

Національний технічний університет України  
«Київський політехнічний інститут імені Ігоря Сікорського»  
Факультет інформатики та обчислювальної техніки  
Кафедра інформаційних систем та технологій

## **Лабораторна робота № 2**

з дисципліни «Технології розроблення  
програмного забезпечення»

Тема: «Основи проектування»

«Менеджер завдань»

Виконав:  
студент групи - ІА-32  
Воробйов Кирило  
Андрійович

Перевірів:  
Мягкий Михайло  
Юрійович

Київ 2025

**Тема:** Основи проектування.

**Мета:** Обрати зручну систему побудови UML-діаграм та навчитися будувати діаграми варіантів використання для системи що проектується, розробляти сценарії варіантів використання та будувати діаграми класів предметної області.

**Тема проєкту:** Менеджер завдань (To Do Manager)

**Патерни:** Strategy, Command, Observer, Mediator, Composite, Client-Server

**Опис:** Додаток повинен мати можливість створювати, редагувати та видаляти завдання, забезпечувати гнучку фільтрацію та сортування (Strategy), та обробляти дії користувача (Command). Система має підтримувати ієрархічну структуру (завдання/підзадачі або проєкт/завдання) (Composite), забезпечувати автоматичне оновлення візуальних компонентів при зміні даних (Observer) та керувати взаємодією елементів інтерфейсу (Mediator). Зберігання та синхронізація завдань реалізується через клієнт-серверний зв'язок (Client-Server).

### Зміст

Теоретичні відомості .....	2
Хід роботи .....	15
Діаграма варіантів використання (Use Case Diagram).....	17
Сценарії Використання (Use Case Scenarios) .....	18
Діаграма класів (Class Diagram) .....	21
Діаграма класів реалізації .....	22
Структура бази даних .....	23
Код програми.....	26
Висновки.....	31
Питання до лабораторної роботи .....	31

### Теоретичні відомості

**Вступ до мови UML** — Мова UML є загальноцільовою мовою візуального моделювання, яка розроблена для специфікації, візуалізації, проектування та документування компонентів програмного забезпечення, бізнес-процесів та інших систем [3]. Мова UML є досить строгим та потужним засобом моделювання, який може бути ефективно використаний для

побудови концептуальних, логічних та графічних 18 моделей складних систем різного цільового призначення.

Ця мова увібрала в себе найкращі якості та досвід методів програмної інженерії, які з успіхом використовувалися протягом останніх років при моделюванні великих та

складних систем. З погляду методології ООАП (об'єктно-орієнтованого аналізу та проєктування) досить повна модель складної системи є певною кількістю взаємопов'язаних уявлень (views), кожне з яких відображає аспект поведінки або структури системи. У цьому найзагальнішими уявленнями складної системи прийнято вважати статичне і динамічне, які у своє чергу можуть поділятися інші більш приватні.

Принцип ієрархічної побудови моделей складних систем передбачає розгляд процес побудови моделей на різних рівнях абстрагування або деталізації в рамках фіксованих уявлень.

**Рівень представлення (layer)**— спосіб організації та розгляду моделі на

одному рівні абстракції, що представляє горизонтальний зріз архітектури моделі, тоді як розбиття представляє її вертикальний зріз. При цьому вихідна або початкова модель складної системи має найбільш загальне уявлення та відноситься до концептуального рівня. Така модель, що отримала назву концептуальної, будується на початковому етапі проєктування і може не містити багатьох деталей та аспектів системи, що моделюється. Наступні моделі конкретизують концептуальну модель, доповнюючи її уявленнями логічного та фізичного рівня.

Загалом процес ООАП можна розглядати як послідовний перехід від

розробки найбільш загальних моделей та уявлень концептуального рівня до більш приватних і детальних уявлень логічного та фізичного рівня. У цьому кожному етапі ООАП дані моделі послідовно доповнюються дедалі більше деталей, що дозволяє їм адекватно відбивати різні аспекти конкретної реалізації складної системи.

В рамках мови UML уявлення про модель складної системи фіксуються у вигляді спеціальних графічних конструкцій, що отримали назву діаграм.

**Діаграма (diagram)**— графічне уявлення сукупності елементів моделі у

формі зв'язкового графа, вершинам і ребрам (дугам) якого приписується певна семантика. Нотація канонічних діаграм є основним засобом розробки моделей мовою UML.

У нотації мови UML визначено такі види діаграм:

- варіантів використання (use case diagram);
- класів (class diagram);
- кооперації (collaboration diagram);
- послідовності (sequence diagram);
- станів (statechart diagram);
- діяльності (activity diagram);
- компонентів (component diagram);
- розгортання (deployment diagram).

Перелічені діаграми є невід'ємною частиною графічної нотації мови UML.

Понад те, процес ООАП нерозривно пов'язаний з процесом побудови цих діаграм. При цьому сукупність побудованих таким чином діаграм є

самодостатньою в тому сенсі, що в них міститься вся інформація, яка потрібна для реалізації проєкту складної системи.

Кожна з цих діаграм деталізує та конкретизує різні уявлення про модель складної системи у термінах мови UML. При цьому діаграма варіантів

використання являє собою найбільш загальну концептуальну модель складної системи, яка є вихідною для побудови інших діаграм. Діаграма класів, за своєю суттю, логічна модель, що відбиває статичні аспекти структурної побудови складної системи.

Діаграми кооперації та послідовностей є різновидами логічної моделі, які відображають динамічні аспекти функціонування складної системи. Діаграми станів та діяльності призначені для моделювання поведінки системи. І, нарешті, діаграми компонентів і розгортання служать уявленням

фізичних компонентів складної системи і тому представляють її фізичну модель.

**Діаграма варіантів використання (Use-Cases Diagram)** — Діаграма варіантів використання (Use-Cases Diagram) – це UML діаграма за допомогою якої у графічному вигляді можна зобразити вимоги до системи, що розробляється. Діаграма варіантів використання – це вихідна концептуальна модель проєктованої системи, вона не описує внутрішню побудову системи.

Діаграми варіантів використання призначені для:

- визначення загальної межі функціональності проєктованої системи;
- формулювання загальних вимоги до функціональної поведінки проєктованої системи;
- подальшої розробка вихідної концептуальної моделі системи (діаграми класів);
- створення основи для виконання аналізу, проєктування, розробки та тестування.

Діаграми варіантів використання є відправною точкою при збиранні вимог до програмного продукту та його реалізації. Дана модель будується на аналітичному етапі побудови програмного продукту (збір та аналіз вимог) і дозволяє бізнес-аналітикам отримати більш повне уявлення про необхідне програмне забезпечення та документувати його.

Діаграма варіантів використання складається з низки елементів. Основними елементами є: варіанти використання або прецеденти (use case), актор або дійова особа (actor) та відносини між акторами та варіантами використання (relationship).

**Актори (actor)**— Актором називається будь-який об'єкт, суб'єкт чи система, що взаємодіє з модельованою бізнес-системою ззовні для досягнення своїх цілей або вирішення певних завдань. Це може бути людина, технічний пристрій, програма або будь-яка інша система, яка служить джерелом впливу на систему, що моделюється. Актора можна розглядати як роль, яка виконується людиною в системі.

Ім'я актора має бути достатньо інформативним з точки зору програмного забезпечення, що розробляється, або предметної області. Для цього підходять найменування посад у компанії (наприклад, продавець, касир, менеджер, президент).

**Варіанти використання (use case)** — Варіант використання служить для опису служб, які система надає актору. Інакше кажучи кожен варіант використання визначає набір дій, здійснюваний системою під час діалогу з актором. Кожен варіант використання являє собою послідовність дій, який повинен бути виконаний системою, що проєктується при взаємодії її з відповідним актором, самі ці дії не відображаються на діаграмі.

Варіант використання відображається еліпсом, всередині якого міститься його коротке ім'я з великої літери у формі іменника або дієслова.

При проєктуванні варіантів використання, потрібно приймати до уваги, що з назви варіанта використання має бути зрозуміла тривалість та результат його виконання.

Приклади варіантів використання: реєстрація, авторизація, оформлення замовлення, переглянути замовлення, перевірка стану поточного рахунку тощо.

**Відносини на діаграмі варіантів використання** — Відношення (relationship) – семантичний зв'язок між окремими елементами моделі.

Один актор може взаємодіяти з кількома варіантами використання. У цьому випадку цей актор звертається до кількох служб цієї системи. У свою чергу, один варіант використання може ініціюватися декількома акторами, надаючи для них свій функціонал.

Існують такі відносини: асоціації, узагальнення, залежність (складається з включення та розширення)

**Асоціація (association)** — узагальнене, невідоме ставлення між актором та варіантом використання.

Позначається суцільною лінією між актором та варіантом використання. Розрізняють ненаправлену (двонаправлену) асоціацію та однонаправлену асоціацію. Ненаправлена асоціація показує взаємодію без акцента на напрямок, або коли напрямок ще не аналізувався.

Спрямована, або направлена асоціація (directed association) – також показує що актор асоціюється з варіантом використання але показує, що варіант використання ініціалізується актором. Позначається стрілкою.

Спрямована асоціація дозволяє запровадити поняття основного актора (він є ініціатором асоціації) та другорядного актора (варіант використання є

ініціатором, тобто передає акторові довідкові відомості або звіт про виконану роботу).

**Відношення узагальнення (generalization)** – показує, що нащадок успадковує атрибути у свого прямого батьківського елемента. Тобто, один елемент моделі є спеціальним або окремим випадком іншого елемента моделі.

Може застосовуватися як до акторів, так і до варіантів використання.

Графічно відношення узагальнення позначається суцільною лінією зі стрілкою у формі незафарбованого трикутника, яка вказує на батьківський варіант використання.

Варіант використання «Оплата замовлення» називається предком (чи батьком), а варіант використання «Оплата замовлення банківською картою» – нащадком (чи дочірнім) стосовно «Оплата замовлення».

Особливості використання відношення узагальнення:

- головною особливістю відношення узагальнення є те, що воно може пов'язувати між собою лише елементи одного типу;
- один варіант використання може мати кілька батьківських варіантів використання (множинне успадкування);
- один варіант використання може бути предком кількох дочірніх варіантів використання (таксономічний характер відносин).

У наведеному прикладі Адміністратор успадковує всі атрибути свого предка Користувача, але може мати свої індивідуальні, які не зображені на рисунку.

Відношення залежності (dependency) визначається як форма взаємозв'язку між двома елементами моделі, призначена для специфікації тієї обставини, що зміна одного елемента моделі призводить до зміни деякого іншого елемента.

Загалом залежність є спрямованим бінарним ставленням, яке пов'язує між собою два елементи моделі: незалежний та залежний.

**Відношення включення (include)** – окремий випадок загального відношення залежності між двома варіантами використання, при якому деякий варіант використання містить поведінку, визначену в іншому варіанті використання.

Залежний варіант використання «Автентифікація користувача» називають базовим, а незалежний – включається («Оформлення замовлення»). На рисунку включення означає, що кожне виконання варіанта використання «Оформлення замовлення» завжди включатиме виконання варіанта використання «Автентифікація користувача». На практиці відношення включення використовується для моделювання ситуації, коли існує загальна частина поведінки двох або більше варіантів використання. Загальна частина виноситься на окремий варіант використання.

Особливості використання відношення включення:

- один базовий варіант використання може бути пов'язаний ставленням включення з кількома варіантами використання, що включаються;
- один варіант використання може бути включений до інших варіантів використання;
- в одній діаграмі варіантів використання не може бути замкнутого шляху стосовно включення.

**Відношення розширення (extend)** – показує, що варіант використання розширює базову послідовність дій та вставляє власну послідовність. У цьому на відміну типу відносин «включення» розширена послідовність може здійснюватися залежно від певних умов.

Графічне зображення відношення розширення – пунктирна стрілка спрямована від залежного варіанта (розширює) до незалежного варіанта (базового) з ключовим словом <<extend>>.

Варіант використання «Оформлення замовлення» є базовим і може бути розширений варіантом використання "Надання знижки", наприклад, за наявності у покупця коду на знижку.

Відношення розширення дозволяють моделювати той факт, що базовий варіант використання може приєднувати до своєї поведінки деякі

додаткові поведінки за рахунок розширення у варіанті іншому варіанті використання

Наявність такого відношення передбачає перевірку умови у точці розширення (extension point) у базовому варіанті використання. Точка розширення може мати деяке ім'я і зображена за допомогою примітки.

Відношення розширення з точкою розширення Особливості використання відношення розширення:

- один базовий варіант використання може мати кілька точок розширення, з кожною з яких повинен бути пов'язаний варіант використання, що розширює;
- один розширюючий варіант використання може бути пов'язаний з відношенням розширення з декількома базовими варіантами використання;
- розширюючий варіант використання може, своєю чергою, мати власні розширюючі варіанти використання;
- на одній діаграмі варіантів використання не може бути замкнутого шляху щодо розширення.

## **Сценарії використання**

Діаграма варіантів використання надає знання про необхідну функціональність системи в інтуїтивно-зрозумілому вигляді, проте не несе відомостей про фактичний спосіб її реалізації. Конкретні варіанти використання можуть звучати надто загально та не бути придатними для реалізації програмістами.

Для документації варіантів використання у вигляді певної специфікації та усунення неточностей і непорозуміння діаграм варіантів використання, як частину процесу збору та аналізу вимог складаються звані сценарії використання.

Сценарії використання – це текстові уявлення тих процесів, які відбуваються при взаємодії користувачів системи та самої системи. Вони є

чітко формалізованими, покроковими інструкціями, що описують той чи інший процес у термінах кроків досягнення мети. Сценарії використання однозначно визначають кінцевий результат.

Сценарії використання описують варіанти використання природною мовою. Вони мають загального, шаблонного вигляду написання, проте рекомендується такий перелік для опису:

- Передумови – умови, які повинні бути виконані для виконання даного варіанту використання;
- Постумови – що виходить в результаті виконання даного варіанту використання;
- Взаємодіючі сторони;
- Короткий опис;
- Основний перебіг подій;
- Винятки;
- Примітки.

## **Діаграми класів**

Діаграми класів використовуються при моделюванні програмних систем найчастіше. Вони є однією із форм статичного опису системи з погляду її проєктування, показуючи її структуру. Діаграма класів не відображає динамічної поведінки об'єктів зображених на ній класів. На діаграмах класів показуються класи, інтерфейси та відносини між ними.

## **Представлення класів**

Клас – це основний будівельний блок програмної системи. Це поняття є і в мовах програмування, тобто між класами UML та програмними класами є відповідність, що є основою для автоматичної генерації програмних кодів або для виконання реінжинірингу. Кожен клас має назву, атрибути та операції. Клас на діаграмі показується як прямокутник, розділений на 3

області. У верхній міститься назва класу, у середній – опис атрибутів (властивостей), у нижній – назви операцій – послуг, що надаються об'єктами цього класу.

Атрибути класу визначають склад та структуру даних, що зберігаються в об'єктах цього класу. Кожен атрибут має ім'я та тип, який визначає, які дані він представляє.

При реалізації об'єкта в програмному коді для атрибутів буде виділено пам'ять, необхідна зберігання всіх атрибутів, і кожен атрибут матиме конкретне значення у час роботи програми. Об'єктів одного класу у програмі може бути скільки завгодно багато, всі вони мають однаковий набір атрибутів, описаний у класі, але значення атрибутів у кожного об'єкта свої і можуть змінюватися в ході виконання програми.

Для кожного атрибуту класу можна встановити видимість (visibility). Ця характеристика показує, чи доступний атрибут для інших класів. У UML визначено такі рівні видимості атрибутів:

- + Відкритий (public) – атрибут видно для будь-якого іншого класу (об'єкта);
- ~ В межах пакету (package) – атрибут видно для будь-якого іншого класу, який знаходиться в цьому ж пакеті;
- # Захищений (protected) – атрибут видно для нащадків цього класу;
- Закритий (private) – атрибут не видно зовнішніми класами (об'єктами) і може використовуватися лише об'єктом, що його містить.

Клас містить оголошення операцій, що є визначення запитів, які повинні виконувати об'єкти даного класу. Кожна операція має сигнатуру, що містить ім'я операції, тип значення, що повертається, і список параметрів, який може бути порожнім. Реалізація операції як процедури – це спосіб, що належить класу. Для операцій, як й у атрибутів класу, визначено поняття «видимість». Закриті операції є внутрішніми для об'єктів класу і недоступні з інших об'єктів. Інші утворюють інтерфейсну частину класу і є засобом інтеграції класу до програмної системи.

## **Відносини між класами**

На діаграмах класів зазвичай показуються асоціації та узагальнення.

Кожна асоціація несе інформацію про зв'язки між об'єктами усередині програмної системи. Найчастіше використовуються бінарні асоціації, які пов'язують два класи. Асоціація може мати назву, яка має виражати суть відображуваного зв'язку. Крім назви, асоціація може мати таку характеристику як множинність. Вона показує, скільки об'єктів кожного класу може брати участь у асоціації.

Множинність вказується у кожного кінця асоціації (полюса) і задається конкретним числом чи діапазоном чисел. Множинність, зазначена у вигляді зірочки, передбачає будь-яку кількість (у тому числі й нуль). Пов'язані між собою можуть і об'єкти одного класу, тому асоціація може пов'язувати клас із собою. Наприклад, для класу «Мешканець міста» можна ввести асоціацію «Сусідство», яке дозволить знаходити всіх сусідів конкретного мешканця.

Асоціація «включає» (рисунок 2.11) показує, що комплект може містити кілька різних товарів. В даному випадку спрямована асоціація дозволяє знайти всі види товарів, що входять у комплект, але не дає відповіді на запитання, чи товар цього виду входить у який-небудь набір.

Асоціація сама може мати властивості класу, тобто мати атрибути та операції. У цьому випадку вона називається клас-асоціацією і може розглядатися як клас, у якого крім явно зазначених атрибутів та операцій є посилення на обидва зв'язувані нею класи.

Асоціація – найбільш загальний вид зв'язку між двома класами системи. Як правило, вона відображає використання одного класу іншим за допомогою певної якості або поля.

Узагальнення (успадкування) на діаграмах класів використовується, щоб показати зв'язок між класом-батьком та класом-нащадком. Воно вводиться на діаграму, коли виникає різновид будь-якого класу, і навіть у випадках, як у системі виявляються кілька класів, які мають подібну поведінку (у разі загальні елементи поведінки виносяться на більш високий рівень, утворюючи узагальнюючий клас).

Агрегацією позначається відношення частина-ціле, коли об'єкти одного класу входять до об'єкта іншого класу. Типовим прикладом таких відносин є списки об'єктів. У цьому випадку список буде виступати агрегатом, а об'єкти, що входять до списку, елементами, що агрегуються.

Приклад агрегації показано на рисунку 2.13. Ромб в нотації частина ціле ставиться біля класу, що представляє ціле у цьому відношенні. Навчальна група містить список студентів. Композиція – це є відображенням зв'язку, при якому об'єкти «Навчальна група» та «Студент» можуть існувати один без одного. Наприклад, при видаленні об'єкту «Навчальна група» об'єкти «Студент» пов'язані з нею продовжують існувати.

Композицією також позначається відношення частина-ціле, але позначає тісніший зв'язок між елементами, що представляють ціле та частини.

Прикладом композиції можна назвати зв'язок між будівлею і приміщеннями в будівлі. Будівля та приміщення складають неділиме ціле. Приклад композиції показано на рисунку 2.14. Композиція показує, що квартири не можуть існувати за межами житлового будинку і перенести квартири до іншого житлового будинку неможливо.

Якщо говорити про відображення агрегації та композиції в програмному коді, то агрегація, як правило відображується як посилання на об'єкт, а композиція – це змінна типу структури. Біля цілого може бути вказана назва, яка вказує на назву поля класу, яке відображений на діаграмі цим відношенням.

## **Застосування діаграм класів**

Діаграми класів створюються при логічному моделюванні програмних систем та служать для наступних цілей:

- Для моделювання даних. Аналіз предметної області дозволяє виявити основні характерні нею сутності та зв'язку з-поміж них. Це зручно моделюється з допомогою діаграм класів. Ці діаграми є основою побудови концептуальної моделі.

- Для представлення архітектури програмної системи. Можна виділити архітектурно значимі класи і відобразити їх на діаграмах, що описують архітектуру програмної системи.

- Для моделювання навігації екранів. На таких діаграмах показуються прикордонні класи та їхній логічний взаємозв'язок. Інформаційні поля моделюються як атрибути класів, а кнопки, що управляють, – як операції та відносини.

- Для моделювання логіки програмних компонентів.
- Для моделювання логіки обробки даних.

## **Логічна структура бази даних**

Розрізняють дві моделі бази даних – логічну та фізичну. Фізична модель бази даних представляє собою набір бінарних даних у вигляді файлів, структурованих та згрупованих згідно з призначенням (сегменти, екстенти та ін.), що використовується для швидкого та ефективного отримання інформації з жорсткого диска, а також для компактного зберігання та розміщення даних на жорсткому диску.

Логічна модель бази даних є структурою таблиць, уявлень, індексів та інших логічних елементів бази даних, що дозволяють власне програмування та використання бази даних.

Процес створення логічної моделі бази даних зветься проєктування бази даних (database design). Проєктування відбувається у зв'язку з опрацюванням архітектури програмної системи, оскільки база даних створюється зберігання даних, одержуваних з програмних класів.

Відповідно можна розрізнити кілька підходів до зв'язування програмних класів та таблиць: одна таблиця – один клас, одна таблиця – кілька класів, один клас – кілька таблиць. Залежно від обраного підходу, буде ускладнюватись (і уповільнюватись) робота з базою даних при додаванні даних, або отримання даних з БД та парсинг їх у відповідні класи.

З іншого боку, якщо програмні класи є сутністю проєктованої системи (елементи предметної області), то таблиці відображають їх технічну реалізацію та спосіб зберігання та зв'язку.

Основним керівництвом під час проєктування таблиць є т. зв. нормальні форми баз даних.

## **Нормальні форми**

Нормальна форма – властивість відношення в реляційній моделі даних, що характеризує його з погляду надмірності, що потенційно призводить до логічно помилкових результатів вибірки або зміни даних. Нормальна форма окреслюється сукупністю вимог, яким має задовольняти ставлення.

Нормалізація призначена для приведення структури БД до виду, що забезпечує мінімальну логічну надмірність, і не має на меті зменшення або збільшення продуктивності роботи або зменшення або збільшення фізичного обсягу бази даних. Кінцевою метою нормалізації є зменшення потенційної суперечливості інформації, що зберігається в базі даних [4].

Змінна відношення знаходиться в першій нормальній формі (1НФ) тоді і тільки тоді, коли в будь-якому допустимому значенні відношення кожен його кортеж містить лише одне значення для кожного з атрибутів.

Змінна відношення знаходиться в другій нормальній формі тоді і тільки тоді, коли вона знаходиться в першій нормальній формі, і кожен неключовий атрибут функціонально повно залежить від її потенційного ключа.

Змінна відношення знаходиться у третій нормальній формі тоді і лише тоді, коли вона знаходиться у другій нормальній формі, і відсутні транзитивні функціональні залежності неключових атрибутів від ключових.

Змінна відношення знаходиться в нормальній формі Бойса-Кодда (інакше – в посиленій третій нормальній формі) тоді і тільки тоді, коли кожна її нетривіальна і неприведена зліва функціональна залежність має в якості свого детермінанта певний потенційний ключ.

## **Проектування БД**

Для проектування бази даних можна використовувати Schema View у відповідних засобах роботи з СУБД (Microsoft Sql Server Management Studio, PL/SQL Developer та інші) або вбудований засіб Microsoft Office Access.

Для створення таблиць необхідно натиснути кнопку «Створити» (CREATE).

Після заповнення полів та їх типів необхідно призначити первинний ключ. Після цього можна виставити зв'язок між таблицями у вікні

## **Хід роботи**

### **Завдання:**

- Ознайомитись з короткими теоретичними відомостями.

- Проаналізувати тему та спроектувати діаграму варіантів використання відповідно до обраної теми лабораторного циклу.
- Спроектувати діаграму класів предметної області.
- Вибрати 3 варіанти використання та написати за ними сценарії використання.
- На основі спроектованої діаграми класів предметної області розробити основні класи та структуру бази даних системи. Класи даних повинні реалізувати шаблон Repository для взаємодії з базою даних.
- Нарисувати діаграму класів для реалізованої частини системи.
- Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму варіантів використання відповідно, діаграму класів системи, вихідні коди класів системи, а також зображення структури бази даних.

## Діаграма варіантів використання (Use Case Diagram)



Рис. 1. Діаграма варіантів використання

## Сценарії Використання (Use Case Scenarios)

### 1. Сценарій: Створення завдання

**Передумови:** Користувач автентифікований в системі та знаходиться на головному екрані (або в конкретному проєкті).

**Постумови:** Нове завдання створено та збережено в системі. Воно з'являється у загальному списку завдань.

**Взаємодіючі сторони:** Користувач, Система ("To Do Manager").

**Короткий опис:** Цей варіант використання описує процес додавання Користувачем нового завдання до списку.

#### **Основний потік подій:**

Користувач ініціює створення завдання (наприклад, натискає кнопку "Додати завдання").

Система відображає форму/поля для введення даних про нове завдання (як мінімум — назву).

Користувач вводить назву завдання.

(Опціонально) Користувач вводить додаткові дані: опис, пріоритет, дедлайн, обирає проєкт.

Користувач підтверджує створення (наприклад, натискає кнопку "Зберегти" або Enter).

Система валідує введені дані (наприклад, перевіряє, що назва не порожня).

Система зберігає нове завдання в базі даних.

Система оновлює список завдань у Користувача, відображаючи нове завдання.

Варіант використання завершується.

#### **Винятки:**

Виняток 1 (Крок 6): Назва завдання порожня.

Система не зберігає завдання.

Система виводить повідомлення про помилку ("Назва завдання не може бути порожньою").

Користувач повертається до кроку 3.

## **2. Сценарій: Редагування завдання**

**Передумови:** Користувач бачить список завдань. Завдання, яке потрібно редагувати, існує в системі.

**Постумови:** Зміни до завдання збережені. Оновлені дані відображаються у списку завдань.

**Взаємодіючі сторони:** Користувач, Система.

**Короткий опис:** Дозволяє Користувачу змінити раніше введені дані завдання (назву, опис, дедлайн тощо).

### **Основний потік подій:**

Користувач обирає завдання для редагування (наприклад, клікає на нього або на іконку "Редагувати").

Система відкриває форму редагування, заповнену поточними даними цього завдання.

Користувач змінює одне або декілька полів (назву, опис, пріоритет, дедлайн).

Користувач підтверджує зміни (натискає "Зберегти").

Система валідує дані (наприклад, перевіряє, що назва не стала порожньою).

Система оновлює дані завдання в базі даних.

Система оновлює вигляд завдання у списку.

Варіант використання завершується.

### **Винятки:**

Виняток 1 (Крок 5): Назва завдання стала порожньою.

Система не зберігає зміни.

Система виводить повідомлення про помилку.

Користувач повертається до кроку 3.

### 3. Сценарій: Відзначити завдання як виконане

**Передумови:** Користувач бачить список активних завдань.

**Постумови:** Завдання отримує статус "Виконане". Вигляд завдання у списку змінюється (наприклад, воно стає перекресленим або переміщується до списку "Виконані").

**Взаємодіючі сторони:** Користувач, Система.

**Короткий опис:** Дозволяє Користувачу змінити статус завдання на "Виконане".

**Основний потік подій:**

Користувач знаходить потрібне завдання у списку.

Користувач взаємодіє з елементом керування для виконання (наприклад, натискає на чекбокс біля назви завдання).

Система миттєво змінює статус завдання на "Виконане".

Система зберігає новий статус в базі даних.

Система оновлює UI, щоб візуально відобразити новий статус (наприклад, перекреслює назву).

Варіант використання завершується.

**Винятки:** (Для цього простого сценарію значних винятків, ініційованих користувачем, зазвичай немає. Можливі системні збої, але ми їх тут не описуємо).

## Діаграма класів (Class Diagram)

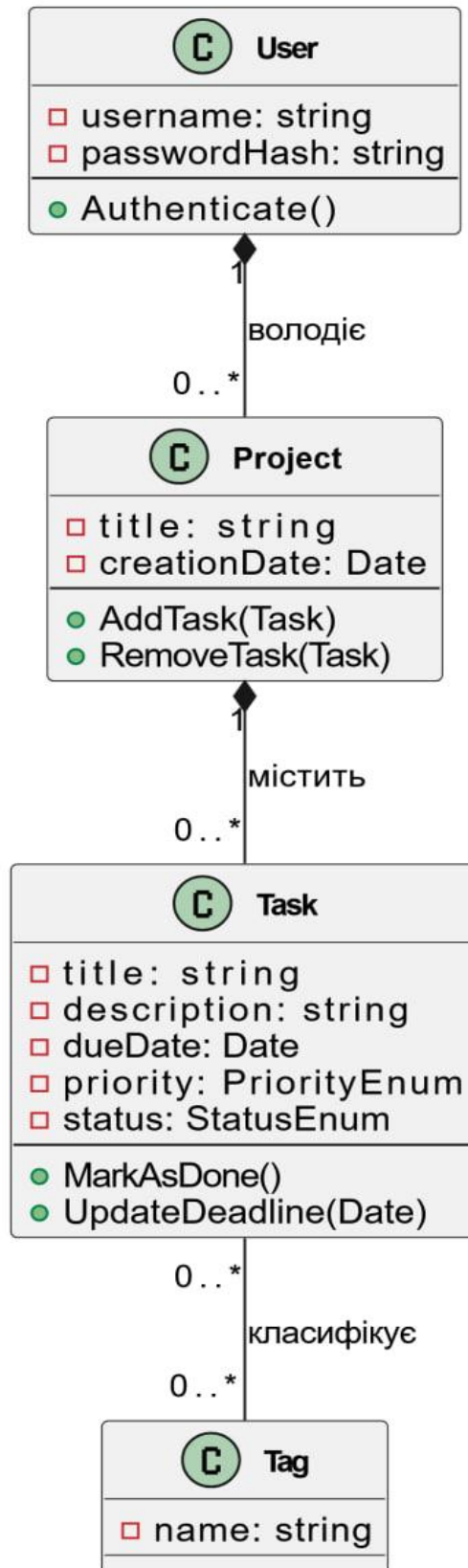


Рис. 2. Діаграма класів

## Зв'язки між класами

### 1. User та Project:

- Один **Користувач** може мати багато **Проектів**.
- Один **Проект** належить тільки одному **Користувачу**.
- **Відношення: Композиція** (Зафарбований ромб) — без Користувача його Проекти не існують. Множинність: **1..\*** (Користувач) до **1** (Проект).

### 2. Project та Task:

- Один **Проект** може містити багато **Завдань**.
- Одне **Завдання** належить тільки одному **Проекту**.
- **Відношення: Композиція** (Зафарбований ромб). Множинність: **1..\*** (Проект) до **1** (Завдання).

### 3. Task та Tag:

- Одне **Завдання** може мати багато **Тегів**.
- Один **Тег** може застосовуватись до багатьох **Завдань**.
- **Відношення: Асоціація** (Звичайний зв'язок). Множинність: \*\*\* \*\* до \*

## Діаграма класів реалізації

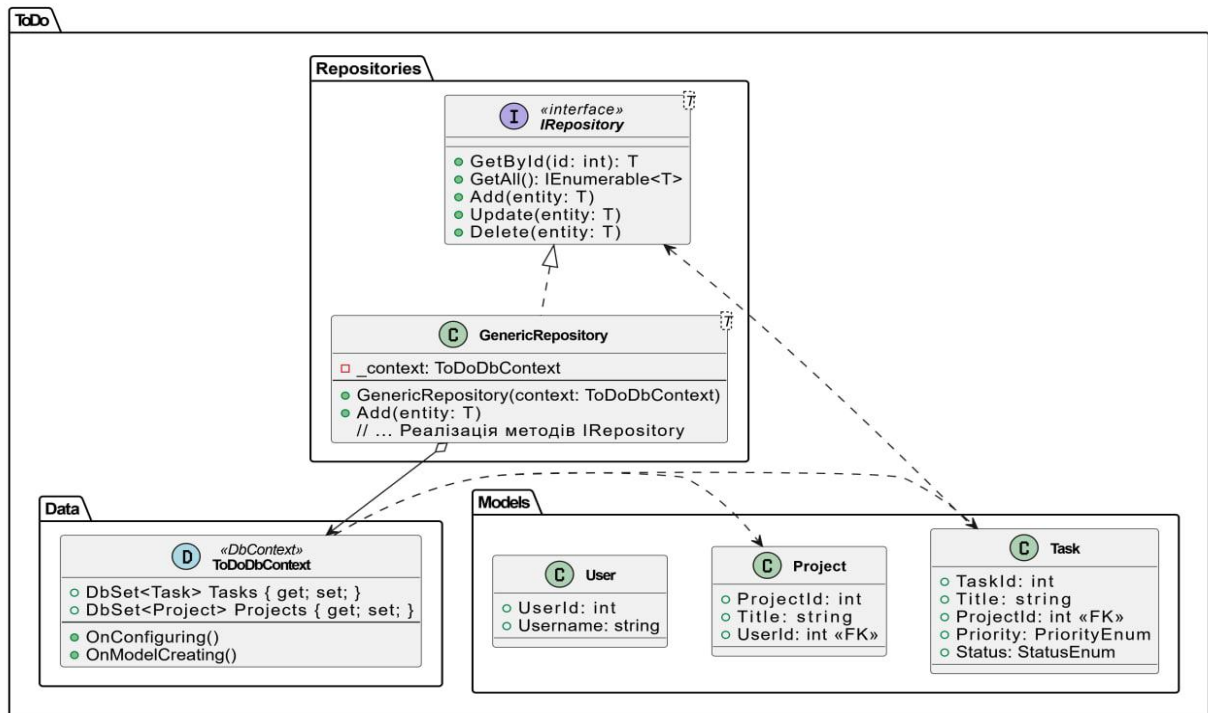
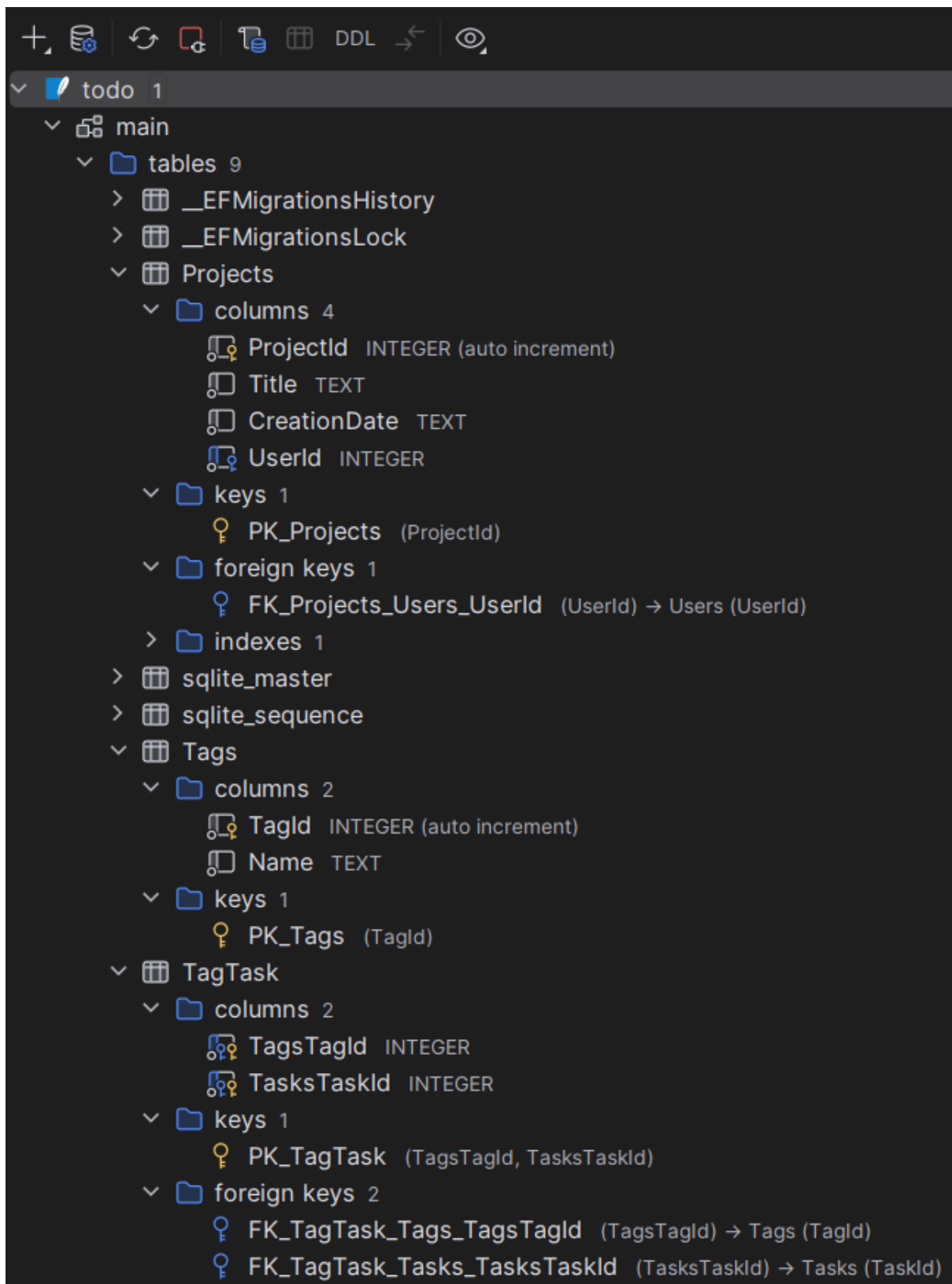























Рис. 3. Діаграма класів реалізації

## Структура бази даних



- ▼  Tasks
  - ▼  columns 7
    -  TaskId INTEGER (auto increment)
    -  Title TEXT
    -  Description TEXT
    -  DueDate TEXT
    -  Priority INTEGER
    -  Status INTEGER
    -  ProjectId INTEGER
  - ▼  keys 1
    -  PK\_Tasks (TaskId)
  - ▼  foreign keys 1
    -  FK\_Tasks\_Projects\_ProjectId (ProjectId) → Projects (ProjectId)
  - >  indexes 1
- ▼  Users
  - ▼  columns 3
    -  UserId INTEGER (auto increment)
    -  Username TEXT
    -  PasswordHash TEXT
  - ▼  keys 1
    -  PK\_Users (UserId)

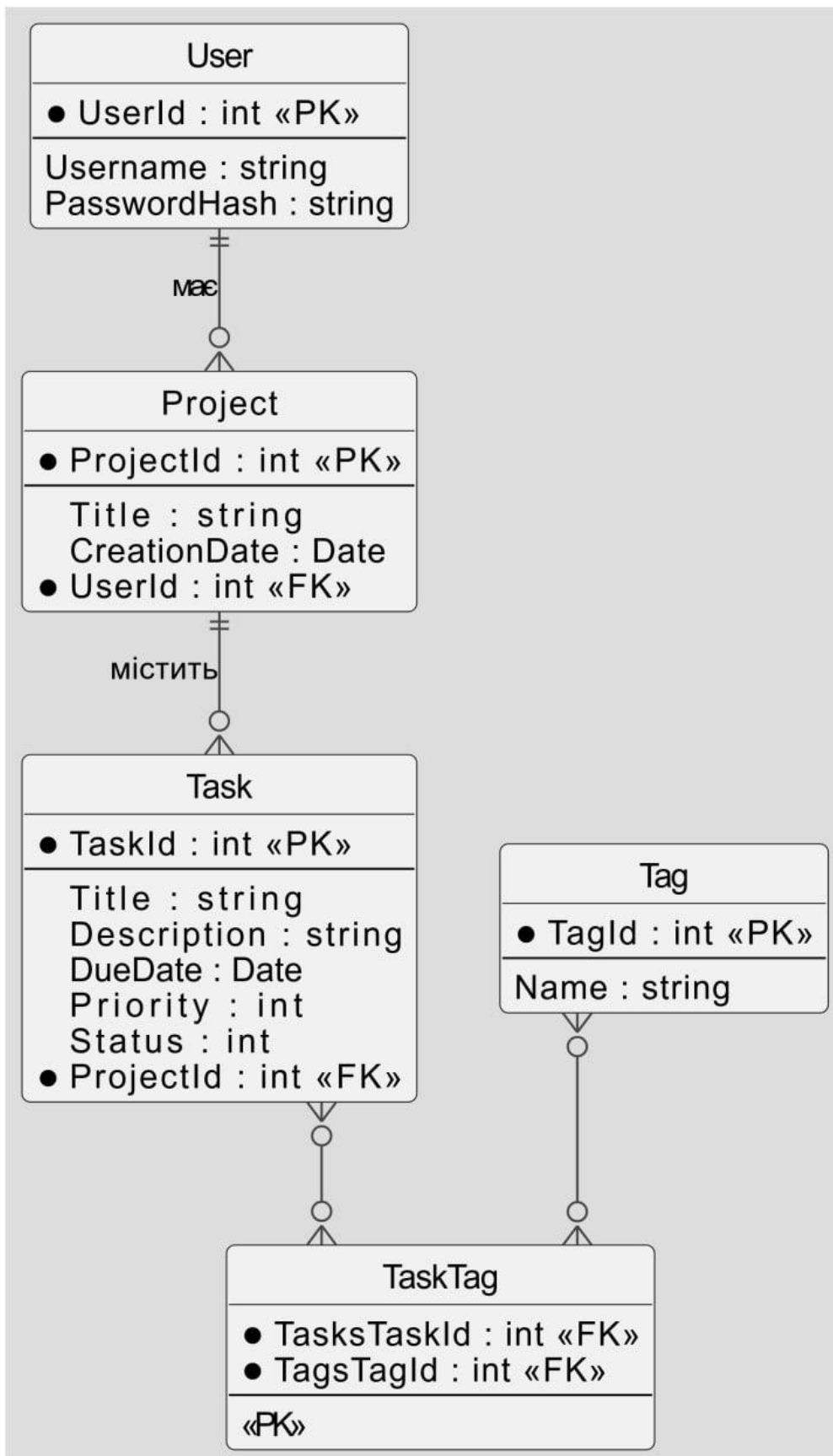


Рис. 4. Структура бази даних

## Код программы

```
Solution
  ▼ ToDoManagerSolution · 1 project
    ▼ ToDo
      > Dependencies
      ▼ Data
        c# ToDoDbContext.cs
      ▼ Enums
        c# PriorityEnum.cs
        c# StatusEnum.cs
      ▼ Migrations
        > c# 20251114231245_InitialCreat
        c# ToDoDbContextModelSnaps
      ▼ Models
        c# Project.cs
        c# Tag.cs
        c# Task.cs
        c# User.cs
      ▼ Repositories
        c# GenericRepository.cs
        c# IRepository.cs
        todo.db
    > Scratches and Consoles

c# ToDoDbContext.cs
1 {} using Microsoft.EntityFrameworkCore;
2   using ToDo.Models;
3   using Task = ToDo.Models.Task;
4
5   namespace ToDo.Data;
6
7   [4 usages]
8   public class ToDoDbContext : DbContext
9   {
10     public DbSet<User> Users { get; set; }
11     public DbSet<Project> Projects { get; set; }
12     public DbSet<Task> Tasks { get; set; }
13     public DbSet<Tag> Tags { get; set; }
14
15     protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
16     {
17         optionsBuilder.UseSqlite(connectionString: "Data Source=todo.db");
18     }
19
20     protected override void OnModelCreating(ModelBuilder modelBuilder)
21     {
22         modelBuilder.Entity<Project>() // EntityTypeBuilder<Project>
23             .HasOne(p:Project => p.User) // ReferenceNavigationBuilder<Project,User>
24             .WithMany(u:User => u.Projects)
25             .HasForeignKey(p:Project => p.UserId);
26
27         modelBuilder.Entity<Task>() // EntityTypeBuilder<Task>
28             .HasOne(t:Task => t.Project) // ReferenceNavigationBuilder<Task,Project>
29             .WithMany(p:Project => p.Tasks)
30             .HasForeignKey(t:Task => t.ProjectId);
31
32         modelBuilder.Entity<Task>() // EntityTypeBuilder<Task>
33             .HasMany(t:Task => t.Tags) // CollectionNavigationBuilder<Task,Tag>
34             .WithMany(t:Tag => t.Tasks);
35     }
}
```

```
c# PriorityEnum.cs
1 {} namespace ToDo.Enums;
2
3   [1 usage] [4 exposing APIs]
4   public enum PriorityEnum
5   {
6       Low,
7       Medium,
8       High
9   }
```

```
C# StatusEnum.cs ×  
1 {} namespace ToDo.Enums;  
2  
3 public enum StatusEnum  
4 {  
5     New,  
6     InProgress,  
7     Done  
8 }
```

```
C# Project.cs ×  
1 {} namespace ToDo.Models;  
2  
3 public class Project  
4 {  
5     public int ProjectId { get; set; }  
6     public string Title { get; set; }  
7     public DateTime CreationDate { get; set; }  
8  
9     public int UserId { get; set; }  
10    public virtual User User { get; set; }  
11  
12    public virtual ICollection<Task> Tasks { get; set; }  
13 }
```

C# Tag.cs ×

```
1 {} namespace ToDo.Models;
2
3 public class Tag
4 {
5     public int TagId { get; set; }
6     public string Name { get; set; }
7
8     public virtual ICollection<Task> Tasks { get; set; }
9 }
```

C# Task.cs ×

```
1 {} using ToDo.Enums;
2
3 namespace ToDo.Models;
4
5 public class Task
6 {
7     public int TaskId { get; set; }
8     public string Title { get; set; }
9     public string Description { get; set; }
10    public DateTime DueDate { get; set; }
11    public PriorityEnum Priority { get; set; }
12    public StatusEnum Status { get; set; }
13
14    public int ProjectId { get; set; }
15    public virtual Project Project { get; set; }
16
17    public virtual ICollection<Tag> Tags { get; set; }
18 }
```

C# User.cs ×

```
1 {} namespace ToDo.Models;
2
3 public class User
4 {
5     public int UserId { get; set; }
6     public string Username { get; set; }
7     public string PasswordHash { get; set; }
8
9     public virtual ICollection<Project> Projects { get; set; }
10 }
```

C# GenericRepository.cs ×

```
1 {} using Microsoft.EntityFrameworkCore;
2 using ToDo.Data;
3
4 namespace ToDo.Repositories;
5
6 ^, public class GenericRepository<T> : IRepository<T> where T : class
7 {
8     private readonly ToDoDbContext _context;
9     private readonly DbSet<T> _dbSet;
10
11     public GenericRepository(ToDoDbContext context)
12     {
13         _context = context;
14         _dbSet = context.Set<T>();
15     }
16
17 ^, public void Add(T entity)
18 {
19     _dbSet.Add(entity);
20     _context.SaveChanges();
21 }
22
23 ^, public void Delete(T entity)
24 {
25     _dbSet.Remove(entity);
26     _context.SaveChanges();
27 }
28 }
```

```

29 ^,      public IEnumerable<T> GetAll()
30          {
31              return _dbSet.ToList();
32          }
33
34 ^,      public T GetById(int id)
35          {
36              return _dbSet.Find(id);
37          }
38
39 ^,      public void Update(T entity)
40          {
41              _dbSet.Update(entity);
42              _context.SaveChanges();
43          }
44      }

```

c# IRepository.cs ×

```

1 {}      namespace ToDo.Repositories;
2
3 ^,      [1 usage] [1 inheritor]
4          public interface IRepository<T> where T : class
5          {
6              [1 implementation]
7              T GetById(int id);
8              [1 implementation]
9              IEnumerable<T> GetAll();
10             [1 implementation]
11             void Add(T entity);
12             [1 implementation]
13             void Update(T entity);
14             [1 implementation]
15             void Delete(T entity);
16         }

```

## **Висновки**

Висновки: під час виконання лабораторної роботи ми навчилися проєктувати діаграму варіантів використання, що визначає функціональні межі системи, а також розробляти детальні сценарії використання. На основі цих вимог ми спроектували діаграму класів предметної області та реалізували її ключові класи, включаючи структуру бази даних та архітектурний шаблон Repository, що забезпечує коректну взаємодію із даними.

## **Питання до лабораторної роботи**

### **1. Що таке UML?**

Мова UML (Unified Modeling Language) — це загальноцільова мова візуального моделювання, розроблена для специфікації, візуалізації, проєктування та документування компонентів програмного забезпечення, бізнес-процесів та інших систем. UML є досить строгим та потужним засобом моделювання.

### **2. Що таке діаграма класів UML?**

Діаграма класів, за своєю суттю, є логічною моделлю, що відображає статичні аспекти структурної побудови складної системи. Вона показує класи, інтерфейси та відносини між ними.

### **3. Які діаграми UML називають канонічними?**

У нотації мови UML визначено такі види канонічних діаграм:

- варіантів використання (use case diagram);
- класів (class diagram);
- кооперації (collaboration diagram);
- послідовності (sequence diagram);
- станів (statechart diagram);
- діяльності (activity diagram);

- компонентів (component diagram);
- розгортання (deployment diagram).

#### 4. Що таке діаграма варіантів використання?

Діаграма варіантів використання (Use-Cases Diagram) — це UML-діаграма, за допомогою якої у графічному вигляді можна зобразити вимоги до системи, що розробляється. Вона є вихідною концептуальною моделлю проєктованої системи і не описує її внутрішню побудову.

#### 5. Що таке варіант використання?

Варіант використання (use case) служить для опису служб, які система надає актору. Кожен варіант використання визначає набір дій, здійснюваний системою під час діалогу з актором.

#### 6. Які відношення можуть бути відображені на діаграмі використання?

На діаграмі варіантів використання можуть бути відображені такі відношення:

- Асоціації (association);
- Узагальнення (generalization);
- Залежність (dependency), яка складається з включення (<<include>>) та розширення (<<extend>>).

#### 7. Що таке сценарій?

Сценарії використання — це текстові уявлення тих процесів, які відбуваються при взаємодії користувачів системи та самої системи. Вони є чітко формалізованими, покроковими інструкціями, що описують процес у термінах кроків досягнення мети.

#### 8. Що таке діаграма класів?

Діаграма класів є однією з форм статичного опису системи з погляду її проектування, показуючи її структуру. На ній показуються класи, інтерфейси та відносини між ними.

9. Які зв'язки між класами ви знаєте?

На діаграмах класів зазвичай показуються:

- Асоціації;
- Узагальнення (успадкування);
- Агрегація (відношення частина-ціле, коли об'єкти одного класу входять до об'єкта іншого);
- Композиція (відношення частина-ціле, що позначає тісніший зв'язок між елементами).

10. Чим відрізняється композиція від агрегації?

Агрегацією позначається відношення частина-ціле, коли об'єкти одного класу входять до об'єкта іншого, але можуть існувати один без одного. Наприклад, при видаленні об'єкта "Навчальна група" об'єкти "Студент" продовжують існувати. Композицією позначається тісніший зв'язок частина-ціле, де частини не можуть існувати за межами цілого.

11. Чим відрізняється зв'язки типу агрегації від зв'язків композиції на діаграмах класів?

На діаграмах класів композиція позначається зафарбованим ромбом біля класу, що представляє ціле, тоді як агрегація позначається порожнім (незафарбованим) ромбом біля цілого.

12. Що являють собою нормальні форми баз даних?

Нормальна форма — це властивість відношення в реляційній моделі даних, що характеризує його з погляду надмірності, яка потенційно призводить до логічно помилкових результатів вибірки або зміни даних. Нормалізація призначена для приведення структури БД до вигляду, що забезпечує мінімальну логічну надмірність.

### 13.Що таке фізична модель бази даних? Логічна?

Фізична модель бази даних представляє собою набір бінарних даних у вигляді файлів, структурованих та згрупованих для швидкого та ефективного отримання інформації з жорсткого диска. Логічна модель бази даних — це структура таблиць, уявлень, індексів та інших логічних елементів, що дозволяють власне програмування та використання бази даних.

### 14.Який взаємозв'язок між таблицями БД та програмними класами?

Програмні класи є сутностями проєктованої системи (елементами предметної області), тоді як таблиці відображають їхню технічну реалізацію та спосіб зберігання і зв'язку. Розрізняють кілька підходів до зв'язування: одна таблиця — один клас, одна таблиця — кілька класів, один клас — кілька таблиць.