Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №**2**
з предмету «Технології розроблення програмного забезпечення»
на тему «**Діаграма варіантів використання. Сценарії варіантів використання. Діаграми UML. Діаграми класів. Концептуальна модель системи**»

Виконав
студент групи ІА-23:
Воронюк Є. В.

Перевірив:
Мягкий М. Ю.

Київ 2024

**Зміст**

**Мета:** розробка та аналіз концептуальної моделі для обраного варіанту із використанням UML-діаграм; зокрема створення діаграми варіантів використання. Розробка детальних сценаріїв використання. Проектування діаграми класів. Створення концептуальної моделі бази даних.

Завдання 1: ознайомитися з короткими теоретичними відомостями.
Завдання 2: проаналізуйте тему та намалюйте схему прецеденту, що відповідає обраній темі лабораторної роботи.
Завдання 3: намалюйте діаграму класів для реалізованої частини системи.
Завдання 4: оберіть 3 прецеденти і напишіть на їх основі прецеденти.
Завдання 5: розробити основні класи і структуру системи баз даних.
Завдання 6: класи даних повинні реалізувати шаблон Репозиторію для взаємодії з базою даних.
Завдання 7: підготувати звіт про хід виконання лабораторних робіт. Звіт, що подається повинен містити: діаграму прецедентів, діаграму класів системи, вихідні коди класів системи, а також зображення структури бази даних.

**Тема для виконання:**

**22 - FTP-server (state, builder, memento, template method, visitor, client- server)**

FTP-сервер повинен вміти коректно обробляти і відправляти відповіді по протоколу FTP, з можливістю створення користувачів (з паролями) і доступних їм папок, розподілу прав за стандартною схемою (rwe), ведення статистики з'єднань, обмеження максимальної кількості підключень і максимальної швидкості поширення глобально і окремо для кожного облікового запису.

**Короткі теоретичні відомості:**

Мова UML являє собою загальноцільову мову візуального моделювання, яка розроблена для специфікації, візуалізації, проектування та документування компонентів програмного забезпечення, бізнес-процесів та інших систем. Мова UML є досить строгим і потужним засобом моделювання, який може бути ефективно використаний для побудови концептуальних, логічних і графічних моделей складних систем різного цільового призначення. Ця мова увібрала в себе найкращі якості та досвід методів програмної інженерії, які успішно використовувалися протягом останніх років при моделюванні великих і складних систем.

З точки зору методології ООАП (об'єктно-орієнтованого аналізу і проектування) досить повна модель складної системи представляє собою певну кількість взаємопов'язаних представлень (views), кожне з яких відображає аспект поведінки або структури системи. При цьому найбільш загальними представленнями складної системи прийнято вважати статичне і динамічне, які в свою чергу можуть підрозділятися на інші більш часткові.

Принцип ієрархічної побудови моделей складних систем приписує розглядати процес побудови моделей на різних рівнях абстрагування або деталізації в рамках фіксованих представлень.

**Рівень представлення**

Рівень представлення (layer) — спосіб організації і розгляду моделі на одному рівні абстракції, який представляє горизонтальний зріз архітектури моделі, в той час як розбиття представляє її вертикальний зріз.

При цьому вихідна або первинна модель складної системи має найбільш загальне представлення і відноситься до концептуального рівня. Така модель, що отримала назву концептуальної, будується на початковому етапі проектування і може не містити багатьох деталей і аспектів модельованої системи. Наступні моделі конкретизують концептуальну модель, доповнюючи її представленнями логічного і фізичного рівня.

В цілому ж процес ООАП можна розглядати як послідовний перехід від розробки найбільш загальних моделей і представлень концептуального рівня до більш часткових і детальних представлень логічного і фізичного рівня. При цьому на кожному етапі ООАП дані моделі послідовно доповнюються все більшою кількістю деталей, що дозволяє їм більш адекватно відображати різні аспекти конкретної реалізації складної системи.

**Діаграма**

У рамках мови UML всі представлення про модель складної системи фіксуються у вигляді спеціальних графічних конструкцій, що отримали назву діаграм.

Діаграма (diagram) — графічне представлення сукупності елементів моделі у формі зв'язного графа, вершинам і ребрам (дугам) якого приписується певна семантика. Нотація канонічних діаграм — основний засіб розробки моделей на мові UML.

В нотації мови UML визначені наступні 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)

**Діаграма варіантів використання**

Перелік цих діаграм та їх назви є канонічними в тому сенсі, що представляють собою невід'ємну частину графічної нотації мови UML. Більше того, 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 необхідна для реалізації проекту складної системи.

Кожна з цих діаграм деталізує і конкретизує різні представлення про модель складної системи в термінах мови UML. При цьому діаграма варіантів використання представляє собою найбільш загальну концептуальну модель складної системи, яка є вихідною для побудови всіх інших діаграм. Діаграма класів, по своїй суті, логічна модель, що відображає статичні аспекти структурної побудови складної системи.

Діаграма варіантів використання (Use-Cases Diagram)

Діаграма варіантів використання (Use-Cases Diagram) - це UML діаграма, за допомогою якої в графічному вигляді можна зобразити вимоги до розроблюваної системи. Діаграма варіантів використання – це вихідна концептуальна модель проектованої системи, вона не описує внутрішній устрій системи.

Діаграми варіантів використання призначені для:

Визначення загальної межі функціональності проектованої системи

Сформулювати загальні вимоги до функціональної поведінки проектованої системи

Розробка вихідної концептуальної моделі системи

Створення основи для виконання аналізу, проектування, розробки і тестування Діаграми варіантів використання є відправною точкою при зборі вимог до програмного продукту та його реалізації. Дана модель будується на аналітичному етапі побудови програмного продукту (збір і аналіз вимог) і дозволяє бізнес-аналітикам отримати більш повне уявлення про необхідне програмне забезпечення і документувати його.

**Актори (actor)**

Актором називається будь-який об'єкт, суб'єкт або 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 модельовану систему.

**Варіанти використання (use case)**

Варіант використання служить для опису послуг, які система надає актору. Іншими словами, кожен варіант використання визначає набір дій, що виконуються системою при діалозі з актором. Кожен варіант використання представляє собою послідовність дій, яка повинна бути виконана проектованою системою 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.

Приклади варіантів використання: реєстрація, авторизація, оформлення замовлення, перегляд замовлення, перевірка стану поточного рахунку і т.д.

Відношення на діаграмі варіантів використання

Відношення (relationship) — семантичний зв'язок між окремими елементами моделі.

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

Існують наступні відношення:

- асоціації
- узагальнення
- залежність (складається з включення і розширення)

**Асоціація**

Асоціація (association) – узагальнене, невідоме відношення між актором і варіантом використання. Позначається суцільною лінією між актором і варіантом використання.

Направлена асоціація (directed association) – те ж, що і проста асоціація, але показує, що варіант використання ініціалізується актором. Позначається стрілкою.

Направлена асоціація дозволяє ввести поняття основного актора (він є ініціатором асоціації) і другорядного актора (варіант використання є ініціатором, тобто передає актору довідкові відомості або звіт про виконану роботу).

Особливості використання відношення асоціації:

Один варіант використання може мати кілька асоціацій з різними акторами. Два варіанти використання, що відносяться до одного і того ж актора, не можуть бути асоційовані, оскільки кожен з них описує закінчений фрагмент функціональності актора.

**Узагальнення**

Відношення узагальнення (generalization) – показує, що нащадок успадковує атрибути і поведінку свого прямого предка, тобто один елемент моделі є спеціальним або частковим випадком іншого елемента моделі. Може застосовуватися як для акторів, так для варіантів використання.

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

**Відношення включення та розширення**

Відношення включення

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 називають базовим, а незалежний – включуваним. Включення означає, що кожне виконання варіанта використання А завжди буде включати в себе виконання варіанта використання Б. На практиці відношення включення використовується для моделювання ситуації, коли існує загальна частина поведінки двох або більше варіантів використання.

Загальна частина виноситься в окремий варіант використання, тобто типовий приклад повторного використання функціональності.

Особливості використання відношення включення:

Один базовий варіант використання може бути пов'язаний відношенням включення з декількома включуваними варіантами використання.

Один варіант використання може бути включений в інші варіанти використання.

На одній діаграмі варіантів використання не може бути замкнутого шляху по відношенню включення.

**Відношення розширення**

Відношення розширення (extend) – показує, що варіант використання розширює базову послідовність дій і вставляє власну послідовність. При цьому на відміну від типу відносин "включення" розширена послідовність може здійснюватися в залежності від певних умов.

Графічно зображення - пунктирна стрілка направлена від залежного варіанта (розширюючого) до незалежного варіанта (базового) з ключовим словом

<<extend>>.

Відношення розширення дозволяє моделювати той факт, що базовий варіант використання може приєднувати до своєї поведінки деякі додаткові поведінки за рахунок розширення в варіанті іншому варіанті використання.

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

Особливості використання відношення розширення:

Один базовий варіант використання може мати кілька точок розширення, з кожною з яких повинен бути пов'язаний розширюючий варіант використання.

Один розширюючий варіант використання може бути пов'язаний відношенням розширення з декількома базовими варіантами використання.

Розширюючий варіант використання може, в свою чергу, мати власні розширюючі варіанти використання.

На одній діаграмі варіантів використання не може бути замкнутого шляху по відношенню розширення.

**Сценарії використання**

Діаграма варіантів використання надає знання про необхідну функціональність кінцевої системи в інтуїтивно-зрозумілому вигляді, однак не несе відомостей

про фактичний спосіб її реалізації. Конкретні варіанти використань можуть звучати занадто загально і розпливчасто і не є придатними для програмістів. Для документації варіантів використання у вигляді деякої специфікації і для усунення неточностей та непорозумінь діаграм варіантів використань, як частина процесу збору та аналізу вимог складаються так звані сценарії використання.

Сценарії використання — це текстові представлення 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 не мають загального, шаблонного виду написання, однак рекомендується наступний вигляд:

- Передумови — умови, які повинні бути виконані для виконання даного варіанту використання

- Постумови — що отримується в результаті виконання даного варіанту використання
- Взаємодіючі сторони
- Короткий опис
- Основний хід подій
- Винятки
- Примітки

**Діаграми класів. Концептуальна модель системи**

Діаграми класів використовуються при моделюванні ПС найбільш часто. Вони є однією з форм статичного опису системи з точки зору її проектування, показуючи її структуру. Діаграма класів не відображає динамічну поведінку об'єктів зображених на ній класів. На діаграмах класів показуються класи, інтерфейси і відношення між ними.

Представлення класів

Клас – це основний будівельний блок ПС. Це поняття присутнє і в ОО мовах програмування, тобто між класами UML і програмними класами є відповідність, яка є основою для автоматичної генерації програмних кодів або для виконання реінжинірингу. Кожен клас має назву, атрибути та операції. Клас на діаграмі показується у вигляді прямокутника, розділеного на 3 області. У верхній міститься назва класу, в середній – опис атрибутів (властивостей), в нижній – назви операцій – послуг, що надаються об'єктами цього класу.

Атрибути та операції класу

Атрибути класу визначають склад і структуру даних, що зберігаються в

об'єктах цього класу. Кожен атрибут має ім'я і тип, що визначає, які дані він представляє. При реалізації об'єкта в програмному коді для атрибутів буде виділена пам'ять, необхідна для зберігання всіх атрибутів, і кожен атрибут матиме конкретне значення в будь-який момент часу роботи програми. Об'єктів одного класу в 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 атрибута класу можна задати видимість (visibility). Ця характеристика показує, чи доступний атрибут для інших класів. В UML визначені наступні рівні видимості атрибутів:

- Відкритий (public) – атрибут видно для будь-якого іншого класу (об'єкта)
- Захищений (protected) – атрибут видно для нащадків даного класу
- Закритий (private) – атрибут не видно зовнішнім класам (об'єктам) і може використовуватися тільки об'єктом, що його містить

Останнє значення дозволяє реалізувати властивість інкапсуляції даних. Наприклад, оголосивши всі атрибути класу закритими, можна повністю приховати від зовнішнього світу його дані, гарантуючи відсутність несанкціонованого доступу до них. Це дозволяє скоротити число помилок у програмі. При цьому будь-які зміни в складі атрибутів класу ніяк не позначаться на решті частини ПС.

**Відношення між класами**

На діаграмах класів зазвичай показуються асоціації та узагальнення.

Кожна асоціація несе інформацію про зв'язки між об'єктами всередині ПС. Найбільш часто використовуються бінарні асоціації, що зв'язують два класи. Асоціація може мати назву, яка повинна виражати суть відображуваного

зв'язку. Крім назви, асоціація може мати таку характеристику, як множинність. Вона показує, скільки об'єктів кожного класу може брати участь в асоціації. Множинність вказується у кожного кінця асоціації (полюса) і задається конкретним числом або діапазоном чисел. Множинність, вказана у вигляді зірочки, передбачає будь-яку кількість (в тому числі, і нуль).

Види відношень

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

Узагальнення (наслідування) на діаграмах класів використовується, щоб показати зв'язок між класом-батьком і класом-нащадком. Воно вводиться на діаграму, коли виникає різновид якогось класу, а також у тих випадках, коли в

системі виявляються кілька класів, що володіють схожою поведінкою (в цьому випадку загальні елементи поведінки виносяться на більш високий рівень, утворюючи клас-батько).

Агрегацією позначається відношення "has-a", коли об'єкти одного класу входять в об'єкт іншого класу. Типовим прикладом такого відношення є списки об'єктів. У даному випадку список буде виступати агрегатом, а об'єкти, що входять у список, агрегованими елементами.

Композицією позначається відношення "owns-a". По своїй суті воно нагадує агрегацію, однак позначає більш тісний зв'язок між агрегатом і агрегованими елементами. Прикладом композиції може служити зв'язок між машиною і карбюратором: машина не буде функціонувати без карбюратора (тому відношення композиції). Список, в свою чергу, не втрачає своїх функцій без окремих елементів списку (тому відношення агрегації).

**Логічна структура бази даних**

Існують дві моделі бази даних — логічна та фізична. Фізична модель представляє набір двійкових даних у вигляді файлів, структурованих та згрупованих відповідно до призначення (сегменти, екстенти тощо), використовуючи їх для швидкого доступу до інформації та ефективного її зберігання. Логічна модель є структурою таблиць, уявлень, індексів та інших логічних елементів бази даних, що використовуються для програмування та роботи з базою.

Процес створення логічної моделі бази даних називається проектуванням бази даних. Проектування відбувається в тісному зв'язку з розробкою архітектури програмної системи, оскільки база створюється для зберігання даних, що надходять від програмних класів.

Є кілька підходів до зв'язування програмних класів із таблицями:

- Одна таблиця — один клас.
- Одна таблиця — кілька класів.
- Один клас — кілька таблиць.

Від вибору підходу залежить складність роботи з базою даних. Програмні класи представляють сутності проектованої системи, а таблиці — технічну реалізацію їх зберігання.

**Нормальні форми**

Нормальна форма — це властивість відношення в реляційній моделі даних, яка характеризує його з точки зору надлишковості, що може призвести до помилок у вибірках або змінах даних. Нормалізація — це процес приведення структури

бази даних до нормальних форм, що мінімізує логічну надлишковість та потенційні протиріччя.

**Основні нормальні форми:**

Перша нормальна форма (1НФ): кожен атрибут у відношенні містить тільки одне значення.

Друга нормальна форма (2НФ): кожен неключовий атрибут залежить від ключа функціонально повно.

Третя нормальна форма (3НФ): немає транзитивних залежностей неключових атрибутів від ключа.

Нормальна форма Бойса-Кодда (BCNF): кожна функціональна залежність має в якості детермінанта потенційний ключ.

Нормалізація спрямована на виключення надлишковості та аномалій оновлення даних, забезпечуючи логічну чистоту моделі бази даних.

## Хід роботи::

## Завдання 2:

### Короткий опис діаграми 1:

Актори – ‘Адміністратор’ та ‘Користувач’: взаємодіють з системою. Адміністратор унаслідує усі варіанти використання Користувача. Між акторами реалізовано відношення узагальнення.

Варіанти використання – конкретні дії, які може виконувати система, також поведінка системи. Авторизація - процес, за якого користувач отримує доступ до системи, або не отримує. Створення користувачів - створення нових користувачів FTP-сервера.

Налаштування прав доступу - включає можливість доступу, зокрема до папок.

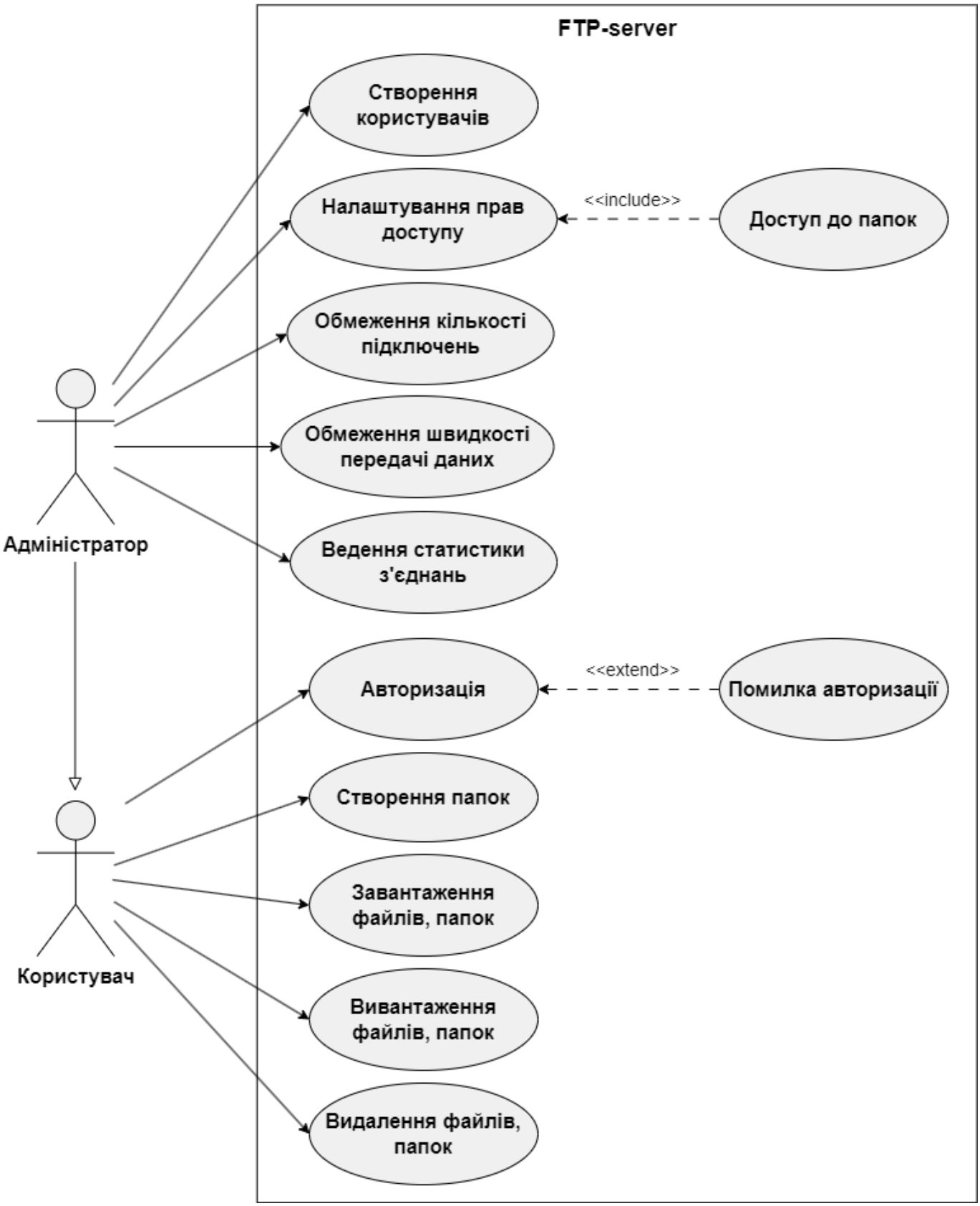
Обмеження кількості підключень – обмеження кількість одночасних підключень до сервера. Обмеження швидкості передачі даних - ліміти на швидкість передачі файлів.

Ведення статистики з'єднань - запис інформації про з'єднання користувачів.

Створення папок – дозволяє створювати нові папки на FTP-сервері. Вивантаження файлів, папок - вивантаження файлів на сервер. Завантаження файлів, папок – дозволяє користувачам завантажувати файли та папки з сервера. Видалення файлів, папок - дозволяє видаляти файли та папки на сервері.

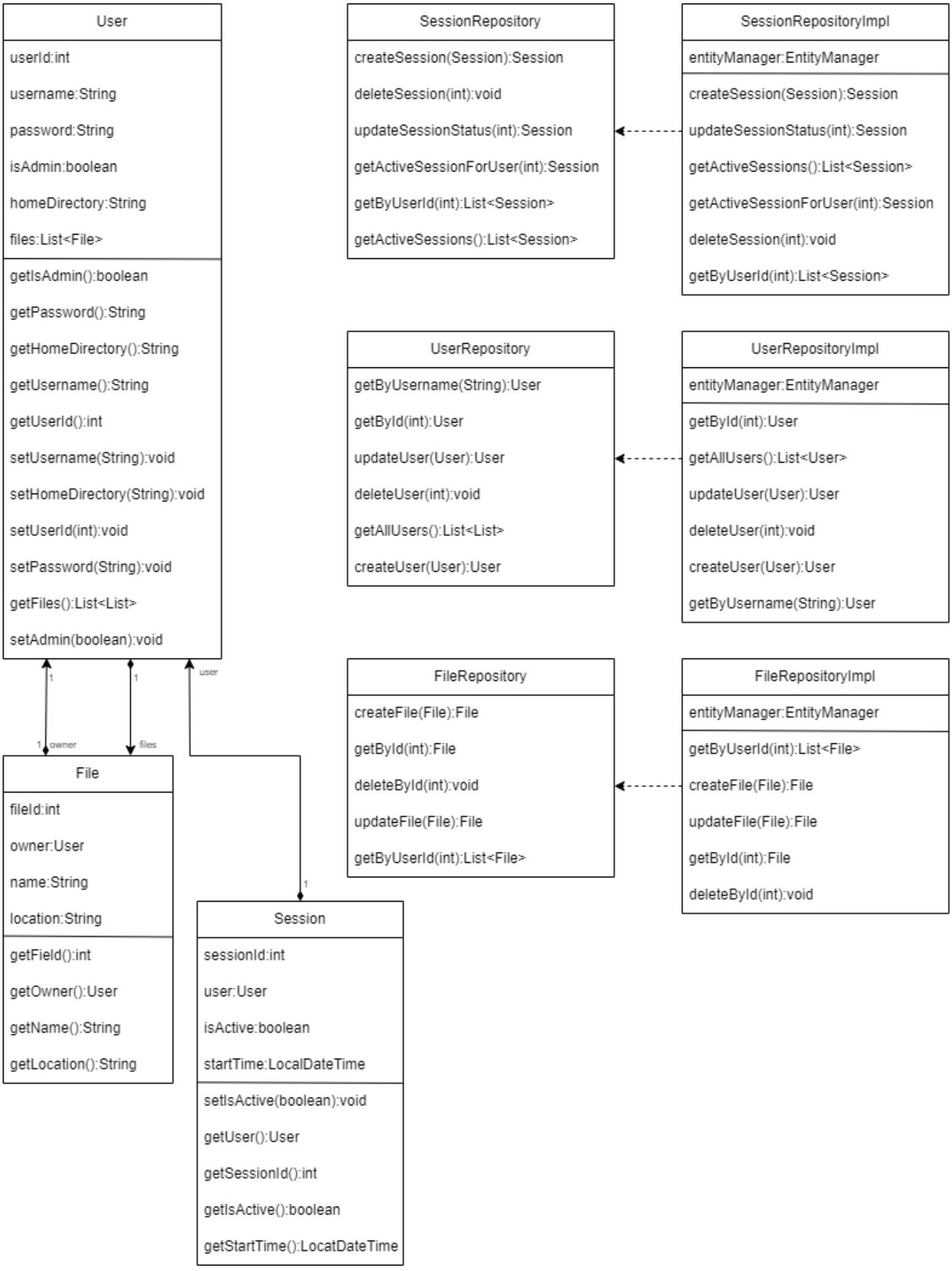
Пунктирна лінія з написом «extend» представляє собою можливе розширення варіанта Авторизація іншою дією Помилка авторизації.

Пунктирна лінія з написом «include» представляє собою можливе включення дії Доступ до папок у варіант використання Налаштування прав доступу.




Діаграма 1 – схема прецеденту (використання)

**Завдання 3:**




Діаграма 2 – основні класи та методи класів

**Короткий опис діаграми 2:**

Діаграма 2 зображає усі основні класи – User, File, Session, які керуються через відповідні репозиторії; структура керування об’єктами реалізована за допомогою Repository Pattern.

Між класами встановлено композиційних зв’язок. Композиційний зв’язок – якщо User буде видалено, то всі його файли також будуть видалені.

Стрілки між класами демонструють наступні відношення: 1. Композиція від Session до User – кожна сесія належить окремому користувачу, якщо сесія закінчується, то данні про користувача стануть недоступними. 2. Композиція від User до File – кожен користувач може володіти файлами, якщо користувача буде видалено, то і всі його файли також. 3. Композиція від File до User – кожен файл пов’язаний із конкретним користувачем, якщо буде видалено файл – інформація про користувача залишається, але зв’язок із файлом буде втрачено.

Репозиторії - SessionRepository: інтерфейс для управління об'єктами сесії. Він визначає методи для додавання, пошуку та видалення сесій. UserRepository: інтерфейс для роботи з даними користувачів. Включає методи для зберігання, пошуку даних про користувачів. FileRepository: інтерфейс для управління файлами на сервері. Визначає методи для збереження, видалення і отримання файлів.

Імплементації - SessionRepositoryImpl: реалізація інтерфейсу SessionRepository. Містить конкретні реалізації методів для управління сесіями. UserRepositoryImpl: реалізація інтерфейсу UserRepository. Визначає конкретні дії з користувачами, наприклад, роботу з базою даних. FileRepositoryImpl: реалізація інтерфейсу FileRepository. Забезпечує збереження, видалення і отримання файлів.

Стрілки між інтерфейсами та їхніми реалізаціями вказують на те, що реалізації імплементують відповідні інтерфейси та реалізації надають фактичну логіку для операцій із моделями.

**Завдання 4:**

Прецедент 1 – «Авторизація в систему».

| | |
|---|---|
| Назва | Авторизація в систему |
| Опис | Цей прецедент описує процес авторизації користувачів на FTP-сервері. |

| | |
|---|---|
| Взаємодіючі сторони | - Адміністратор або користувач<br>- FTP-сервер |
| Частота користування | Постійно |
| Передумови | Адміністратор або користувач звернувся до сервера для доступу. |
| Постумови | Сервер надає доступ або відмовляє у доступі. |
| Основний розвиток | 1. Користувач або адміністратор надсилає запит на підключення до сервера.<br>2. Сервер вимагає введення імені користувача та пароля.<br>3. Користувач або адміністратор вводять ім'я користувача та пароль.<br>4. Сервер перевіряє коректність імені користувача та пароля в базі даних.<br>5. Якщо ім'я користувача та пароль вірні, сервер надає доступ.<br>6. Якщо ім'я користувача та пароль невірні, сервер відмовляє у доступі.<br>7. Процес завершується. |
| Виняткові ситуації | Якщо сервер не може з'єднатися з базою даних для перевірки авторизації, то він може відмовити у доступі з повідомленням про помилку. |

Прецедент 2 – «Створення користувачів для системи»

| | |
|---|---|
| Назва | Створення користувачів адміністратором для системи |
| Опис | Цей прецедент описує процес створення нового користувача адміністратором. |
| Взаємодіючі сторони | - Адміністратор<br>- FTP-сервер |
| Частота користування | Часто |

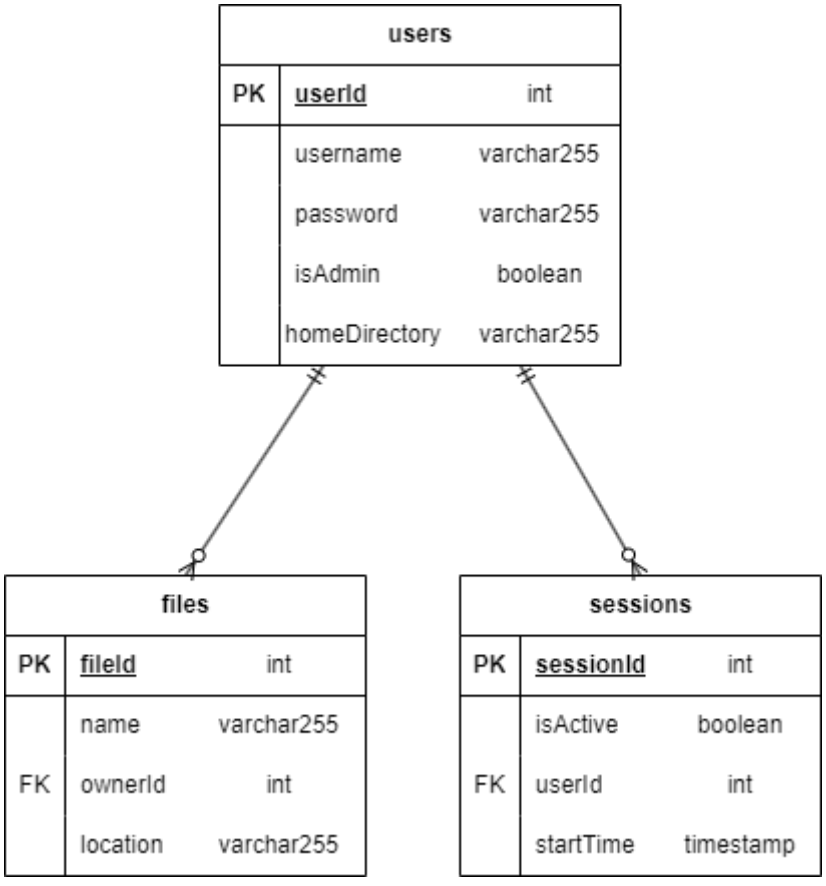
| | |
|---|---|
| Передумови | Адміністратор має права на створення користувачів. |
| Постумови | Новий користувач створений у системі. |
| Основний розвиток | 1. Адміністратор входить до системи.<br>2. Адміністратор надає команду створення нового користувача.<br>3. Система запитує ім'я та пароль для нового користувача.<br>4. Адміністратор вводить інформацію про нового користувача.<br>5. Система створює нового користувача в базі даних.<br>6. Процес завершується. |
| Виняткові ситуації | Якщо адміністратор не має прав на створення користувачів, система відмовить у доступі та повідомить про це адміністратора. |

Прецедент 3 – «Завантаження файлів в систему»

| | |
|---|---|
| Назва | Завантаження файлів в систему |
| Опис | Цей прецедент описує процес завантаження файлів на FTP-сервер користувачем або адміністратором. |
| Взаємодіючі сторони | - Користувач або адміністратор<br>- FTP-сервер |
| Частота користування | Постійно |
| Передумови | Користувач або адміністратор має підключення до FTP-сервера. |
| Постумови | Файли успішно завантажені на FTP-сервер. |
| Основний розвиток | 1. Користувач або адміністратор здійснює підключення до FTP-сервера.<br>2. Користувач/адміністратор переходить до потрібної директорії на сервері, де вони бажають завантажити файл.<br>3. Система очікує вибору файлів для завантаження. |

| | |
|---|---|
| | 4. Користувач/адміністратор вибирає файл(и) для завантаження.<br>5. Система завантажує вибрані файли на FTP-сервер.<br>6. Після завершення завантаження система оновлює інформацію про файли та їх статуси.<br>7. Процес завершується. |
| Виняткові ситуації | Якщо під час завантаження файлів виникають помилки (наприклад, недостатньо місця на сервері, втрата з'єднання), система повідомляє користувача або адміністратора про це та надає інформацію про причину невдалого завантаження. |

**Завдання 5:**




Діаграма 3 – структура системи бази даних

Дана діаграма ілюструє три основні сутності - users, files, і sessions, а також їхні зв’язки та атрибути. Зв’язок users – files: один користувач може бути власником або створювати багато файлів (зв’язок один-до-багатьох). Зв’зок users – sessions: один користувач може мати багато сесій (зв’язок один-до-багатьох).

Кожна сутність має Primary key – унікальний ідентифікатор запису в таблиці. Сутності files та sessions мають Foreign key – посилання на таблицю users та відповідно зв’язок із сутністю.

**Вихідні коди усіх наявних класів системи:**

```
@Entity
@Table(name="files")
public class File {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int fileId;

    @Column(nullable=false)
    private String name;

    @Column(nullable=false)
    private String location;

    @ManyToOne
    @JoinColumn(nullable = false, name="ownerId")
    private User owner;

    public File() { }

    public File(String name, String location, User owner) {
        this.name = name;
        this.location = location;
        this.owner = owner;
    }

    public int getFileId() {
        return fileId;
    }

    public String getName() {
        return name;
    }

    public String getLocation() {
        return location;
    }

    public User getOwner() {
        return owner;
    }

}
```

```
@Entity
@Table(name = "sessions")
public class Session {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int sessionId;

    @ManyToOne
    @JoinColumn(unique = true, name="userId")
    private User user;

    @Column
    private LocalDateTime startTime;
```

```
    @Column
    private boolean isActive;

    public Session() { }

    public Session(User user) {
        this.user = user;
        this.startTime = LocalDateTime.now();
        this.isActive = true;
    }

    public int getSessionId() {
        return sessionId;
    }

    public boolean getIsActive() {
        return isActive;
    }

    public User getUser() {
        return user;
    }

    public LocalDateTime getStartTime() {
        return startTime;
    }

    public void setIsActive(boolean isActive) {
        this.isActive = isActive;
    }

}
```

```
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int userId;

    @Column(nullable = false, unique = true)
    private String username;

    @Column(nullable = false)
    private String password;

    @Column(nullable = false)
    private boolean isAdmin;

    @Column(nullable = false)
    private String homeDirectory;

    @OneToMany(mappedBy = "owner", cascade = CascadeType.REMOVE, orphanRemoval =
true)
    private final List<File> files = new ArrayList<>();

    public User() { }

    public User(String username, String password, boolean isAdmin, String
homeDirectory) {
        this.username = username;
        this.password = password;
        this.isAdmin = isAdmin;
        this.homeDirectory = homeDirectory;
```

```
    }

    public int getUserId() {
        return userId;
    }

    public String getUsername() {
        return username;
    }

    public String getPassword() {
        return password;
    }

    public boolean getIsAdmin() {
        return isAdmin;
    }

    public String getHomeDirectory() {
        return homeDirectory;
    }

    public List<File> getFiles() {
        return files;
    }

    public void setUserId(int userId) {
        this.userId = userId;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public void setAdmin(boolean admin) {
        isAdmin = admin;
    }

    public void setHomeDirectory(String homeDirectory) {
        this.homeDirectory = homeDirectory;
    }

}
```

```
public interface FileRepository {
    File getById(int fileId);
    List<File> getByUserId(int userId);
    File createFile(File file);
    File updateFile(File modifiedFile);
    void deleteById(int fileId);

}
```

```
public interface SessionRepository {
    List<Session> getByUserId(int userId);
    Session createSession(Session session);
    Session updateSessionStatus(int sessionId);
    List<Session> getActiveSessions();
    Session getActiveSessionForUser(int userId);
    void deleteSession(int sessionId); }
```

```java
public interface UserRepository {
    User getById(int id);
    User getByUsername(String username);
    List<User> getAllUsers();
    User createUser(User user);
    User updateUser(User modifiedUser);
    void deleteUser(int userId);

}
```

```java
public class FileRepositoryImpl implements FileRepository {
    @PersistenceContext
    private final EntityManager entityManager;

    public FileRepositoryImpl() {
        EntityManagerFactory entityManagerFactory =
Persistence.createEntityManagerFactory("myPersistenceUnit");
        this.entityManager = entityManagerFactory.createEntityManager();
    }

    @Override
    public File getById(int fileId) {
        File file = entityManager.find(File.class, fileId);
        return file;
    }

    @Override
    public List<File> getByUserId(int userId) {
        TypedQuery<File> query = entityManager.createQuery(
                "SELECT f FROM File WHERE f.owner.userId = :userId", File.class);
        query.setParameter("userId", userId);
        return query.getResultList();
    }

    @Override
    public File createFile(File file) {
        EntityTransaction transaction = entityManager.getTransaction();
        transaction.begin();
        entityManager.persist(file);
        transaction.commit();
        return file;
    }

    @Override
    public File updateFile(File modifiedFile) {
        EntityTransaction transaction = entityManager.getTransaction();
        transaction.begin();
        File updatedFile = entityManager.merge(modifiedFile);
        transaction.commit();
        return updatedFile;
    }

    @Override
    public void deleteById(int fileId) {
        EntityTransaction transaction = entityManager.getTransaction();
        transaction.begin();
        File file = entityManager.find(File.class, fileId);
        if (file != null) {
            entityManager.remove(file);
            transaction.commit();
        }
    }

}
```

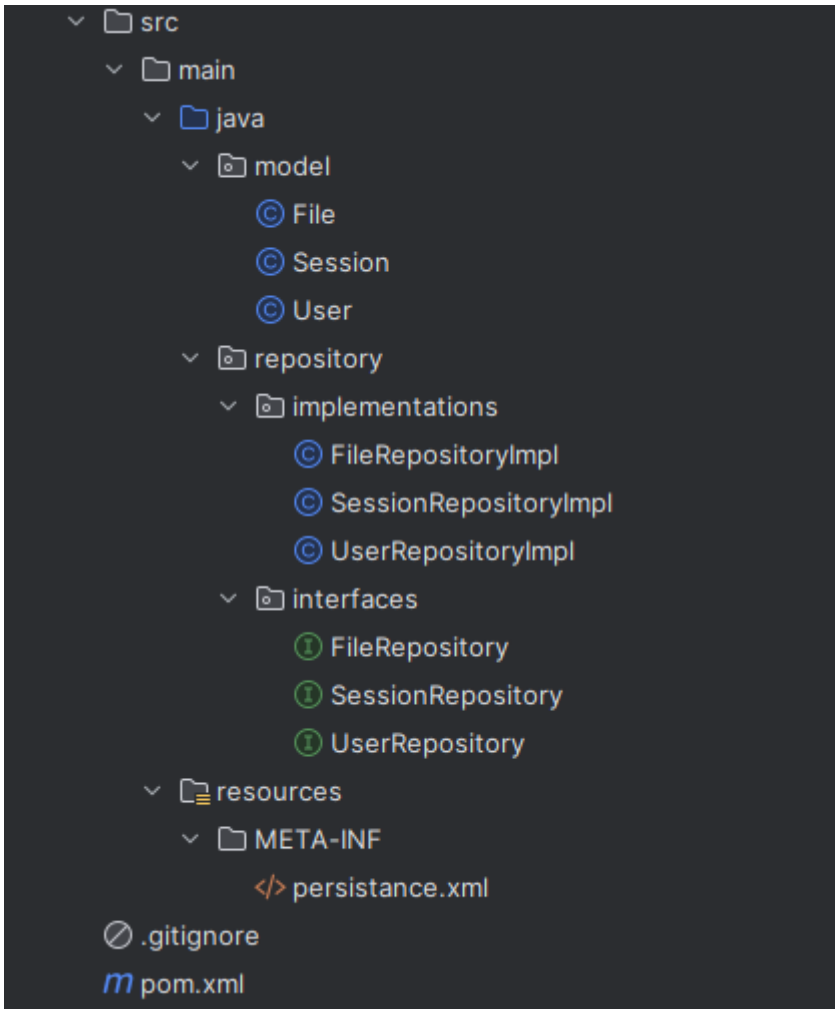
```java
public class SessionRepositoryImpl implements SessionRepository {
    @PersistenceContext
    private final EntityManager entityManager;

    public SessionRepositoryImpl(){
        EntityManagerFactory entityManagerFactory =
Persistence.createEntityManagerFactory("myPersistenceUnit");
        this.entityManager = entityManagerFactory.createEntityManager();
    }

    @Override
    public List<Session> getByUserId(int userId) {
        TypedQuery<Session> query = entityManager.createQuery(
                "SELECT s FROM Session s WHERE s.user.userId = :userId",
Session.class);
        query.setParameter("userId", userId);
        return query.getResultList();
    }

    @Override
    public Session createSession(Session session) {
        EntityTransaction transaction = entityManager.getTransaction();
        transaction.begin();
        entityManager.persist(session);
        transaction.commit();

        return session;
    }

    @Override
    public Session updateSessionStatus(int sessionId) {
        Session session = entityManager.find(Session.class, sessionId);
        if(session != null) {
            EntityTransaction transaction = entityManager.getTransaction();
            transaction.begin();
            session.setIsActive(false);
            session = entityManager.merge(session);
            transaction.commit();
        }
        return session;
    }

    @Override
    public List<Session> getActiveSessions() {
        String queryText = "SELECT s FROM Session s WHERE s.isActive = true";
        TypedQuery<Session> query = entityManager.createQuery(queryText,
Session.class);
        return query.getResultList();
    }

    @Override
    public Session getActiveSessionForUser(int userId) {
        try {
            String query = "SELECT s FROM Session s WHERE s.user.id = :userId AND
s.isActive = true";
            List<Session> sessions = entityManager.createQuery(query, Session.class)
                    .setParameter("userId", userId)
                    .setMaxResults(1)
                    .getResultList();

            return sessions.get(0);
        } catch (Exception e) {
            return null;
        }

    }
```

```
    @Override
    public void deleteSession(int sessionId) {
        EntityTransaction transaction = entityManager.getTransaction();
        transaction.begin();
        Session session = entityManager.find(Session.class, sessionId);
        if(session != null)
            entityManager.remove(session);
        transaction.commit();
    }
}
```

```
public class UserRepositoryImpl implements UserRepository {
    @PersistenceContext
    private final EntityManager entityManager;

    public UserRepositoryImpl(){
        EntityManagerFactory entityManagerFactory =
Persistence.createEntityManagerFactory("myPersistenceUnit");
        this.entityManager = entityManagerFactory.createEntityManager();
    }

    @Override
    public User getById(int userId) {
        User user = entityManager.find(User.class, userId);
        return user;
    }

    @Override
    public User getByUsername(String username) {
        User user = entityManager.createQuery("SELECT u FROM User u WHERE u.username
= :username", User.class)
                .setParameter("username", username)
                .setMaxResults(1)
                .getResultStream()
                .findFirst()
                .orElse(null);

        return user;
    }

    @Override
    public List<User> getAllUsers() {
        String queryText = "SELECT u FROM User u";
        TypedQuery<User> query = entityManager.createQuery(queryText, User.class);
        return query.getResultList();
    }

    @Override
    public User createUser(User user) {
        EntityTransaction transaction = entityManager.getTransaction();
        transaction.begin();
        entityManager.persist(user);
        transaction.commit();
        return user;
    }

    @Override
    public User updateUser(User modifiedUser) {
        EntityTransaction transaction = entityManager.getTransaction();
        transaction.begin();
        User updatedUser = entityManager.merge(modifiedUser);
        transaction.commit();
        return updatedUser;
    }
```

```
    @Override
    public void deleteUser(int userId) {
        EntityTransaction transaction = entityManager.getTransaction();
        transaction.begin();
        User user = entityManager.find(User.class, userId);
        if(user != null){
            user.getFiles().clear();
            entityManager.remove(user);
        }
        transaction.commit();
    }
}
```

**Структура існуючої частини проекту:**




**Висновок:** під час виконання даної лабораторної роботи було розроблено та проаналізовано концептуальну модель системи з використанням UML- діаграм. Створення діаграми варіантів використання. Розробка детальних сценаріїв використання. Проектування діаграми класів. Створення концептуальної моделі бази даних.