// <summary>
    /// Inserts a new task into the TasksTable.
    /// </summary>
    /// <param name="task">The EmployeeTask object representing the task to in
sert.</param>
    /// <returns>The inserted EmployeeTask object.</returns>
    public EmployeeTask Insert(EmployeeTask task)
    {
        db.ExecuteProcedure<int>("dbo.TasksTable_AddTask", new { task.Title, t
ask.Status, task.Priority, task.Author, task.Executor, task.Estimate, task.Descr
ption });

        return task;
    }

    /// <summary>
    /// Updates an existing task in the TasksTable.
    /// </summary>
    /// <param name="task">The EmployeeTask object representing the task to up
date.</param>
    /// <returns>The updated EmployeeTask object.</returns>
    public EmployeeTask Update(EmployeeTask task)
    {
        db.ExecuteProcedure<int>("dbo.TasksTable_UpdateTask", new { task.Title
, task.Status, task.Priority, task.Author, task.Executor, task.Estimate, task.Desc
ription });

        return task;
    }
}

```

```

    /// <summary>
    /// Saves a task based on its state (Added or Changed).
    /// </summary>
    /// <param name="task">The EmployeeTask object representing the task to sa
ve.</param>
    /// <param name="state">The state of the task (Added or Changed).</param>
    /// <returns>The saved EmployeeTask object.</returns>
    public EmployeeTask Save(EmployeeTask task, TaskState state)
    {
        if (state == TaskState.Added)
        {
            task = Insert(task);
        }
        else if (state == TaskState.Changed)
        {
            task = Update(task);
        }
        return task;
    }
}

```

#### 4 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

При написанні різноманітного типу програм завжди постає проблема тестування створеного програмного продукту. Це важливий етап, на який розробники витрачають велику кількість часу. Тому вони намагаються звести кількість ручного тестування до мінімуму та використовувати автоматизацію тестування.

Тестування – це процес перевірки відповідності між реальною і очікуваною поведінкою програми, здійснювана на кінцевому наборі тестів, вибраному певним чином.

Додаток створено за допомогою тришарової архітектури: користувач виконує операції з даними через користувацький інтерфейс, цей запит обробляється шаром логіки додатку і результат зберігається в базі даних.

Тестування додатку проводилося за методикою причина / наслідок. Введення комбінацій умов (причин), для отримання відповіді від системи (наслідок). Наприклад, треба перевірити можливість додавати завдання, використовуючи певну екранну форму. Для цього необхідно ввести дані в кілька полів, таких як «Назва», «Статус», «Пріоритет», «Автор», «Виконавець», «Час виконання», «Опис», а потім, натиснути кнопку «Додати» – це «Причина». Після натискання кнопки «Додати», програма додає завдання в базу даних і показує його на екрані – це «Наслідок».

Для тестування додатку було застосоване автоматизоване функціональне модульне тестування (Unit Testing).

Компонентне (модульне) тестування перевіряє функціональність і шукає дефекти в частинах додатка, які доступні і можуть бути протестовані по-окремо (модулі програм, об'єкти, класи, функції і т.д) [9].

Якщо подивитися на Unit тест, то можна чітко виділити 3 частини коду:

- Arrange (налаштування) – в цьому блоці коду ми налаштовуємо тестове оточення тестованого юніта;

- Act – виконання або виклик тестованого сценарію;
- Assert – перевірка того, що тестований виклик поводитьсь певним чином.

Функціональне тестування (functional testing) розглядає наперед вказану поведінку і ґрунтується на аналізі специфікацій функціональності компоненту.

Автоматизоване тестування передбачає використання спеціального програмного забезпечення для контролю виконання тестів і порівняння очікуваного фактичного результату роботи програми.

Тестовий проект було створено за допомогою шаблону Visual Studio Community, який називається Unit Test Project. Тестовий проект отримав назву BusinessLibraryTests.

Проект складається з двох тестових класів, EmployeeOperationsTest та EmployeeTaskOPTests, які тестують функціонал відповідних класів у BusinessLibrary: EmployeeOperations та EmployeeTaskOperations.

Клас EmployeeOperationsTest містить об'єкт класу EmployeeOperations та імітацію об'єкта IDbContext - Mock<IDbContext>, метод Setup, який ініціалізує ці об'єкти та тестовий метод Login\_Test, який перевіряє роботу методу Login з класу EmployeeOperations з використанням спеціально створених тестових даних.

Клас EmployeeTaskOPTests містить об'єкт класу EmployeeTaskOperations та імітацію об'єкта IDbContext - Mock<IDbContext>, метод Setup, який ініціалізує ці об'єкти та методи Delete\_Test, GetAll\_Test, Insert\_Test, Update\_Test, SaveStateAdded\_Test та SaveStateChanged\_Test, які перевіряють роботу відповідних методів класу EmployeeTaskOperations з використанням спеціально створених тестових даних.

Оскільки функціональність тестованих класів пов'язана з базою даних то потрібно було створити функцію – імітацію виконання запитів до бази даних. Для цього використано фреймворк Moq, який призначений для імітації об'єктів.

У цьому випадку імітується функціональність бази даних. Для тестування були створені набори тестових даних, які розміщені в коді тестових класів і за структурою повторюють дані, що містяться в базі даних.

## 4.1 Лістинг програмного коду BusinessLibraryTests

### 4.1.1 Клас *EmployeeOperationsTest*:

```
using BusinessLibrary;
using DataAccessLibrary;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using ModelLibrary;
using Moq;
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BusinessLibraryTests
{
    [TestClass]
    public class EmployeeOperationsTest
    {
        private Mock<IDbContext> mockDb;
        private EmployeeOperations employeeOperations;

        [TestInitialize]
        public void Setup()
        {
            // Setting up the mock objects and initializing the class under test
            mockDb = new Mock<IDbContext>();
            employeeOperations = new EmployeeOperations(mockDb.Object);
        }

        [TestMethod]
        public void Login_Test()
        {
            // Arrange
            var expectedRole = "Developer";
            var emp = new Employee { Name = "John", Password = "password" };

            // Setting up the mock behavior for the database context's ExecuteScalar method
            mockDb.Setup(x => x.ExecuteScalar<string>(It.IsAny<string>(), It.IsAny<object>()))
                .Returns(expectedRole);

            // Act
            var actualRole = employeeOperations.Login(emp);
        }
    }
}
```

```

        // Assert
        Assert.AreEqual(expectedRole, actualRole);

        // Verifying that the ExecuteScalar method was called exactly once
        mockDb.Verify(x => x.ExecuteScalar<string>(It.IsAny<string>()), It.IsAn
y<object>()), Times.Once);
    }
}

```

#### 4.1.2 Класс EmployeeTaskOPTests:

```

using BusinessLibrary;
using Dapper;
using DataAccessLibrary;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using ModelLibrary;
using Moq;
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BusinessLibraryTests
{
    [TestClass]
    public class EmployeeTaskOPTests
    {
        private Mock<IDbContext> mockDb;
        private EmployeeTaskOperations employeeTaskOperations;

        [TestInitialize]
        public void Setup()
        {
            // Setting up the mock objects and initializing the class under test
            mockDb = new Mock<IDbContext>();
            employeeTaskOperations = new EmployeeTaskOperations(mockDb.Object);
        }

        [TestMethod]
        public void Delete_Test()
        {
            // Arrange
            int expectedId = 1;

            // Setting up the mock behavior for the database context's ExecuteScalar
ar method
            mockDb.Setup(
                x => x.ExecuteScalar<int>(
                    It.Is<string>(sql => sql == "delete from dbo.TasksTable where Id =
@id"),
                    It.IsAny<object>()))
                .Returns(1);

            // Act
            bool result = employeeTaskOperations.Delete(expectedId);

```

```

        // Assert
        Assert.IsTrue(result);
        mockDb.VerifyAll();
    }

    [TestMethod]
    public void GetAll_Test()
    {
        //Arrange
        List<EmployeeTask> expectedTasks = new List<EmployeeTask> {

            // Creating a list of expected tasks
            new EmployeeTask
            {
                Title = "Create",
                Status = "New",
                Priority = "Normal",
                Author = "Mike",
                Executor = "Sam",
                Estimate = "20.05.2023",
                Description = "Description",

            },
            new EmployeeTask
            {
                Title = "Read",
                Status = "New",
                Priority = "Normal",
                Author = "Mike",
                Executor = "Mark",
                Estimate = "25.05.2023",
                Description = "Description",

            },
            new EmployeeTask
            {
                Title = "Update",
                Status = "New",
                Priority = "Normal",
                Author = "Mike",
                Executor = "James",
                Estimate = "05.06.2023",
                Description = "Description",

            },
            new EmployeeTask
            {
                Title = "Delete",
                Status = "New",
                Priority = "Normal",
                Author = "Mike",
                Executor = "Dan",
                Estimate = "10.06.2023",
                Description = "Description"

            }
        };

        // Setting up the mock behavior for the database context's ExecuteQuer
y method
        mockDb.Setup(x => x.ExecuteQuery<EmployeeTask>("select Id, Title, Stat
us, Priority, Author, Executor, Estimate, Description from dbo.TasksTable",

```

```

        It.IsAny<object>()))
        .Returns(expectedTasks);

//Act
List<EmployeeTask> actualTasks = employeeTaskOperations.GetAll();

//Assert
CollectionAssert.AreEqual(expectedTasks, actualTasks);
mockDb.VerifyAll();
}

[TestMethod]
public void Insert_Test()
{
    //Arrange
    EmployeeTask expectedTask = new EmployeeTask {
        Title = "Test",
        Status = "New",
        Priority = "Normal",
        Author = "Nick",
        Executor = "James",
        Estimate = "30.05.2023",
        Description = "Description"
    };

    // Setting up the mock behavior for the database context's ExecuteProc
    edure method
    mockDb.Setup(x => x.ExecuteProcedure<int>("dbo.TasksTable_AddTask", It
    .IsAny<object>()))
        .Returns(1);

    // Act
    EmployeeTask actualTask = employeeTaskOperations.Insert(expectedTask);

    //Assert
    Assert.AreEqual(expectedTask, actualTask);
    mockDb.VerifyAll();
}

[TestMethod]
public void Update_Test()
{
    //Arrange
    EmployeeTask expectedTask = new EmployeeTask
    {
        Title = "Create",
        Status = "InProgress",
        Priority = "Normal",
        Author = "Mike",
        Executor = "Sam",
        Estimate = "20.05.2023",
        Description = "Description"
    };

    // Setting up the mock behavior for the database context's ExecuteProc
    edure method
    mockDb.Setup(x => x.ExecuteProcedure<int>("dbo.TasksTable_UpdateTask",
    It.IsAny<object>()))
        .Returns(1);

    //Act

```



```

        EmployeeTask actualTask = employeeTaskOperations.Update(expectedTask);

        //Assert
        Assert.AreEqual(expectedTask, actualTask);
        mockDb.VerifyAll();
    }

    [TestMethod]
    public void SaveStateAdded_Test()
    {
        //Arrange
        EmployeeTask expectedTask = new EmployeeTask
        {
            Title = "Test",
            Status = "New",
            Priority = "Normal",
            Author = "Nick",
            Executor = "James",
            Estimate = "30.05.2023",
            Description = "Description"
        };
        TaskState state = TaskState.Added;

        // Setting up the mock behavior for the database context's ExecuteProc
        edure method
        mockDb.Setup(x => x.ExecuteProcedure<int>("dbo.TasksTable_AddTask", It
        .IsAny<object>()))
            .Returns(1);

        //Act
        EmployeeTask actualTask = employeeTaskOperations.Save(expectedTask, st
        ate);

        //Assert
        Assert.AreEqual(expectedTask, actualTask);
        mockDb.VerifyAll();
    }

    [TestMethod]
    public void SaveStateChanged_Test()
    {
        //Arrange
        EmployeeTask expectedTask = new EmployeeTask
        {
            Title = "Create",
            Status = "InProgress",
            Priority = "Normal",
            Author = "Mike",
            Executor = "Sam",
            Estimate = "20.05.2023",
            Description = "Description"
        };
        TaskState state = TaskState.Changed;

        // Setting up the mock behavior for the database context's ExecuteProc
        edure method
        mockDb.Setup(x => x.ExecuteProcedure<int>("dbo.TasksTable_UpdateTask",
        It.IsAny<object>()))
            .Returns(1);

        //Act
    }

```

```

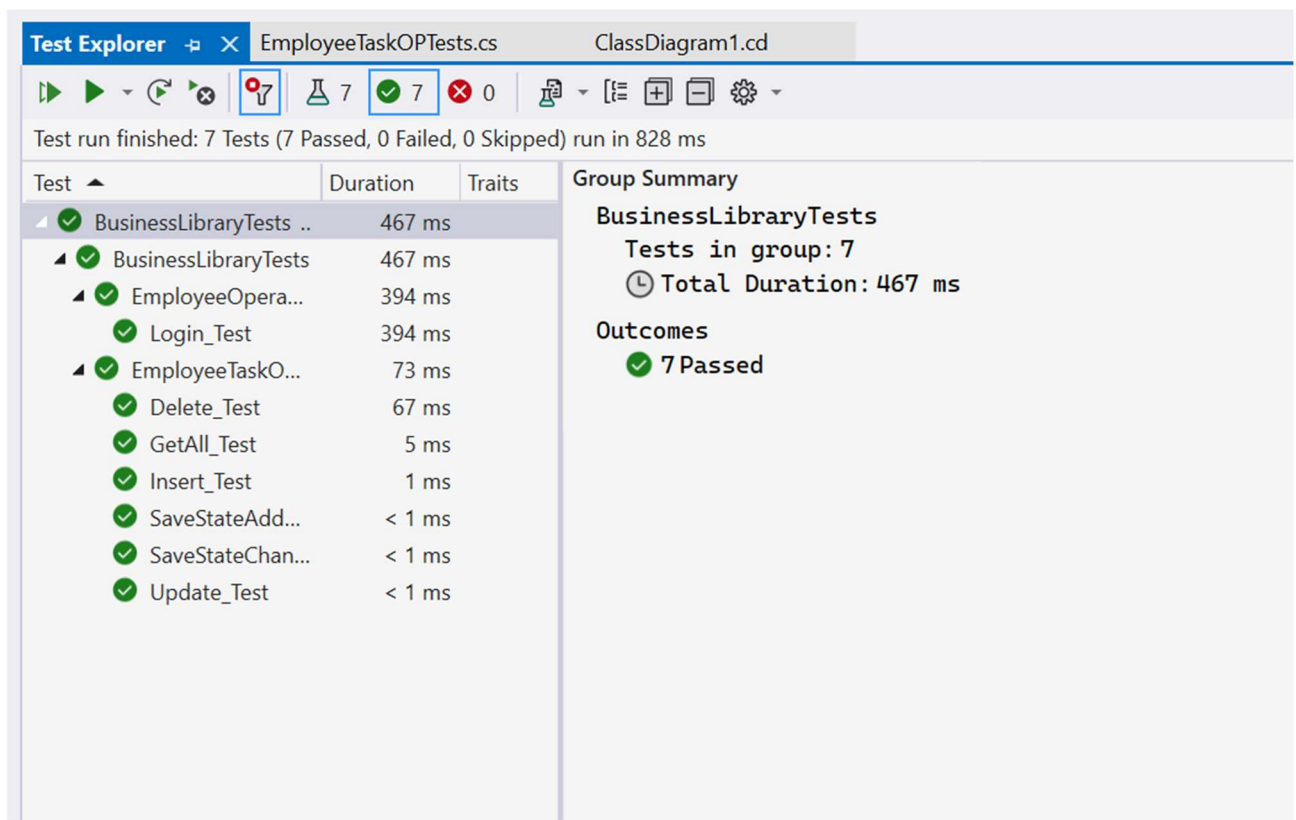
    EmployeeTask actualTask = employeeTaskOperations.Save(expectedTask, st
ate);

    //Assert
    Assert.AreEqual(expectedTask, actualTask);
    mockDb.VerifyAll();
}
}
}

```

## 4.2 Результати тестів

Нижче наведені результати тестів (Рисунок 4.1).



Test Explorer EmployeeTaskOPTests.cs ClassDiagram1.cd

Test run finished: 7 Tests (7 Passed, 0 Failed, 0 Skipped) run in 828 ms

Test	Duration	Traits
BusinessLibraryTests ..	467 ms	
BusinessLibraryTests	467 ms	
EmployeeOpera...	394 ms	
Login_Test	394 ms	
EmployeeTaskO...	73 ms	
Delete_Test	67 ms	
GetAll_Test	5 ms	
Insert_Test	1 ms	
SaveStateAdd...	< 1 ms	
SaveStateChan...	< 1 ms	
Update_Test	< 1 ms	

**Group Summary**

**BusinessLibraryTests**

Tests in group: 7

Total Duration: 467 ms

**Outcomes**

7 Passed

Рисунок 4.1

## ВИСНОВКИ

### Висновок до розділу 1

Ринок додатків для планування та контролю завдань досить насичений і конкурентний. Існує безліч різних програм та сервісів, які пропонують різні функції, інтерфейси та тарифи. Щоб створити подібну програму, потрібно не тільки мати гарні навички розробки програмного забезпечення, але й провести аналіз ринку, вивчити потреби та переваги користувачів, виділити свою цільову аудиторію та конкурентні переваги. Також потрібно бути готовим до того, що програма може не отримати достатньої уваги чи відгуків від користувачів, або зіткнутися з негативною критикою чи порівнянням з іншими продуктами.

Я вважаю, що важливо знайти баланс між контролем та довірою, між ефективністю та комфортом. Такі програми можуть бути корисними, якщо вони використовуються правильно та враховують інтереси та потреби всіх учасників процесу. Також важливо вибирати такі додатки, які підходять для конкретних завдань та цілей, а також мають зручний та зрозумілий інтерфейс.

### Висновок до розділу 2

При проектуванні додатку було проаналізовано предметну галузь та технічні вимоги. Після аналізу було вирішено розроблювати додаток, заснований на тришаровій архітектурі з застосуванням CRUD операцій для роботи з базою даних. Програмні продукти, які використовувались для розробки додатку, були обрані за принципом зручності у використанні, простоти опанування для початківця, яким є автор цієї роботи, та актуальності в наш час.

В сукупності це дало змогу побудувати простий у використанні інтуїтивно зрозумілий додаток, який може бути використаний для планування завдань та робочого часу у невеликих групах.

### **Висновок до розділу 3**

У пояснювальній записці наведені лістинги програмних кодів класів додатку. Повний вихідний код додатку та скрипт для відтворення об'єктів бази даних знаходяться за посиланням [https://github.com/granny-dev/OP2\\_KR](https://github.com/granny-dev/OP2_KR).

### **Висновок до розділу 4**

За результатами тестування (Рисунок 4.1) можна побачити, що методи, які містяться в класах, що відповідають за функціонал додатку, пройшли тестування, отже можна зробити висновок, що функціонал додатку працює правильно і додаток може бути використаний за призначенням.

## 5 БІБЛІОГРАФІЯ

- [1] D. Allen, Getting Things Done: The Art of Stress-Free Productivity, Penguin Books, March 17, 2015.
- [2] Microsoft, "Microsoft Learn".
- [3] R. Turner, SQL: 2 books in 1 - The Ultimate Beginner's & Intermediate Guide to Learn SQL Programming step by step, LIGHTNING SOURCE INC, 11 April 2020.
- [4] I. Griffiths, Programming C# 8.0: Build Cloud, Web, and Desktop Applications: Build Windows, Web, and Desktop Applications, Revised edition ed., O'Reilly Media, 7 Jan. 2020.
- [5] R. Taher, Hands-On Object-Oriented Programming with C#, Packt Publishing, February 2019.
- [6] S. W. Ambler, The Elements of UML™ 2.0 Style, Cambridge University Press, December 2010.
- [7] A. Molinaro, SQL Cookbook: Query Solutions and Techniques for Database Developers. Covers SQL Server, PostgreSQL, Oracle, MySQL, and DB2, 1st edition ed., O'Reilly and Associates, 1 Jan. 2006.
- [8] M. J. Hernandez, Database Design for Mere Mortals: 25th Anniversary Edition, 4th Edition ed., Addison-Wesley Professional, December 17, 2020.
- [9] V. Khorikov, Unit Testing Principles, Practices, and Patterns, Manning, January 14, 2020.