



НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
КОРАБЛЕБУДУВАННЯ
ІМЕНІ АДМІРАЛА МАКАРОВА

Н. Р. КНИРІК, Є. Ю. БЕРКУНСЬКИЙ

БАЗИ ДАНИХ У СУЧАСНИХ КОРПОРАТИВНИХ ЗАСТОСУНКАХ

Навчальний посібник



*Н*серія
навчальні посібники

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет кораблебудування
імені адмірала Макарова

Н. Р. КНИРІК, Є. Ю. БЕРКУНСЬКИЙ

**БАЗИ ДАНИХ У СУЧАСНИХ
КОРПОРАТИВНИХ ЗАСТОСУНКАХ**

Навчальний посібник

Рекомендовано Вченою радою НУК



ВИДАВНИЦТВО
НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ
КОРАБЛЕБУДУВАННЯ
ІМ. АДМІРАЛА МАКАРОВА

2026

УДК 004.65(075)
К53

Автори:

Н. Р. Книрік, канд. техн. наук, доцент;
Є. Ю. Беркунський, старш. викладач

Рецензенти:

М. В. Ушкац, д-р фіз.-мат. наук, проф., завідувач кафедри фізики та математики Національного університету кораблебудування імені адмірала Макарова;
В. І. Передерій, д-р техн. наук, проф., професор кафедри комп'ютерних технологій та інформаційної безпеки Національного університету кораблебудування імені адмірала Макарова;
І. М. Журавська, д-р техн. наук, проф., завідувачка кафедри комп'ютерної інженерії Чорноморського національного університету імені Петра Могили

*Рекомендовано Вченою радою НУК ім. адмірала Макарова
як навчальний посібник
(протокол № 11 від 28.11.2025 р.)*

Книрік Н. Р.

К53 Бази даних у сучасних корпоративних застосунках : навчальний посібник / Н. Р. Книрік, Є. Ю. Беркунський. – Миколаїв : НУК, 2026. – 182 с.

ISBN 978-966-321-509-9

Вміщено теоретичні відомості та практичні приклади щодо проектування і реалізації баз даних у складі сучасних корпоративних застосунків. Наведено приклади із використанням СУБД MySQL та H2. Як мову програмування обрано Java.

Призначено для студентів та аспірантів галузі «Інформаційні технології». Також навчальний посібник може бути корисним для фахівців галузі та професійних розробників сучасних корпоративних застосунків.

УДК 004.65(075)

ISBN 978-966-321-509-9

© Н. Р. Книрік, Є. Ю. Беркунський, 2026
© Національний університет кораблебудування імені адмірала Макарова, 2026

ЗМІСТ

ВСТУП.....	4
РОЗДІЛ 1. ЖИТТЄВИЙ ЦИКЛ ІНФОРМАЦІЙНИХ СИСТЕМ.....	5
1.1. Поняття інформаційної системи.....	5
1.2. Планування бази даних.....	8
1.3. Опис предметної області.....	9
РОЗДІЛ 2. ПРОЄКТУВАННЯ БАЗИ ДАНИХ.....	18
2.1. Проєктування бази даних.....	18
2.2. Приклад проєкту інформаційної системи.....	38
2.3. Вступ до методології проєктування бази даних.....	49
2.4. ER-моделювання. Побудова концептуальної моделі даних.....	53
2.5. Побудова логічної моделі бази даних.....	73
2.6. Нормалізація.....	81
РОЗДІЛ 3. РЕАЛІЗАЦІЯ ЗАСТОСУНКУ З БД НА ПЛАТФОРМІ JAVA.....	95
3.1. Аналіз специфікації Java Persistence API.....	95
3.2. Вибір фреймворку: ORM, JPA, Hibernate та Spring Data.....	105
3.3. Прості проєкти з використанням JPA та Spring Data.....	112
3.4. Відображення персистентних класів.....	131
3.5. Відображення типів Java та SQL за допомогою конверторів.....	141
3.6. Шаблони роботи з персистентними даними.....	150
3.7. Створення інфраструктури застосунку Java для роботи з БД.....	157
3.8. Розробка компонентів для роботи із БД.....	159
СПИСОК ЛІТЕРАТУРИ.....	181

ВСТУП

Інформаційні системи є фундаментальними компонентами сучасного суспільства, забезпечуючи ефективне функціонування організацій, підприємств та установ. В умовах, коли інформація виступає ключовим ресурсом, здатність до її збору, збереження, обробки та аналізу визначає рівень конкурентоспроможності та перспективи розвитку суб'єктів господарювання. У зв'язку із цим дослідження життєвого циклу інформаційних систем набуває особливої ваги для фахівців у галузі інформаційних технологій, менеджменту та керівних кадрів різних сфер діяльності.

У посібнику аналізуються основні стадії життєвого циклу інформаційних систем, а також розглядаються особливості проєктування і впровадження систем баз даних як невід'ємного елементу сучасних інформаційних систем. Від етапу планування до експлуатаційного супроводу кожна стадія характеризується специфічними завданнями, викликами та методами їх вирішення, що обумовлює необхідність комплексного та системного підходу. Необхідно підкреслити, що інформаційна система розглядається не лише як технічний інструмент, а й як стратегічний чинник, що впливає на організаційну структуру, бізнес-процеси та процеси прийняття управлінських рішень.

У посібнику подано визначення основних понять, окреслено структурні компоненти інформаційної системи, а також детально проаналізовано роль бази даних у забезпеченні цілісності, надійності та ефективності функціонування всієї системи. Особлива увага приділяється інтеграції процесів розробки баз даних із загальною стратегією розвитку інформаційної системи організації з метою забезпечення її гнучкості, масштабованості та відповідності сучасним вимогам.

РОЗДІЛ 1

ЖИТТЄВИЙ ЦИКЛ ІНФОРМАЦІЙНИХ СИСТЕМ

1.1. Поняття інформаційної системи

Інформаційна система – це ресурси, які дозволяють збирати, управляти, контролювати та поширювати інформацію в організації.

Інформаційна система включає в себе базу даних, програмне забезпечення бази даних, прикладне програмне забезпечення, комп'ютерне обладнання та персонал, який використовує і розвиває систему.

База даних є фундаментальним компонентом інформаційної системи, і її розробку та використання слід розглядати з точки зору ширших потреб організації. Тому життєвий цикл інформаційної системи організації нерозривно пов'язаний із життєвим циклом системи баз даних, яка її підтримує. Зазвичай етапи життєвого циклу інформаційної системи включають у себе: планування, збір та аналіз вимог, проєктування, створення прототипу, впровадження, тестування, перетворення та експлуатаційне обслуговування. У цьому розділі ми розглянемо ці етапи з точки зору розробки системи баз даних. Однак важливо зазначити, що розробку системи баз даних слід також розглядати з більш широкої перспективи розробки складової частини більшої інформаційної системи організації.

Життєвий цикл розробки системи баз даних

Оскільки система баз даних є фундаментальним компонентом більшої інформаційної системи організації,

життєвий цикл розробки системи баз даних нерозривно пов'язаний із життєвим циклом інформаційної системи. Етапи життєвого циклу розробки системи баз даних показані на рис. 1.1.

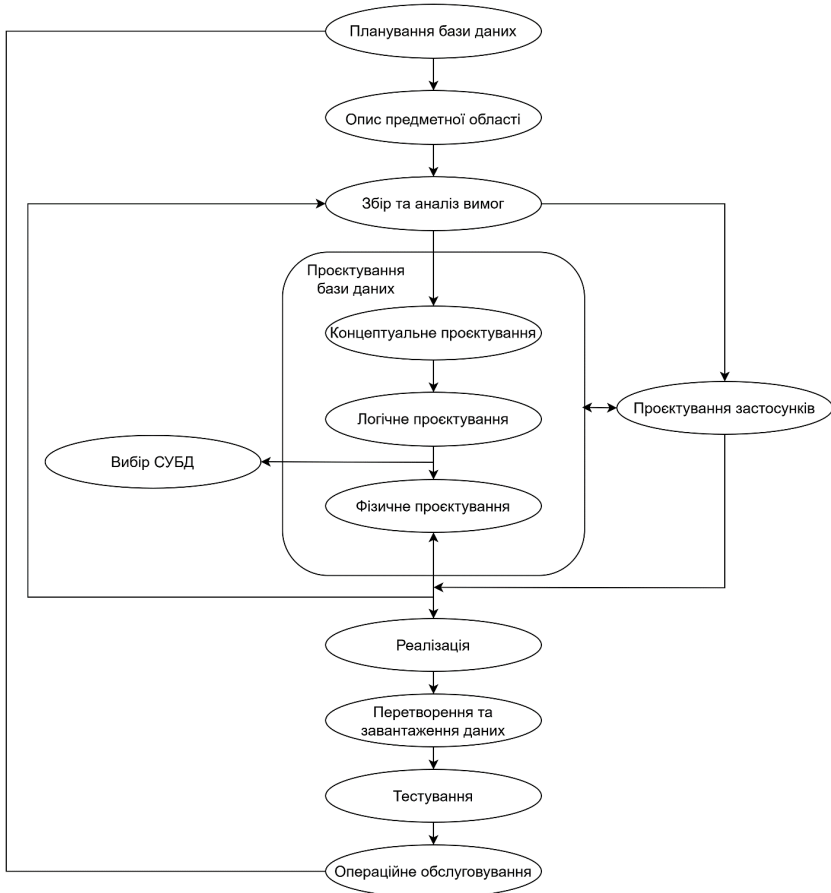


Рис. 1.1. Етапи життєвого циклу розробки інформаційної системи

Важливо розуміти, що етапи життєвого циклу розробки системи баз даних не є строго послідовними, а передбачають певне повторення попередніх етапів через петлі зворотного зв'язку. Наприклад, проблеми, що виникають під час проєктування бази даних, можуть вимагати додаткового збору та аналізу вимог. Оскільки між більшістю етапів існують петлі зворотного зв'язку, на рис. 1.1 показано лише деякі з найбільш очевидних з них.

Планування бази даних – це планування того, як етапи життєвого циклу можуть бути реалізовані найбільш ефективно й результативно.

Визначення системи – це визначення обсягу та меж системи баз даних, включаючи основні представлення користувачів та сфери застосування.

Збір та аналіз вимог – це збір та аналіз вимог до нової системи баз даних.

Проектування бази даних – це концептуальне, логічне та фізичне проектування бази даних.

Вибір СУБД – це вибір відповідної системи управління базами даних (СУБД) для інформаційної системи.

Проектування застосунків – це проектування інтерфейсу користувача та прикладних програм, які використовують і обробляють базу даних.

Реалізація – це створення фізичної структури бази даних та прикладних програм.

Перетворення та завантаження даних – це завантаження даних зі старої системи в нову і, де це можливо, конвертація існуючих додатків для роботи з новою базою даних.

Тестування – система баз даних тестується на наявність помилок і перевіряється на відповідність вимогам, визначеним користувачами.

Операційне обслуговування – система баз даних повністю впроваджена. Система постійно контролюється та

підтримується. За потреби нові вимоги вносяться до системи баз даних на попередніх етапах життєвого циклу.

Для невеликих систем баз даних з невеликою кількістю користувачів життєвий цикл не повинен бути дуже складним. Однак під час розробки середніх і великих систем баз даних з десятками та тисячами користувачів, які використовують сотні запитів і прикладних програм, життєвий цикл може стати надзвичайно складним. У наступних розділах ми більш детально опишемо основні дії, пов'язані з кожним етапом життєвого циклу розробки системи баз даних.

1.2. Планування бази даних

Планування бази даних – це управлінська діяльність, яка дозволяє реалізувати етапи життєвого циклу розробки системи, що використовує базу даних максимально ефективно.

Планування бази даних повинно бути інтегроване із загальною стратегією ІС організації. Існує три основні питання, пов'язані з формулюванням стратегії ІС, а саме:

- ідентифікація планів і цілей підприємства з подальшим визначенням потреб в інформаційних системах;
- оцінка поточних інформаційних систем для визначення існуючих сильних і слабких сторін;
- оцінка ІТ-можливостей, які можуть забезпечити конкурентну перевагу.

Важливим першим кроком у плануванні бази даних є чітке визначення головної задачі інформаційної системи, ядром якої буде база даних. Формулювання головної задачі визначає основні цілі системи баз даних. Зазвичай головну задачу інформаційної системи визначають ті, хто керує проектом створення бази даних в організації (наприклад, директор та/або власник). Формулювання головної задачі допомагає прояснити мету системи баз даних і забезпечити більш

чіткий шлях до ефективного та результативного створення необхідної системи баз даних. Після визначення головної задачі інформаційної системи наступним кроком є визначення цілей. Кожна мета повинна визначати конкретну задачу, яку повинна підтримувати система баз даних. Формулювання головної задачі та цілей можуть супроводжуватися деякою додатковою інформацією, яка в загальних рисах визначає роботу, яку потрібно виконати, ресурси, за допомогою яких це можна зробити, і гроші, щоб заплатити за все це.

Планування бази даних має також включати в себе розробку стандартів, які визначають, як збиратимуться дані, як визначатиметься їхній формат, яка документація буде необхідна, а також як відбуватиметься розробка та впровадження. Розробка та підтримка стандартів може зайняти багато часу, вимагаючи ресурсів для їх початкового встановлення та подальшої підтримки контролю якості, і може забезпечити відповідність роботи шаблону, незалежно від навичок і досвіду персоналу. Наприклад, конкретні правила можуть визначати, як називати елементи даних у словнику даних, що зі свого боку може запобігти як надмірності, так і неузгодженості. Будь-які юридичні або корпоративні вимоги щодо даних мають бути задокументовані, наприклад, положення про те, що з деякими типами даних слід поводитися конфіденційно.

1.3. Опис предметної області

Опис предметної області визначає сферу застосування і межі системи баз даних, а також основні уявлення користувачів.

Перш ніж приступити до проєктування системи баз даних, необхідно спочатку визначити межі системи, яку ми досліджуємо, і те, як вона взаємодіє з іншими частинами інформаційної системи організації. Важливо, щоб ми

включили в межі нашої системи не тільки поточних користувачів і прикладні області, але й майбутніх користувачів та прикладні області. Ми представляємо діаграму, яка відображає сферу застосування та межі системи баз даних. До сфери застосування та меж системи бази даних входять основні представлення користувачів, які повинні підтримуватися базою даних.

Представлення користувача

Представлення користувача – визначає, що вимагається від системи баз даних з точки зору певної посадової ролі (наприклад, менеджера або керівника) або сфери застосування на підприємстві (наприклад, маркетинг, управління персоналом або контроль запасів).

Система баз даних може мати одне або кілька представлень користувача. Ідентифікація представлень користувачів є важливим аспектом розробки системи баз даних, оскільки вона допомагає гарантувати, що жоден з основних користувачів бази даних не буде забутий під час розробки вимог до нової системи баз даних. Представлення користувачів також особливо корисні під час розробки відносно складних систем баз даних, оскільки дозволяють розбити вимоги на керовані частини.

Представлення користувача визначає, що вимагається від системи баз даних з точки зору даних, які будуть зберігатися, і операцій, які будуть виконуватися над цими даними (іншими словами, що користувачі будуть робити з цими даними). Вимоги користувацького представлення можуть бути відмінними для цього представлення або перетинатися з іншими представленнями. На рис. 1.2 показано схематичне представлення системи бази даних з декількома представленнями користувача (позначено представленнями користувача від 1 до 6).

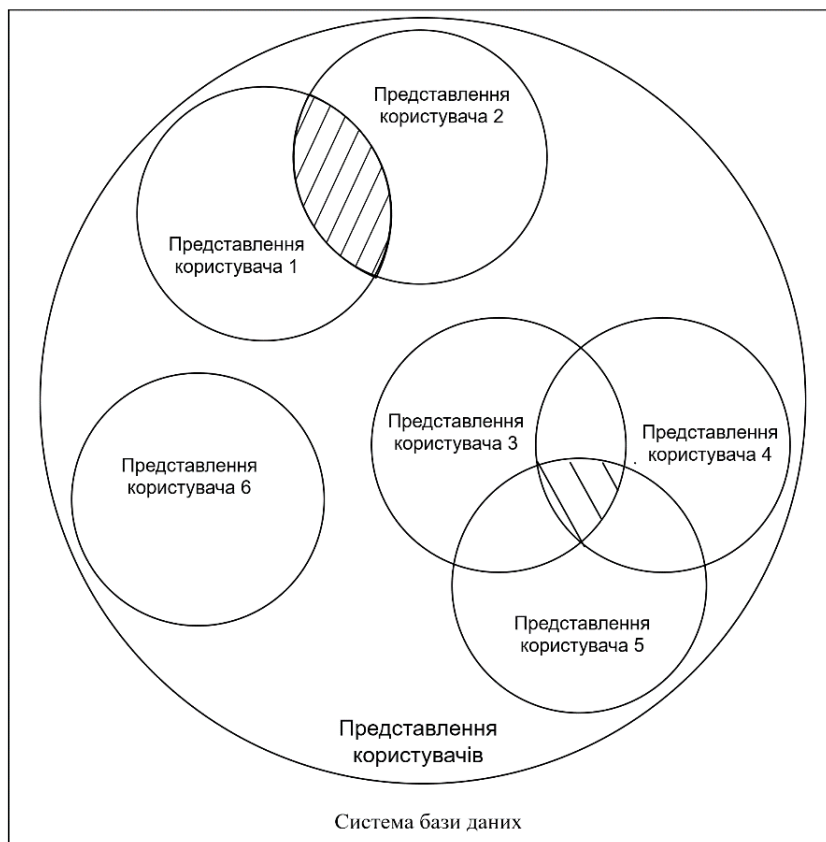


Рис. 1.2. Інформаційна система з кількома представленнями користувачів

Зауважте, що в той час як представлення користувача (1 і 2) і (3, 4 і 5) мають вимоги, що перетинаються (показано заштрихованими областями), представлення користувача 6 має окремі вимоги.

Збір та аналіз вимог

Збір та аналіз вимог – це процес збору та аналізу інформації про частину організації, яка повинна підтримуватися системою баз даних, і використання цієї інформації для визначення вимог до нової системи.

Цей етап передбачає збір та аналіз інформації про ту частину підприємства, яку буде обслуговувати база даних. Існує багато методів збору цієї інформації, які називаються *методами встановлення фактів*. Інформація збирається для кожної основної точки зору користувача (тобто посадової ролі або області застосування підприємства), включаючи:

- опис даних, що використовуються або генеруються;
- деталі того, як дані будуть використовуватися або генеруватися;
- будь-які додаткові вимоги до нової системи баз даних.

Потім ця інформація аналізується для визначення вимог (або функцій), які мають бути включені в нову систему баз даних. Ці вимоги описуються в документах, які разом називаються *специфікаціями вимог до нової системи баз даних*.

Збір та аналіз вимог є попереднім етапом проєктування бази даних. Обсяг зібраних даних залежить від характеру проблеми та політики підприємства. Занадто велике дослідження занадто рано призводить до паралічу аналізу. Занадто мало продуманості може призвести до непотрібної витрати часу і грошей через роботу над неправильним рішенням неправильної проблеми.

Інформація, зібрана на цьому етапі, може бути погано структурованою і включати деякі неформальні запити, які необхідно перетворити на більш структурований виклад вимоги. Це досягається за допомогою методів специфікації вимог, які включають у себе, наприклад, методи структурного аналізу та проєктування (SAD), діаграми потоків даних

(DFD) та діаграми ієрархічного вхідного процесу-виходу (HIPO), що підтримуються документацією. Інструменти автоматизованої розробки програмного забезпечення (CASE) можуть надавати автоматизовану допомогу для забезпечення повноти та узгодженості вимог. Уніфікована мова моделювання (UML) підтримує аналіз та проєктування вимог.

Визначення необхідної функціональності для системи баз даних є критично важливою діяльністю, оскільки системи з недостатньою або неповною функціональністю дратуватимуть користувачів, що може призвести до відмови або недостатнього використання системи. Однак надмірна функціональність також може бути проблематичною, оскільки вона може надмірно ускладнити систему, ускладнюючи її впровадження, підтримку, використання або навчання.

Ще однією важливою діяльністю, пов'язаною з цим етапом, є вирішення питання про те, як поводитися із ситуацією, коли для системи баз даних існує більше одного перегляду користувача.

Існує три основні підходи до управління вимогами системи баз даних з кількома переглядами користувачів:

- централізований підхід;
- підхід інтеграції представлень;
- поєднання обох підходів.

Централізований підхід – вимоги кожного представлення користувача об'єднуються в єдиний набір вимог до нової системи баз даних. На етапі проєктування бази даних створюється модель даних, що представляє всі вимоги користувачів.

Централізований (або одномоментний) підхід передбачає об'єднання вимог різних представлень користувачів в єдиний список вимог. Колекції представлень користувачів дається ім'я, яке дає деяке уявлення про область застосування, що охоплюється всіма об'єднаними представленнями користувачів.

На етапі проєктування бази даних створюється глобальна модель даних, яка представляє всі представлення користувачів. Глобальна модель даних складається з діаграм і документації, які формально описують вимоги користувачів до даних. Діаграма, що представляє управління представленнями користувачів з використанням централізованого підходу, показана на рис. 1.3. Зазвичай цьому підходу надають перевагу, коли вимоги до кожного представлення користувача значно збігаються, а система баз даних не є надто складною.

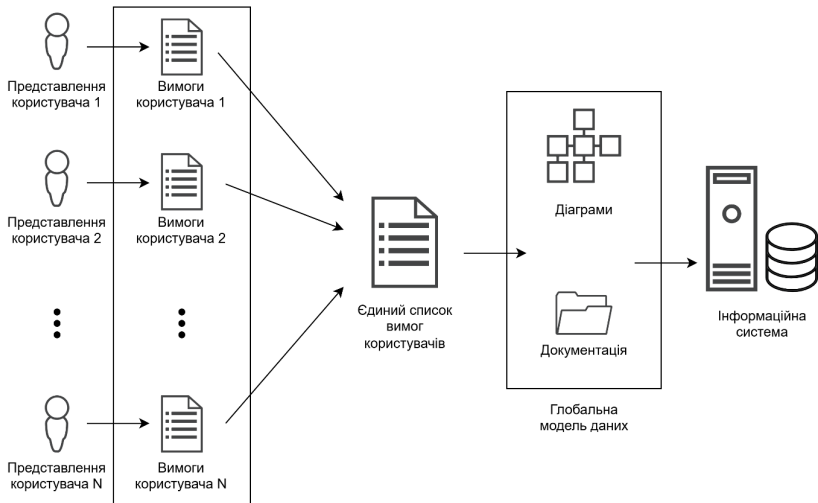


Рис. 1.3. Централізований підхід до управління вимогами з кількома представленнями користувачів

Підхід інтеграції представлень – вимоги для кожного представлення користувача залишаються у вигляді окремих списків. Моделі даних, що представляють кожне представлення користувача, створюються, а потім об’єднуються пізніше на етапі проєктування бази даних.

Підхід інтеграції представлень передбачає, що вимоги для кожного представлення користувача залишаються у вигляді окремих списків вимог. На етапі проєктування бази даних ми спочатку створимо модель даних для кожного подання користувача. Модель даних, яка представляє одне представлення користувача (або підмножину всіх представлень користувача), називається *локальною моделлю даних*.

Кожна модель складається з діаграм і документації, яка формально описує вимоги одного або декількох (але не всіх) користувацьких представлень бази даних. Локальні моделі даних потім об'єднуються на пізнішому етапі проєктування бази даних для створення глобальної моделі даних, яка представляє всі вимоги користувачів до бази даних. Діаграма, що представляє управління представленнями користувачів з використанням підходу інтеграції представлень, показана на рис. 1.4. Зазвичай цьому підходу надають перевагу тоді, коли існують значні відмінності між представленнями користувачів, а система бази даних є достатньо складною, щоб виправдати поділ роботи на більш керовані частини.

Для деяких складних систем баз даних може бути доцільним використання *комбінації централізованого підходу та підходу інтеграції представлень* для управління декількома представленнями користувача. Наприклад, вимоги до двох або більше представлень користувача можуть бути спочатку об'єднані за допомогою централізованого підходу, який використовується для побудови локальної логічної моделі даних. Потім ця модель може бути об'єднана з іншими локальними логічними моделями даних за допомогою підходу інтеграції представлень для створення глобальної логічної моделі даних. У цьому випадку кожна локальна логічна модель даних представляє вимоги двох або більше представлень користувача, а остаточно глобальна логічна модель даних представляє вимоги всіх представлень користувача системи баз даних.

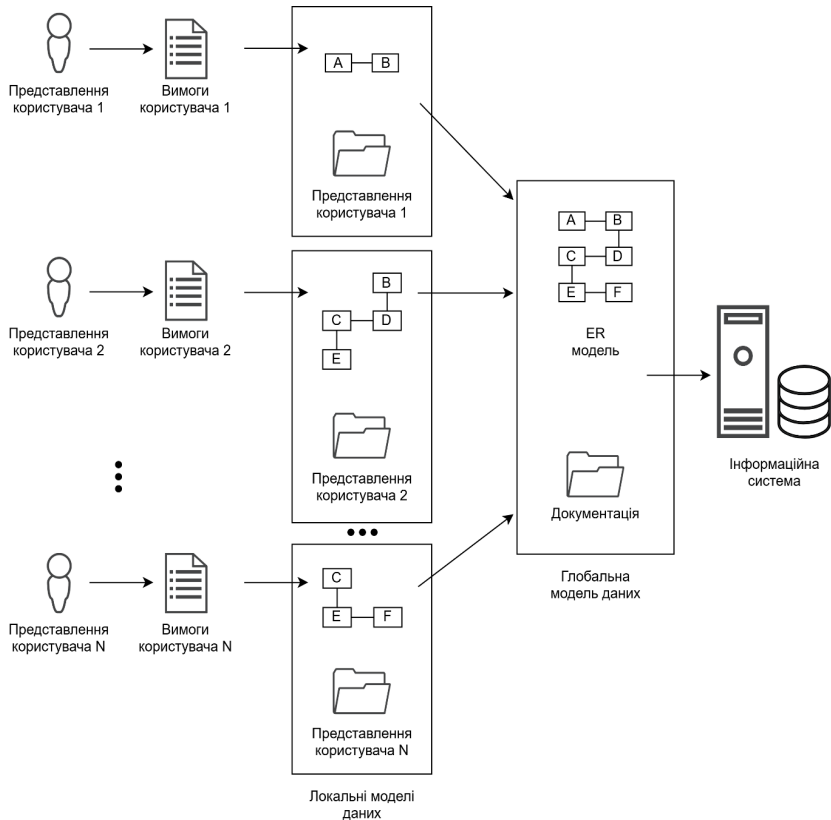


Рис. 1.4. Підхід інтеграції представлень кількох користувачів

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Дайте визначення інформаційної системи.
2. Назвіть основні етапи життєвого циклу розробки системи баз даних.
3. Чому важливо розглядати розробку БД як циклічний процес?
4. Які основні цілі та завдання етапу планування БД?
5. Які ключові фактори впливають на план розробки БД?

6. Які ресурси необхідно врахувати на етапі планування?
7. Що таке предметна область і яку роль її опис відіграє у створенні БД?
8. Які категорії користувачів ІС зазвичай виділяють і чим відрізняються їхні вимоги?
9. Назвіть основні методи збору вимог до системи.
10. Які типи вимог (функціональні та нефункціональні) необхідно враховувати?

РОЗДІЛ 2

ПРОЄКТУВАННЯ БАЗИ ДАНИХ

2.1. Проєктування бази даних

Проєктування бази даних – це процес створення проєкту, який буде підтримувати формулювання головної задачі та цілей для необхідної системи баз даних.

У цьому розділі ми представляємо огляд основних підходів до проєктування баз даних. Ми також обговоримо призначення і використання моделювання даних під час проєктування баз даних. Потім ми опишемо три етапи проєктування баз даних: концептуальне, логічне та фізичне проєктування.

Два основні підходи до проєктування бази даних називаються «знизу вгору» і «зверху вниз». Висхідний підхід починається з фундаментального рівня атрибутів (тобто властивостей сутностей і зв'язків), які шляхом аналізу асоціацій між атрибутами групуються у зв'язки, що представляють типи сутностей і зв'язки між сутностями. Далі буде розглядатися висхідний підхід до проєктування бази даних – процес нормалізації. Нормалізація передбачає визначення необхідних атрибутів та їх подальше об'єднання в нормалізовані відношення на основі функціональних залежностей між атрибутами.

Висхідний підхід підходить для проєктування простих баз даних з відносно невеликою кількістю атрибутів. Однак цей підхід стає складним під час проєктування більш складних баз даних з більшою кількістю атрибутів, де важко встановити всі функціональні залежності між атрибутами.

Оскільки концептуальні та логічні моделі даних для складних баз даних можуть містити від сотень до тисяч атрибутів, важливо визначити підхід, який спростить процес проектування. Крім того, на початкових етапах визначення вимог до даних для складної бази даних може бути важко визначити всі атрибути, які необхідно включити в моделі даних.

Більш доречною стратегією для проектування складних баз даних є використання низхідного підходу. Цей підхід починається з розробки моделей даних, які містять кілька високорівневих сутностей і зв'язків, а потім застосовуються послідовні низхідні уточнення для визначення сутностей, зв'язків і пов'язаних з ними атрибутів нижчих рівнів. Низхідний підхід ілюструється за допомогою концепцій моделі «сутність-зв'язок» (ER), починаючи з ідентифікації сутностей і зв'язків між сутностями, які становлять інтерес для організації.

Існують й інші підходи до проектування баз даних, такі як підхід «зсередини назовні» та підхід змішаної стратегії. Підхід «зсередини назовні» пов'язаний з підходом «знизу-угору», але відрізняється тим, що спочатку визначається набір основних об'єктів, а потім розглядаються інші об'єкти, зв'язки та атрибути, пов'язані з першими визначеними об'єктами. Змішаний стратегічний підхід використовує як висхідний, так і низхідний підхід для різних частин моделі, перш ніж нарешті об'єднати всі частини разом.

Моделювання даних

Дві основні цілі моделювання даних – допомогти зрозуміти зміст (семантику) даних і полегшити комунікацію щодо інформаційних потреб. Побудова моделі даних вимагає відповідей на питання про сутності, зв'язки та атрибути. Таким чином, розробники виявляють семантику даних, яка існує незалежно від того, записані вони у формальній моделі

даних чи ні. Сутності, зв'язки та атрибути є фундаментальними поняттями для баз даних. Однак їхнє значення може залишатися незрозумілим доти, доки вони не будуть правильно задокументовані. Модель даних сприяє повнішому розумінню змісту даних:

- точку зору кожного користувача на дані;
- природу самих даних, незалежно від їх фізичного представлення;
- використання даних з точки зору різних користувачів.

Моделі даних можуть бути використані для того, щоб передати розуміння розробником інформаційних потреб підприємства. За умови, що обидві сторони знайомі з нотацією, яка використовується в моделі, вона буде підтримувати комунікацію між користувачами і дизайнерами. Підприємства все частіше стандартизують спосіб моделювання даних, обираючи певний підхід до моделювання даних і використовуючи його у своїх проєктах з розробки баз даних. Найпопулярніша високорівнева модель даних, що використовується під час розробки баз даних, базується на концепціях ER-моделі.

Оптимальна модель даних інформаційної системи, яка розробляється, повинна задовольняти наступним критеріям [1].

Структурна обґрунтованість – це узгодженість з тим, як підприємство визначає та організовує інформацію.

Простота – це легкість розуміння фахівцями з ІС та нетехнічними користувачами.

Виразність – це здатність розрізняти різні дані, взаємозв'язки між даними та обмеження.

Ненадлишковість – це виключення зайвої інформації; зокрема, представлення будь-якої частини інформації лише один раз.

Спільне використання – не є специфічним для якоїсь конкретної програми чи технології і, таким чином, може бути використаний багатьма.

Розширюваність – це здатність розвиватися для підтримки нових вимог з мінімальним впливом на існуючих користувачів.

Цілісність – це узгодженість зі способом використання та управління інформацією на підприємстві.

Діаграмне представлення – це здатність представляти модель за допомогою легко зрозумілої діаграмної нотації.

Однак іноді ці критерії несумісні один з одним, і тоді доводиться йти на компроміси. Наприклад, намагаючись досягти більшої виразності моделі даних, ми можемо втратити її простоту.

Етапи проєктування бази даних

Проєктування бази даних складається з трьох основних етапів: концептуального, логічного та фізичного проєктування.

Концептуальне проєктування бази даних – це процес побудови моделі бази даних, що розробляється, незалежно від усіх фізичних міркувань.

Перший етап проєктування бази даних називається *концептуальним проєктуванням бази даних* і передбачає створення концептуальної моделі даних тієї частини підприємства, яка нас цікавить. Модель даних будується з використанням інформації, задокументованої у специфікації вимог користувачів. Концептуальне проєктування бази даних не залежить від деталей реалізації, таких як програмне забезпечення цільової СУБД, прикладні програми, мови програмування, апаратна платформа або будь-які інші фізичні міркування.

У процесі розробки концептуальної моделі даних модель тестується і перевіряється на відповідність вимогам користувачів. Концептуальна модель даних підприємства є джерелом інформації для наступного етапу, а саме логічного проєктування бази даних.

Логічне проєктування бази даних – це процес побудови моделі даних, що використовуються на підприємстві, на основі певної концептуальної моделі даних, але незалежно від конкретної СУБД та інших фізичних міркувань.

Другий етап проєктування бази даних називається *логічним проєктуванням бази даних*, у результаті якого створюється логічна модель даних тієї частини підприємства, яка нас цікавить. Концептуальна модель даних, створена на попередньому етапі, уточнюється і відображається на логічну модель даних. Логічна модель даних базується на цільовій моделі даних для бази даних (наприклад, реляційній моделі даних).

У той час як концептуальна модель даних не залежить від усіх фізичних міркувань, логічна модель виводиться, знаючи базову модель даних цільової СУБД. Іншими словами, ми знаємо, що СУБД є, наприклад, реляційною, мережевою, ієрархічною або об'єктно-орієнтованою. Однак ми ігноруємо будь-які інші аспекти обраної СУБД і, зокрема, будь-які фізичні деталі, такі як структури зберігання або індекси.

У процесі розробки логічної моделі даних модель тестується і перевіряється на відповідність вимогам користувачів. Для перевірки коректності логічної моделі даних використовується метод нормалізації. Нормалізація гарантує, що відношення, отримані з моделі даних, не відображають надлишковість даних, яка може спричинити аномалії оновлення під час реалізації. Логічна модель даних також повинна бути перевірена, щоб переконатися, що вона підтримує транзакції, визначені користувачами.

Логічна модель даних є джерелом інформації для наступного етапу, а саме фізичного проєктування бази даних, надаючи розробнику фізичної бази даних інструмент для досягнення компромісів, які є дуже важливими для ефективного проєктування бази даних. Логічна модель також відіграє

важливу роль на етапі оперативної підтримки життєвого циклу розробки системи баз даних. Належним чином підтримувана й оновлювана модель даних дозволяє точно та ефективно відображати в базі даних майбутні зміни в прикладних програмах або даних.

Проектування фізичної моделі бази даних – це процес створення опису реалізації бази даних на вторинному сховищі; він описує базові відносини, організації файлів та індекси, що використовуються для досягнення ефективного доступу до даних, а також будь-які пов'язані з ними обмеження цілісності та заходи безпеки.

Фізичне проектування бази даних – це третій і останній етап процесу проектування бази даних, під час якого проєктувальник вирішує, як буде реалізована база даних. На попередньому етапі проектування бази даних розробляється логічна структура бази даних, яка описує зв'язки та обмеження підприємства. Хоча ця структура не залежить від СУБД, вона розробляється відповідно до певної моделі даних, наприклад, реляційної, мережевої або ієрархічної. Однак, розробляючи фізичний дизайн бази даних, ми повинні спочатку визначити цільову СУБД. Тому фізичний дизайн адаптується до конкретної системи СУБД. Між фізичним і логічним проектуванням існує зворотній зв'язок, оскільки під час фізичного проектування приймаються рішення щодо підвищення продуктивності, які можуть вплинути на структуру логічної моделі даних.

Загалом, основною метою фізичного проектування бази даних є опис того, як ми маємо намір фізично реалізувати логічний проєкт бази даних. Для реляційної моделі це включає в себе:

- створення набору реляційних таблиць та обмежень на ці таблиці з інформації, представленої в логічній моделі даних;

- визначення конкретних структур зберігання та методів доступу до даних для досягнення оптимальної продуктивності системи баз даних;
- розробка захисту системи від несанкціонованого доступу до даних.

Проектування бази даних – це ітеративний процес, який має відправну точку і майже нескінченну низку вдосконалень. Його слід розглядати як процес навчання. У міру того, як розробники починають розуміти роботу підприємства та значення його даних, і виражають це розуміння у вибраних моделях даних, отримана інформація може вимагати змін в інших частинах проекту. Зокрема, концептуальний і логічний дизайн бази даних має вирішальне значення для загального успіху системи. Якщо проекти не відображають дійсного стану справ на підприємстві, буде важко, якщо не неможливо, визначити всі необхідні подання користувачів або підтримувати цілісність бази даних. Може навіть виявитися складно визначити фізичну реалізацію або підтримувати прийнятну продуктивність системи. З іншого боку, здатність пристосовуватися до змін є однією з ознак хорошого дизайну бази даних. Тому варто витратити час та енергію, необхідні для створення найкращого з можливих проєктів.

Вибір СУБД

Вибір СУБД – це вибір відповідної СУБД для підтримки системи баз даних.

Якщо СУБД не існує, то відповідна частина життєвого циклу, на якій можна зробити вибір, знаходиться між фазами концептуального та логічного проектування бази даних. Однак вибір можна зробити в будь-який час до логічного проектування за умови наявності достатньої інформації про системні вимоги, такі як продуктивність, легкість реструктуризації, безпека та обмеження щодо цілісності.

Хоча вибір СУБД може бути нечастим, у міру розширення потреб підприємства або заміни існуючих систем може виникнути потреба оцінити нові продукти СУБД. У таких випадках метою є вибір системи, яка відповідає поточним і майбутнім потребам підприємства, збалансований з витратами, що включають у себе придбання продукту СУБД, будь-якого додаткового програмного / апаратного забезпечення, необхідного для підтримки системи баз даних, а також витрати, пов'язані з переходом на іншу систему і навчанням персоналу.

Простий підхід до вибору полягає у зіставленні можливостей СУБД з вимогами. Під час вибору нового продукту СУБД є можливість переконатися, що процес вибору добре спланований, а система принесе реальну користь підприємству. У наступному розділі ми опишемо типовий підхід до вибору СУБД.

Процес вибору СУБД

Основні кроки вибору СУБД:

- визначити технічне завдання дослідження;
- скласти список з двох-трьох продуктів;
- оцінити продукти;
- рекомендувати вибір та підготувати звіт.

Технічне завдання на вибір СУБД складається із зазначенням цілей та обсягу дослідження, а також завдань, які необхідно виконати. Цей документ може також включати в себе опис критеріїв (на основі специфікації вимог користувачів), які будуть використовуватися для оцінки продуктів СУБД, попередній список можливих продуктів, а також усі необхідні обмеження та часові рамки для проведення дослідження.

Критерії, які вважаються «критичними» для успішного впровадження, можуть бути використані для створення попереднього списку продуктів СУБД для оцінки. Наприклад,

рішення про включення продукту СУБД може залежати від наявного бюджету, рівня підтримки постачальника, сумісності з іншим програмним забезпеченням, а також від того, чи працює продукт на певному апаратному забезпеченні. Додаткову корисну інформацію про продукт можна зібрати, зв'язавшись з існуючими користувачами, які можуть надати конкретні відомості про те, наскільки якісною є підтримка постачальника, як продукт підтримує певні додатки і чи є певні апаратні платформи більш проблемними, ніж інші. Після початкового вивчення функціональних можливостей і особливостей продуктів СУБД визначається короткий список з двох або трьох продуктів.

Існують різні характеристики, які можна використовувати для оцінки СУБД. Для цілей оцінювання ці характеристики можна оцінювати як групи (наприклад, визначення даних) або окремо (наприклад, доступні типи даних). Далі наведено перелік можливих характеристик для оцінювання продукту СУБД, згрупованих за визначенням даних, фізичним визначенням, доступністю, обробкою транзакцій, утилітами, розробкою та іншими характеристиками.

1. Визначення даних:

- обмеження первинного ключа;
- специфікація зовнішнього ключа;
- доступні типи даних;
- розширюваність типів даних;
- специфікація домену;
- простота реструктуризації;
- контроль цілісності;
- механізм представлення;
- словник даних;
- незалежність даних;
- базова модель даних;
- еволюція схем.

2. Фізичні характеристики:
 - доступність файлових структур;
 - підтримка файлових структур;
 - простота реорганізації;
 - індексування;
 - поля / записи змінної довжини;
 - стискання даних;
 - механізми шифрування;
 - вимоги до пам'яті;
 - вимоги до сховища.
3. Доступність:
 - мова запитів: сумісність зі стандартами;
 - сумісність із сучасними мовами програмування;
 - багатокористувацька;
 - безпека (контроль доступу, механізм авторизації).
4. Управління транзакціями:
 - процедури резервного копіювання і відновлення та механізм установлення контрольних точок;
 - механізм ведення журналів;
 - деталізація паралелізму;
 - стратегія розв'язання тупикових ситуацій;
 - розширені моделі транзакцій;
 - паралельна обробка запитів.
5. Утиліти:
 - вимірювання продуктивності;
 - налаштування;
 - механізми завантаження / вивантаження;
 - моніторинг використання користувачами;
 - підтримка адміністрування баз даних.
6. Розробка:
 - CASE-засоби;
 - збережені процедури, тригери та правила;
 - інструменти веброботки.

7. Інші особливості:

- можливість модернізації;
- стабільність постачальника;
- спільнота користувачів;
- утиліти для реплікації бази даних;
- навчальні програми для користувачів та їх підтримки;
- механізми взаємодії з іншими СУБД та іншими

системами;

- інтеграція з інтернетом;
- утиліти для роботи з базами даних;
- розподілені можливості;
- документація;
- необхідна операційна система;
- вартість;
- онлайн-допомога;
- використані стандарти;
- управління версіями;
- розширювана оптимізація запитів;
- масштабованість;
- підтримка звітності та аналітичних інструментів;
- переносимість;
- необхідне обладнання;
- мережева підтримка;
- об'єктно-орієнтовані можливості;
- архітектура (2- або 3-рівневий клієнт / сервер);
- продуктивність;
- пропускну здатність транзакцій;
- максимальна кількість одночасних користувачів;
- підтримка XML та вебсервісів.

Якщо просто відмічати характеристики із зазначенням того, наскільки кожна з них хороша чи погана, може бути складно порівнювати продукти СУБД між собою. Більш корисним підходом є зважування функцій та/або груп

функцій відповідно до їхньої важливості для організації та отримання загальної зваженої експертної оцінки, яка може використовувати для порівняння продуктів.

Заключним етапом вибору СУБД є документування процесу та надання звіту з висновками та рекомендаціями щодо конкретного продукту СУБД.

Проектування застосунків

Проектування застосунків – це проектування інтерфейсу користувача і прикладних програм, які використовують та обробляють базу даних.

Проектування бази даних і прикладних програм є паралельними етапами життєвого циклу розробки системи баз даних. У більшості випадків неможливо завершити проектування застосунку, поки не буде розроблено саму базу даних. З іншого боку, база даних існує для підтримки застосунків, тому між проектуванням застосунків і проектуванням баз даних повинен існувати обмін інформацією.

Ми повинні переконатися, що вся функціональність, зазначена у специфікації вимог користувачів, присутня в проєкті програми для системи баз даних. Це передбачає розробку прикладних програм, які отримують доступ до бази даних, і розробку транзакцій (тобто методів доступу до бази даних). На додаток до проектування того, як буде досягнута необхідна функціональність, ми повинні розробити відповідний користувацький інтерфейс до системи баз даних. Цей інтерфейс повинен представляти необхідну інформацію у зручний для користувача спосіб. Важливість дизайну користувацького інтерфейсу іноді ігнорується або залишається на пізніх етапах проектування. Однак слід визнати, що інтерфейс може бути одним з найважливіших компонентів системи. Якщо він легкий у вивченні, простий у використанні, зрозумілий і пробачливий, користувачі

будуть схильні до ефективного використання представленої інформації. З іншого боку, якщо інтерфейс не має жодної з цих характеристик, система буде викликати проблеми.

Коротко розглянемо два аспекти дизайну додатків:

- дизайн транзакцій;
- дизайн інтерфейсу користувача.

Дизайн транзакцій

Транзакція – це дія або серія дій, що виконуються одним користувачем або прикладною програмою, яка отримує доступ до вмісту бази даних або змінює його.

Транзакції представляють «реальні» події, такі як реєстрація нерухомості в оренду, прийняття нового співробітника, реєстрація нового клієнта, здача нерухомості в оренду. Ці транзакції мають бути внесені до бази даних, щоб гарантувати, що дані, які зберігаються в базі даних, залишаються актуальними у «реальному світі», а також для підтримки інформаційних потреб користувачів.

Транзакція може складатися з декількох операцій, наприклад, переказ грошей з одного рахунку на інший. Однак, з точки зору користувача, ці операції все одно виконують одне завдання. З точки зору СУБД, транзакція переводить базу даних з одного узгодженого стану в інший. СУБД забезпечує цілісність бази даних навіть у випадку збою. СУБД також гарантує, що після завершення транзакції внесені зміни назавжди зберігаються в базі даних і не можуть бути втрачені або скасовані (без запуску іншої транзакції для компенсації наслідків першої транзакції). Якщо транзакція не може завершитися з будь-якої причини, СУБД повинна забезпечити скасування змін, внесених цією транзакцією. У прикладі банківського переказу, якщо гроші списуються з одного рахунку, а транзакція не завершується до зарахування на інший рахунок, СУБД повинна відмінити дебет.

Якщо ми визначимо операції дебетування і кредитування як окремі транзакції, то після того, як ми дебетуємо перший рахунок і завершимо транзакцію, ми не зможемо скасувати цю зміну (без виконання іншої транзакції для кредитування дебетованого рахунку на необхідну суму).

Метою проєктування транзакцій є визначення та документування високорівневих характеристик транзакцій, необхідних для бази даних, включаючи:

- дані, які будуть використовуватися транзакцією;
- функціональні характеристики транзакції;
- результат транзакції;
- важливість для користувачів;
- очікувана частота використання.

Цю роботу слід проводити на ранній стадії проєктування, щоб переконатися, що реалізована база даних здатна підтримувати всі необхідні транзакції. Існує три основні типи транзакцій: транзакції вилучення даних, транзакції оновлення даних та змішані транзакції.

Транзакції вилучення даних використовуються для отримання даних для відображення на екрані або у створенні звіту. Наприклад, операція пошуку та відображення деталей об'єкта (за номером об'єкта) є прикладом транзакції вилучення.

Транзакції оновлення використовуються для вставки нових записів, видалення старих записів або модифікації існуючих записів у базі даних. Наприклад, операція вставки даних про новий об'єкт нерухомості в базу даних є прикладом транзакції оновлення.

Змішані транзакції передбачають як пошук, так і оновлення даних. Наприклад, операція пошуку та відображення інформації про об'єкт нерухомості (за номером об'єкта), а потім оновлення значення щомісячної орендної плати є прикладом змішаної транзакції.

Принципи проєктування користувацького інтерфейсу

Перш ніж упроваджувати форму або звіт, важливо спочатку розробити макет. Корисні рекомендації, яких слід дотримуватися під час розробки форм або звітів, наведені у роботі [2].

Змістовий заголовок.

Інформація, що міститься в заголовку, повинна чітко й однозначно визначати мету форми / звіту.

Зрозумілі інструкції.

Для донесення інструкцій до користувача слід використовувати знайому йому термінологію. Інструкції мають бути короткими, а якщо потрібна додаткова інформація, слід передбачити довідкові екрани. Інструкції повинні бути написані в єдиному граматичному стилі, з використанням стандартного формату.

Логічне групування та послідовність полів.

Пов'язані поля повинні бути розташовані разом у формі / звіті. Послідовність полів має бути логічною та послідовною.

Візуально привабливий макет форми / звіту.

Форма / звіт повинна мати привабливий інтерфейс для користувача. Форма / звіт має виглядати збалансовано, з полями або групами полів, рівномірно розташованими по всій формі / звіту. У формі / звіті не повинно бути областей, які мають занадто мало або занадто багато полів. Поля або групи полів повинні бути відокремлені однаковою кількістю пробілів. Там, де це доречно, поля повинні бути вирівняні по вертикалі або горизонталі. У випадках, коли форма на екрані має друкований еквівалент, зовнішній вигляд обох форм має бути однаковим.

Звичні назви полів.

Назви полів мають бути звичними. Наприклад, якщо поле «Score» замінити на «Grade», деякі користувачі можуть заплутатися.

Узгодженість термінології та скорочень.

Узгоджений перелік звичних термінів та скорочень повинен використовуватися послідовно.

Послідовне використання кольору.

Колір слід використовувати для покращення зовнішнього вигляду форми / звіту та для виділення важливих полів або важливих повідомлень. Щоб досягти цього, колір повинен використовуватися послідовно і змістовно. Наприклад, поля на формі з білим фоном можуть указувати на поля для введення даних, а поля із синім фоном можуть указувати на поля, призначені лише для відображення.

Видимий простір і межі полів для введення даних.

Користувач повинен візуально бачити загальну кількість місця, доступного для кожного поля. Це дозволяє користувачеві обдумати відповідний формат даних перед тим, як вводити значення в поле.

Зручне переміщення курсору.

Користувач повинен легко визначити операцію, необхідну для переміщення курсору по формі / звіту. Слід використовувати прості механізми, такі як клавіша Tab, стрілки або вказівник миші.

Виправлення помилок для окремих символів і цілих полів.

Користувач повинен легко визначити операцію, необхідну для внесення змін до значень полів. Повинні бути доступні прості механізми, такі як використання клавіші Backspace або перенабір.

Повідомлення про помилки для неприпустимих значень.

Якщо користувач намагається ввести неправильні дані в поле, має з'явитися повідомлення про помилку. Повідомлення повинно інформувати користувача про помилку та вказувати на допустимі значення.

Чітке позначення необов'язкових полів.

Необов'язкові поля повинні бути чітко ідентифіковані для користувача. Цього можна досягти за допомогою відповідної мітки поля або шляхом відображення поля кольором,

який указує на тип поля. Необов'язкові поля слід розміщувати після обов'язкових полів.

Пояснювальні повідомлення для полів.

Коли користувач наводить курсор на поле, інформація про поле має з'являтися у звичайному місці на екрані, наприклад, у рядку стану вікна.

Сигнал завершення.

Користувачеві має бути зрозуміло, коли процес заповнення полів форми завершено. Однак можливість завершити процес не повинна бути автоматичною, оскільки користувач може захотіти переглянути введені дані.

Реалізація

Реалізація проєкту – це фізична реалізація бази даних та додатку.

Після завершення етапів проєктування (які могли включати або не включати в себе створення прототипів), ми можемо реалізувати базу даних та прикладні програми. Реалізація бази даних досягається за допомогою DDL обраної СУБД або графічного інтерфейсу, який забезпечує ту ж функціональність, але приховує низькорівневі оператори DDL. Оператори DDL використовуються для створення структур бази даних та порожніх файлів бази даних. На цьому етапі також реалізуються будь-які задані користувацькі представлення.

Прикладні програми створюються з використанням сучасних мов програмування. Це включає в себе транзакції бази даних, які реалізуються за допомогою DML у цільовій СУБД, інтегрованої в мови програмування, такі як Python, Java, Kotlin, Delphi, C++, C# тощо. Ми також створюємо меню, форми введення даних і звіти. Цільова СУБД може мати інструменти інтеграції із сучасними мовами програмування для швидкої розробки застосунків із

об'єктно-орієнтованими та функціональними мовами програмування для створення запитів і використання генераторів форм і звітів.

Також реалізовано засоби контролю безпеки та цілісності системи. Деякі із цих засобів контролю реалізовано за допомогою DDL, але інші, можливо, доведеться визначати поза DDL, використовуючи, наприклад, утиліти СУБД або засоби операційної системи, що постачаються у комплекті.

Перетворення та завантаження даних

Перетворення і завантаження даних – це перенесення всіх наявних даних до нової бази даних і перетворення всіх наявних додатків для роботи в новій базі даних.

Цей етап необхідний лише тоді, коли нова система баз даних замінює стару систему. На сьогодні СУБД зазвичай має утиліту, яка завантажує існуючі файли в нову базу даних. Ця утиліта зазвичай вимагає вказати вихідний файл і цільову базу даних, а потім автоматично перетворює дані у необхідний формат файлів нової бази даних. Там, де це можливо, розробник може конвертувати і використовувати прикладні програми зі старої системи для використання в новій системі. У разі потреби конвертації та завантаження, процес повинен бути належним чином спланований, щоб забезпечити плавний перехід до повноцінної роботи.

Тестування

Тестування – це процес запуску системи баз даних з метою пошуку помилок.

Перед запуском в експлуатацію новостворена система баз даних повинна бути ретельно протестована. Це досягається за допомогою ретельно спланованих тестових стратегій і реалістичних даних. Зверніть увагу, що в нашому визначенні тестування ми не використовували загальноприйняту

думку, що тестування – це процес демонстрації відсутності помилок. Насправді, тестування не може показати відсутність помилок; воно може показати лише те, що помилки в програмному забезпеченні присутні. Якщо тестування проведено успішно, воно виявить помилки в прикладних програмах і, можливо, у структурі бази даних. Як вторинна перевага, тестування демонструє, що база даних і прикладні програми працюють відповідно до своїх специфікацій і що вимоги до продуктивності задовольняються. Крім того, метрики, зібрані на етапі тестування, дають змогу оцінити надійність та якість програмного забезпечення.

Як і у випадку з розробкою бази даних, користувачі нової системи повинні бути залучені до процесу тестування. Ідеальною ситуацією для тестування системи є наявність тестової бази даних на окремій апаратній системі, але часто це неможливо. Якщо будуть використовуватися реальні дані, важливо мати резервні копії на випадок помилок.

Тестування також має охоплювати зручність використання системи баз даних. В ідеалі, оцінка повинна проводитися відповідно до специфікації юзабіліті. Приклади критеріїв, які можуть бути використані для проведення оцінки, включають у себе наступні [3]:

- здатність до навчання: скільки часу потрібно новому користувачеві, щоб почати продуктивно працювати із системою?
- продуктивність: наскільки добре реакція системи відповідає робочій практиці користувача?
- надійність: наскільки толерантною є система до помилок користувача?
- відновлюваність: наскільки добре система відновлюється після помилок користувача?
- адаптивність: наскільки тісно система прив'язана до єдиної моделі роботи?

Деякі із цих критеріїв можуть бути оцінені на інших етапах життєвого циклу. Після завершення тестування система баз даних готова до «підписання» і передачі користувачам.

Експлуатаційне обслуговування

Експлуатаційне обслуговування – це процес моніторингу та підтримки системи баз даних після встановлення.

На попередніх етапах система баз даних була повністю впроваджена та протестована. Зараз система переходить у стадію обслуговування, яка передбачає наступні види діяльності:

- моніторинг продуктивності системи. Якщо продуктивність падає нижче прийнятного рівня, може знадобитися налаштування або реорганізація бази даних;

- обслуговування та модернізація системи баз даних (за потреби). Нові вимоги вносяться до системи баз даних на попередніх етапах життєвого циклу.

Після того, як система баз даних повністю введена в експлуатацію, здійснюється ретельний моніторинг, щоб гарантувати, що продуктивність залишається на прийнятному рівні. СУБД зазвичай надає різні утиліти для адміністрування бази даних, включаючи утиліти для завантаження даних до бази даних і для моніторингу системи. Утиліти, які дозволяють здійснювати моніторинг системи, надають інформацію, наприклад, про використання бази даних, ефективність блокування (включаючи кількість тупикових ситуацій, що виникли, і так далі) та стратегію виконання запитів. DBA може використовувати цю інформацію, щоб налаштувати систему на кращу продуктивність, наприклад, створюючи додаткові індекси для прискорення запитів, змінюючи структури зберігання, об'єднуючи або розбиваючи таблиці.

Процес моніторингу триває протягом усього життя системи баз даних і з часом може призвести до реорганізації

бази даних, щоб задовольнити мінливі вимоги. Ці зміни зі свого боку надають інформацію про ймовірний розвиток системи та майбутні ресурси, які можуть знадобитися. Це, разом зі знанням пропонованих нових додатків, дає змогу DBA брати участь у плануванні потужностей і повідомляти або попереджати старший персонал, щоб він відповідно скоригував плани. Якщо СУБД бракує певних утиліт, DBA може або розробити необхідні утиліти власними силами, або придбати додаткові інструменти постачальника, якщо вони доступні [4].

Коли нова система баз даних вводиться в експлуатацію, користувачі повинні деякий час працювати з нею паралельно зі старою системою. Такий підхід захищає поточні операції на випадок непередбачуваних проблем з новою системою. Необхідно періодично перевіряти узгодженість даних між двома системами, і лише тоді, коли обидві системи видають однакові результати, слід відмовитися від старої системи. Якщо перехід буде занадто поспішним, кінцевий результат може бути катастрофічним. Незважаючи на зазначене вище припущення, що від старої системи можна відмовитися, можливі ситуації, у яких обидві системи зберігаються.

2.2. Приклад проєкту інформаційної системи

Особливо важливим на ранніх етапах життєвого циклу, включаючи етапи планування бази даних, опису предметної області, збору та аналізу вимог, є пошук фактів. Саме на цих ранніх етапах розробник бази даних фіксує основні факти, необхідні для створення необхідної бази даних. Збір фактів також використовується під час проєктування бази даних і на пізніших етапах життєвого циклу, але меншою мірою. Наприклад, під час фізичного проєктування бази даних пошук фактів стає технічним, оскільки розробник бази

даних намагається дізнатися більше про СУБД, обрану для системи баз даних. Крім того, на завершальному етапі, оперативного обслуговування, пошук фактів використовується для визначення того, чи потребує система налаштування для підвищення продуктивності або подальшого розвитку з урахуванням нових вимог.

Розробник бази даних зазвичай використовує кілька методів пошуку фактів під час проєкту з однією базою даних. Існує п'ять найпоширеніших методів пошуку фактів:

- вивчення документації;
- інтерв'ювання;
- спостереження за роботою підприємства;
- дослідження;
- анкетування.

Представимо застосування методів пошуку фактів і створення документації на ранніх етапах життєвого циклу розробки інформаційної системи, а саме на етапах планування бази даних, опису предметної області, а також збору та аналізу вимог на прикладі IC DreamUniversity.

Предметна область:

Інформаційна система деканату університету

Зібрані дані аналізуються, щоб визначити обсяг і межі інформаційної системи та її основних представлень користувачів. Представлення кожного користувача описує вимоги, які повинна підтримувати інформаційна система, визначені певною посадовою роллю. Межі IC DreamUniversity показані на рис. 2.1.

Далі ми аналізуємо зібрані дані, щоб визначити основні погляди користувачів. Більшість даних про користувачів було зібрано під час інтерв'ю з викладачами та студентами в ролі співробітника деканату, завідувача кафедри, куратора групи, викладача предмету та студента.

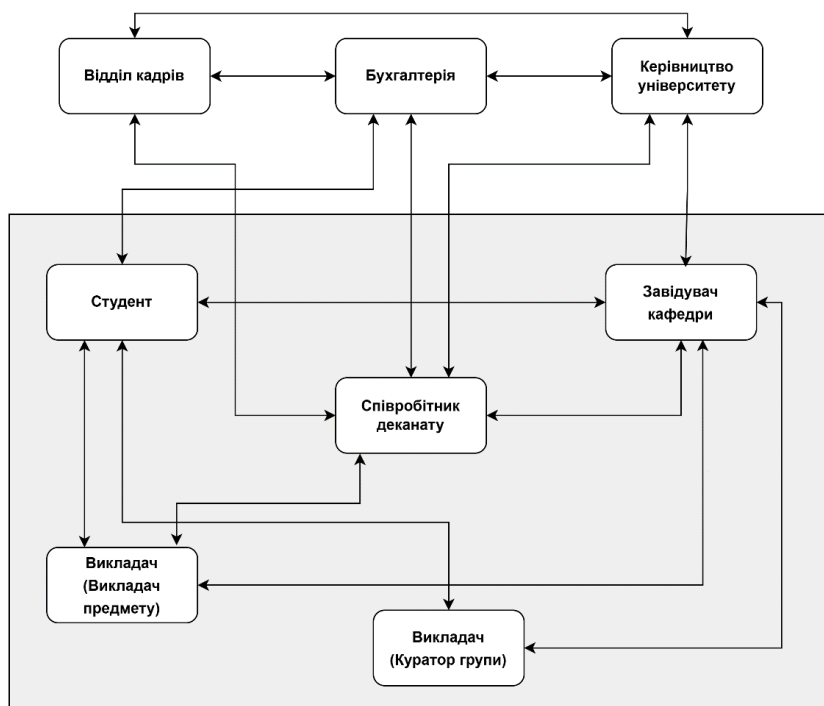


Рис. 2.1. Межі інформаційної системи DreamUniversity

Предметна область охоплює процеси та дані, пов'язані з діяльністю деканату університету, спрямованої на управління процесом обліку контингенту на рівні факультету або інституту. Інформаційна система, що розробляється, має на меті автоматизувати ключові функції деканату, забезпечити ефективне зберігання, обробку та доступ до інформації, необхідної для студентів, викладачів та адміністративного персоналу.

Основні цілі інформаційної системи:

- Оптимізація документообігу: перехід від паперових документів до електронного формату для прискорення та спрощення процесів.

- Покращення обліку студентів: централізоване зберігання та актуалізація інформації про студентів.
- Ефективне управління академічними групами: ведення обліку груп, їх складу та кураторів.
- Генерація звітів: надання необхідних статистичних даних та звітів для аналізу та прийняття рішень.
- Покращення комунікації: забезпечення ефективного обміну інформацією між студентами, викладачами та адміністрацією.

Основні процеси, які має підтримувати інформаційна система:

- Ведення обліку студентів: реєстрація, оновлення особистих та навчальних даних студентів.
- Формування та управління академічними групами: створення груп, призначення кураторів, відстеження змін у складі.
- Облік успішності студентів: внесення оцінок, формування відомостей, розрахунків середнього бала, підготовка документів про успішність.
- Оформлення студентських документів: замовлення та видача довідок.
- Ведення архіву документів: зберігання та забезпечення доступу до електронних копій документів.
- Формування звітів: генерація статистичних звітів про контингент студентів, успішність, завантаженість викладачів тощо.
- Комунікація зі студентами та викладачами: надання доступу до необхідної інформації через особисті кабінети (за потреби).

Основні користувачі інформаційної системи:

- Співробітники деканату (декан, заступники декана, секретарі).
- Викладачі (завідувачі кафедр, куратори груп, викладачі дисциплін).

– Студенти (для перегляду власної інформації, успішності).

Розробка інформаційної системи для деканату університету вимагає детального аналізу потреб користувачів (*user stories*), проєктування зручного інтерфейсу та надійної бази даних для забезпечення ефективної роботи деканату та підтримки якісного навчального процесу.

На цьому етапі ми продовжуємо збирати більш детальну інформацію про погляди користувачів, визначені на попередньому етапі, щоб створити специфікацію вимог користувачів, яка детально описує дані, що зберігатимуться в базі даних, і те, як ці дані використовуватимуться. Збираючи більше інформації про бачення користувачів, ми також збираємо будь-які загальні вимоги до системи. Метою збору цієї інформації є створення системної специфікації, яка описує всі функції, що мають бути включені в нову систему баз даних, такі як вимоги до мережі та спільного доступу, вимоги до продуктивності та необхідний рівень безпеки.

1. Співробітник деканату:

Управління кафедрами:

– Як співробітник деканату, я хочу мати можливість **переглядати повний список кафедр** з інформацією про їх назву та завідувача, щоб мати загальне уявлення про структуру університету.

– Як співробітник деканату, я хочу мати можливість **фільтрувати та сортувати список кафедр** за назвою або завідувачем, щоб швидко знаходити потрібну інформацію.

– Як співробітник деканату, я хочу мати можливість **додавати нову кафедру**, указуючи її назву та, за бажанням, одразу призначати завідувача з існуючих викладачів.

– Як співробітник деканату, я хочу мати можливість **редагувати назву кафедри** у випадку її перейменування.

– Як співробітник деканату, я хочу мати можливість **призначати існуючого викладача завідувачем кафедри**, щоб формалізувати керівництво.

– Як співробітник деканату, я хочу мати можливість **змінювати завідувача кафедри**, якщо відбуваються кадрові зміни.

– Як співробітник деканату, я хочу мати можливість **переглядати історію змін завідувачів кафедри** (якщо така функціональність буде реалізована), щоб відстежувати адміністративні процеси.

– Як співробітник деканату, я хочу мати можливість **видаляти кафедру** (з попередженням про можливі наслідки для пов'язаних груп та викладачів), якщо кафедра розформовується.

Управління спеціальностями:

– Як співробітник деканату, я хочу мати можливість **переглядати повний список спеціальностей** з їх назвами та кодами.

– Як співробітник деканату, я хочу мати можливість **фільтрувати та сортувати список спеціальностей** за назвою або кодом.

– Як співробітник деканату, я хочу мати можливість **додавати нову спеціальність**, указуючи її повну назву та унікальний код.

– Як співробітник деканату, я хочу мати можливість **редагувати назву та код спеціальності**.

– Як співробітник деканату, я хочу мати можливість **переглядати всі академічні групи**, які навчаються за певною спеціальністю.

– Як співробітник деканату, я хочу мати можливість **видаляти спеціальність** (з попередженням про можливі наслідки для пов'язаних груп), якщо вона більше не актуальна.

Управління посадами:

- Як співробітник деканату, я хочу мати можливість **переглядати список усіх існуючих посад.**
- Як співробітник деканату, я хочу мати можливість **додавати нову посаду**, указуючи її назву.
- Як співробітник деканату, я хочу мати можливість **редагувати назву посади.**
- Як співробітник деканату, я хочу мати можливість **переглядати всіх викладачів**, які займають певну посаду.
- Як співробітник деканату, я хочу мати можливість **видаляти посаду** (з попередженням про можливі наслідки для пов'язаних викладачів), якщо вона більше не використовується.

Управління викладачами:

- Як співробітник деканату, я хочу мати можливість **переглядати повний список викладачів** з їхніми іменами, прізвищами, посадами та кафедрами.
- Як співробітник деканату, я хочу мати можливість **фільтрувати та сортувати список викладачів** за прізвищем, кафедрою або посадою.
- Як співробітник деканату, я хочу мати можливість **додавати нового викладача**, указуючи його ім'я, прізвище, науковий ступінь, посаду та кафедру.
- Як співробітник деканату, я хочу мати можливість **редагувати всю інформацію про викладача.**
- Як співробітник деканату, я хочу мати можливість **призначати викладача завідувачем кафедри.**
- Як співробітник деканату, я хочу мати можливість **переглядати список усіх академічних груп**, куратором яких є певний викладач.
- Як співробітник деканату, я хочу мати можливість **переглядати список усіх навчальних предметів**, які викладає певний викладач.

– Як співробітник деканату, я хочу мати можливість **видаляти викладача** (з попередженням про можливі наслідки для пов'язаних груп та предметів).

Управління академічними групами:

– Як співробітник деканату, я хочу мати можливість **переглядати повний список академічних груп** з їхніми назвами, кураторами, спеціальностями та кафедрами.

– Як співробітник деканату, я хочу мати можливість **фільтрувати та сортувати список груп** за назвою, куратором, спеціальністю або кафедрою.

– Як співробітник деканату, я хочу мати можливість **додавати нову академічну групу**, указуючи її назву, призначаючи куратора, указуючи спеціальність та кафедру.

– Як співробітник деканату, я хочу мати можливість **редагувати всю інформацію про академічну групу**.

– Як співробітник деканату, я хочу мати можливість **переглядати список усіх студентів** у певній академічній групі.

– Як співробітник деканату, я хочу мати можливість **переводити студентів** між академічними групами.

– Як співробітник деканату, я хочу мати можливість **випускати академічну групу** (архівувати її).

Управління студентами:

– Як співробітник деканату, я хочу мати можливість **переглядати повний список студентів** з їхніми іменами, прізвищами та групами.

– Як співробітник деканату, я хочу мати можливість **фільтрувати та сортувати список студентів** за прізвищем, групою або містом.

– Як співробітник деканату, я хочу мати можливість **додавати нового студента**, указуючи всю необхідну інформацію.

– Як співробітник деканату, я хочу мати можливість **редагувати всю інформацію про студента**.

– Як співробітник деканату, я хочу мати можливість **переводити студента до іншої академічної групи**.

– Як співробітник деканату, я хочу мати можливість **виключати студента з університету (із зазначенням причини)**.

Звіти:

– Як співробітник деканату, я хочу мати можливість **генерувати звіт про кількість студентів** на кожній спеціальності.

– Як співробітник деканату, я хочу мати можливість **генерувати звіт про кількість викладачів** на кожній кафедрі.

– Як співробітник деканату, я хочу мати можливість **генерувати список усіх академічних груп** та їх кураторів.

2. Завідувач кафедри:

Управління викладачами кафедри:

– Як завідувач кафедри, я хочу мати можливість **переглядати повний список викладачів** моєї кафедри з їхніми посадами та науковими ступенями.

– Як завідувач кафедри, я хочу мати можливість **переглядати детальну інформацію про кожного викладача** моєї кафедри.

– Як завідувач кафедри (за наявності прав), я хочу мати можливість **редагувати інформацію про викладачів** моєї кафедри (наприклад, посаду).

Управління академічними групами кафедри:

– Як завідувач кафедри, я хочу мати можливість **переглядати список усіх академічних груп**, які належать до моєї кафедри, з інформацією про їх кураторів та спеціальності.

— Як завідувач кафедри, я хочу мати можливість **призначати / змінювати кураторів** для академічних груп моєї кафедри.

— Як завідувач кафедри, я хочу мати можливість **переглядати список студентів** у кожній академічній групі моєї кафедри.

Управління навчальними предметами кафедри (якщо ця функціональність буде додана):

— Як завідувач кафедри, я хочу мати можливість **переглядати перелік усіх навчальних предметів**, які викладаються на моїй кафедрі.

— Як завідувач кафедри, я хочу мати можливість **призначати викладачів** для ведення певних навчальних предметів.

Звіти кафедри:

— Як завідувач кафедри, я хочу мати можливість **генерувати звіт про кількість студентів** на спеціальностях моєї кафедри.

— Як завідувач кафедри, я хочу мати можливість **генерувати звіт про викладацький склад** моєї кафедри за посадами та науковими ступенями.

3. Викладач (Куратор групи):

Інформація про групу:

— Як куратор групи, я хочу мати можливість **переглядати назву та іншу основну інформацію** про мою академічну групу.

— Як куратор групи, я хочу мати можливість **переглядати список усіх студентів** у моїй групі з їхніми контактними даними (якщо дозволено).

— Як куратор групи, я хочу мати можливість **переглядати детальну інформацію про кожного студента** моєї групи, включаючи особисті дані, контактну інформацію та місце проживання.

Взаємодія зі студентами:

- Як куратор групи, я хочу мати можливість **зв'язуватися зі студентами** моєї групи індивідуально або груповим повідомленням (якщо буде інтегрована система комунікації).
- Як куратор групи, я хочу мати можливість **робити нотатки** щодо окремих студентів моєї групи (наприклад, щодо їх успішності чи проблем).
- Як куратор групи, я хочу мати можливість **відстежувати успішність** студентів моєї групи (якщо буде інтегрована система оцінювання).

4. Викладач (Викладач предмету):

Інформація про предмети та групи:

- Як викладач предмету, я хочу мати можливість **переглядати список усіх навчальних предметів**, які я викладаю.
- Як викладач предмету, я хочу мати можливість **переглядати список академічних груп**, яким я викладаю кожен свій предмет.
- Як викладач предмету, я хочу мати можливість **переглядати список студентів** у кожній академічній групі, які вивчають мій предмет.

Інформація про навчальні предмети:

- Як викладач предмету, я хочу мати можливість **переглядати детальну інформацію про кожен свій навчальний предмет** (назва, опис, тривалість, семестр).

Взаємодія зі студентами (можливо у майбутньому):

- Як викладач предмету, я хочу мати можливість **зв'язуватися зі студентами**, які вивчають мій предмет (індивідуально або груповим повідомленням).
- Як викладач предмету, я хочу мати можливість **переглядати успішність студентів** з мого предмету.

5. Студент:

Особиста інформація:

- Як студент, я хочу мати можливість **переглядати свій профіль** з усією своєю особистою інформацією.

– Як студент, я хочу мати можливість **редагувати свою контактну інформацію** (якщо це дозволено політикою університету).

Інформація про навчання:

– Як студент, я хочу мати можливість **переглядати назву своєї академічної групи** та інформацію про куратора (контактні дані, якщо дозволено).

– Як студент, я хочу мати можливість **переглядати список студентів у своїй групі**.

– Як студент, я хочу мати можливість **переглядати свій навчальний план** (якщо ця інформація буде додана до бази даних).

– Як студент, я хочу мати можливість **переглядати перелік навчальних предметів**, які я вивчаю в поточному семестрі.

– Як студент, я хочу мати можливість **переглядати інформацію про викладачів** своїх навчальних предметів (контактні дані, якщо дозволено).

– Як студент, я хочу мати можливість **переглядати свою успішність** (якщо буде інтегрована система оцінювання).

Ці деталізовані *user stories* допомагають краще зрозуміти потреби кожного типу користувачів та окреслити потенційну функціональність системи управління навчальним процесом. У процесі розробки ці історії можуть бути ще більше поділені на конкретніші завдання.

2.3. Вступ до методології проєктування бази даних

Методологія проєктування бази даних – це структурований підхід, який використовує процедури, методи, інструменти й засоби документації для підтримки та полегшення процесу проєктування.

Методологія проєктування складається з етапів, кожен з яких містить низку кроків, що спрямовують розробника

до методів, доречних на кожному етапі проєкту. Методологія проєктування також допомагає розробнику планувати, керувати, контролювати та оцінювати проєкти розробки баз даних. Крім того, це структурований підхід для аналізу та моделювання набору вимог до бази даних у стандартизований та організований спосіб.

Концептуальне, логічне та фізичне проєктування баз даних

Процес проєктування поділяється на три основні фази: концептуальне, логічне та фізичне проєктування баз даних.

Концептуальне проєктування бази даних – це процес побудови моделі даних, що використовуються на підприємстві, незалежно від усіх фізичних міркувань.

Етап проєктування концептуальної бази даних починається зі створення концептуальної моделі даних, яка повністю не залежить від деталей реалізації, таких як цільова СУБД, прикладні програми, мови програмування, апаратна платформа, проблеми продуктивності або будь-які інші фізичні міркування.

Логічне проєктування бази даних – це процес побудови моделі даних, що використовуються на підприємстві, на основі певної моделі даних, але незалежно від конкретної СУБД та інших фізичних міркувань.

На етапі логічного проєктування бази даних концептуальна модель даних перетворюється на логічну модель, на яку впливає модель даних цільової бази даних (наприклад, реляційна модель). Логічна модель даних є джерелом інформації для етапу фізичного проєктування, надаючи розробнику фізичної бази даних інструмент для досягнення компромісів, які дуже важливі для проєктування ефективної бази даних.

Фізичне проєктування баз даних – це процес створення опису реалізації бази даних на вторинному сховищі; він

описує відношення бази, організацію файлів та індекси, що використовуються для досягнення ефективного доступу до даних, а також будь-які пов'язані із цим обмеження цілісності та заходи безпеки.

Етап фізичного проектування бази даних дозволяє проектувальнику приймати рішення про те, як буде реалізована база даних. Тому фізичне проектування адаптується до конкретної СУБД. Між фізичним і логічним проектуванням існує зворотній зв'язок, оскільки рішення, прийняті під час фізичного проектування для підвищення продуктивності, можуть вплинути на логічну модель даних.

Крок за кроком методологія виглядає наступним чином:

Концептуальна модель бази даних

1. Побудова концептуальної моделі даних:
 - 1.1. Визначення сутностей.
 - 1.2. Визначення типів зв'язків.
 - 1.3. Визначення та асоціювання атрибутів із сутностями або зв'язками.
 - 1.4. Визначення доменів атрибутів.
 - 1.5. Визначення атрибутів-кандидатів, первинних та альтернативних ключів.
 - 1.6. Розгляд використання розширених концепцій моделювання (необов'язковий крок).
 - 1.7. Перевірка моделі на надлишковість.
 - 1.8. Перевірка концептуальної моделі на відповідність транзакціям користувачів.
 - 1.9. Перегляд концептуальної моделі даних з користувачем.

Проектування логічної моделі для реляційних баз даних

2. Побудова логічної моделі даних:
 - 2.1. Виведення відношень для логічної моделі даних.
 - 2.2. Перевірка відношень за допомогою нормалізації.

2.3. Перевірка відношень на основі транзакцій користувача.

2.4. Перевірка обмежень цілісності.

2.5. Перегляд логічної моделі даних з користувачем.

2.6. Об'єднання логічних моделей даних у глобальну модель (необов'язковий крок).

2.7. Перевірка на предмет майбутнього зростання.

Проектування фізичної моделі для реляційних баз даних

3. Трансляція логічної моделі даних для цільової СУБД:

3.1. Проектування основних відношень.

3.2. Проектування представлення даних, які можуть бути обчислені.

3.3. Проектування загальних обмежень.

4. Проектування файлових організацій та індексів:

4.1. Аналіз транзакцій.

4.2. Вибір файлових організацій.

4.3. Вибір індексів.

4.4. Оцінка вимог до дискового простору.

5. Проектування представлень користувачів.

6. Проектування механізмів безпеки.

7. Розгляд упровадження контрольованої надмірності.

8. Моніторинг та налаштування операційної системи.

Цю методологію можна використовувати для проектування як відносно простих, так і дуже складних систем баз даних. Подібно до того, як етап проектування бази даних життєвого циклу розробки систем баз даних складається з трьох етапів – концептуального, логічного та фізичного проектування, так само і ця методологія має три етапи. На етапі 1 створюється концептуальний проєкт бази даних, на етапі 2 – логічний проєкт бази даних, а на етапах з 3 по 8 – фізичний проєкт бази даних. Залежно від складності системи баз даних, що створюється, деякі із цих кроків можуть бути пропущені.

Проектування бази даних – це ітеративний процес, який має відправну точку і майже нескінченну низку вдосконалень. Знання, отримані на одному кроці, можуть змінити рішення, прийняті на попередньому кроці. Аналогічно, може бути корисним коротко розглянути наступний крок, щоб допомогти з попереднім. Таким чином, методологія повинна діяти як основа, що допомагає розробнику ефективно вести процес проектування бази даних.

Щоб проілюструвати методологію проектування бази даних, ми використовуємо приклад DreamUniversity. База даних DreamUniversity має декілька представлень користувачів, які управляються за допомогою поєднання централізованого підходу та підходу інтеграції представлень.

2.4. ER-моделювання. Побудова концептуальної моделі даних

Першим кроком у концептуальному проектуванні бази даних є створення однієї (або декількох) концептуальних моделей даних, що відповідають вимогам підприємства до даних. Концептуальна модель даних включає в себе:

- сутності;
- зв'язки;
- атрибути та домени атрибутів;
- первинні ключі та альтернативні ключі;
- обмеження цілісності.

Концептуальна модель даних підтримується документацією, включаючи ER-діаграми та словник даних, який створюється протягом розробки моделі.

Першим кроком у побудові концептуальної моделі даних є визначення основних об'єктів, які цікавлять користувачів. Ці об'єкти є сутностями моделі. Одним з методів ідентифікації сутностей є вивчення специфікації вимог користувачів. З цієї специфікації ми визначаємо іменники або іменникові

словосполучення, які згадуються (наприклад, ім'я та прізвище викладача, науковий ступінь, посада, кафедра, назва предмету, опис предмету, семестр). Ми також шукаємо основні об'єкти, такі як люди, місця або поняття, що становлять інтерес, виключаючи ті іменники, які є лише якостями інших об'єктів. Наприклад, ми можемо згрупувати *прізвище викладача* та *науковий ступінь* в об'єкт або сутність під назвою Professor, а *назва навчального предмету*, *опис предмету* – у сутність під назвою Subject.

Альтернативним способом ідентифікації сутностей є пошук об'єктів, які існують самі по собі. Наприклад, викладач є об'єктом, оскільки він існує незалежно від того, чи знаємо ми його ім'я, посаду та дату народження.

Не завжди очевидно, чи є певний об'єкт сутністю, відношенням або атрибутом. Наприклад, як би ми класифікували шлюб? Насправді, залежно від фактичних вимог, ми можемо класифікувати шлюб як будь-який із цих об'єктів або всі разом. Проєктувальники баз даних повинні дуже вибірково дивитися на світ і класифікувати речі, які вони спостерігають, у контексті предметної області. Таким чином, може не існувати унікального набору типів сутностей, що виводяться з даної специфікації вимог. Однак послідовні ітерації процесу проєктування повинні привести до вибору сутностей, які, принаймні, адекватні для необхідної системи. Для уявлень користувачів DreamUniversity ми виділили наступні сутності: Professor, Student, City, Group, Major, Department, Position, Score.

Сутності

Сутність – це група об'єктів з однаковими властивостями, які ідентифікуються як такі, що мають незалежне існування.

Сутність представляє групу «об'єктів» у «реальному світі» з однаковими властивостями. Сутності мають

незалежне існування і можуть бути об'єктами з фізичним (або «реальним») існуванням:

- Professor.
- Student,

або об'єктами з концептуальним (або «абстрактним») існуванням:

- Score.

Зауважте, що ми можемо дати лише робоче визначення сутності, оскільки строгого формального визначення не існує. Ми ідентифікуємо кожну сутність за назвою та переліком властивостей.

Екземпляр сутності – це однозначно ідентифікований об'єкт сутності.

Кожен унікально ідентифікований об'єкт сутності називається просто екземпляром сутності.

Схематичне представлення сутностей

На етапі проектування бази даних для більш простого представлення сутностей та їх зв'язків між собою ми будемо використовувати ER-діаграми (Entity-Relationship).

ER-діаграми можуть бути представлені за допомогою різних нотацій, кожна з яких має власний набір символів та умовних позначень для зображення сутностей, атрибутів і зв'язків. Найпоширеніші нотації: Chen, Crow's Foot, UML та Bachman.

Далі для проектування ми будемо використовувати нотацію Crow's Foot.

Кожна сутність відображається у вигляді прямокутника, позначеного ім'ям сутності, яке зазвичай є іменником в однині. Перша буква кожного слова в назві сутності є великою (наприклад, Student і Professor). На рис. 2.2 показано схематичне представлення сутностей.

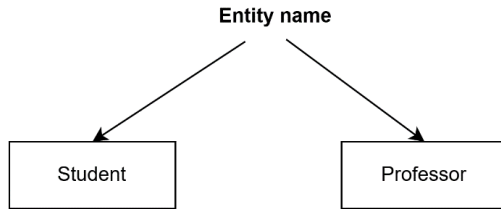


Рис. 2.2. Схема зображення сутностей Student і Professor

Ідентифікація зв'язків між сутностями

Після ідентифікації об'єктів наступним кроком є визначення всіх зв'язків, які існують між цими об'єктами. Коли ми ідентифікуємо сутності, один з методів, це пошук іменників у специфікації вимог користувачів. Ми можемо використовувати граматику специфікації вимог для визначення зв'язків. Зазвичай, зв'язки позначаються дієсловами або дієслівними виразами. Наприклад:

- певний викладач *є куратором* академічної групи (Professor «advises» Group);
- на кафедрі *працюють* певні викладачі (Department «employs» Professor).

Той факт, що специфікація вимог фіксує ці взаємозв'язки, свідчить про те, що вони важливі для підприємства і повинні бути включені в модель.

Нас цікавлять лише необхідні зв'язки між сутностями. Ми також можемо включити зв'язок між Professor і Student (наприклад, викладач є керівником кваліфікаційної роботи студента). Однак хоча це можливий зв'язок, зі специфікації вимог впливає, що це не той зв'язок, який ми зацікавлені моделювати.

Зв'язки

Зв'язок – це набір асоціацій між однією або декількома сутностями, що беруть участь у зв'язку. Кожному зв'язку присвоюється ім'я, яке описує його функцію.

Необхідно розрізняти терміни «зв'язок» та «екземпляр зв'язку».

Екземпляр зв'язку – це унікальний зв'язок, який включає по одному екземпляру з кожної сутності, що бере участь у зв'язку.

Розглянемо тип зв'язку «has», який представляє зв'язок між сутностями Group і Student, тобто Group «has» Student. Кожне входження зв'язку «has» пов'язує одне входження сутності Group з одним входженням сутності Student.

Схематичне представлення зв'язків

Кожен тип зв'язку показаний у вигляді лінії, що з'єднує пов'язані сутності і позначений назвою зв'язку. Зазвичай, зв'язок називається дієсловом (наприклад, «generates» або «operates») або короткою фразою, що включає в себе дієслово (наприклад, «has»), як показано на рис. 2.3. По можливості, назва зв'язку повинна бути унікальною для даної ER-моделі.

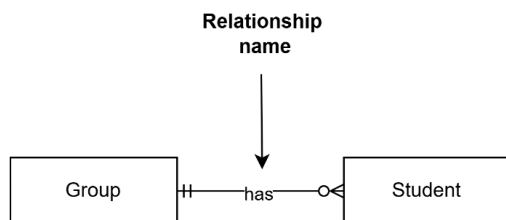


Рис. 2.3. Схеми зображення зв'язку Group «has» Student

Ступінь зв'язку

Ступінь зв'язку – це кількість сутностей, що беруть участь у зв'язку.

Сутності, що беруть участь у певному зв'язку, називаються *учасниками* цього зв'язку. Кількість учасників зв'язку називається *ступенем* цього зв'язку. Таким чином, ступінь зв'язку вказує на кількість сутностей, що беруть участь

у зв'язку. Відношення другого ступеня називається *бінарним*. Прикладом бінарного зв'язку є зв'язок «has», показаний на рис. 2.3, з двома типами сутностей, що беруть участь у зв'язку, а саме: Group і Student. Другим прикладом бінарного відношення є відношення «teaches», що показано на рис. 2.4 з двома типами сутностей, а саме: Professor і Subject.

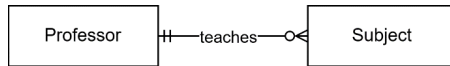


Рис. 2.4. Схема зображення зв'язку Professor «teaches» Subject

У більшості випадків зв'язки є бінарними; іншими словами, зв'язки існують між двома сутностями. Однак слід бути обережними і звертати увагу на складні зв'язки, які можуть включати в себе більше двох сутностей і рекурсивні зв'язки, які включають у себе лише одну сутність.

Зв'язок третього ступеня називається *тернарним*. Прикладом тернарного зв'язку є «score» з трьома суб'єктами-учасниками: Professor, Student та Subject. Цей зв'язок відображає оцінювання викладачем студента з певного предмету. Термін «складний зв'язок» використовується для опису зв'язків зі ступенями, вищими за бінарні.

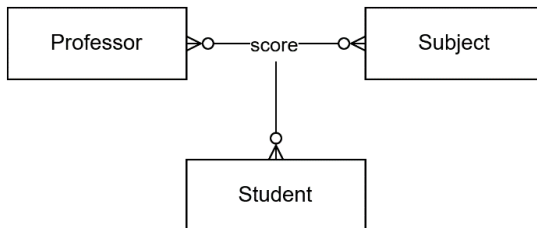


Рис. 2.5. Приклад тернарного зв'язку під назвою «score»

Рекурсивний зв'язок

Рекурсивний зв'язок – це тип зв'язку, у якому одна й та сама сутність бере участь у різних ролях.

Розглянемо рекурсивний зв'язок під назвою «reports to», який представляє об'єднання співробітників з керівником, де керівник також є співробітником. Іншими словами, сутність Professor двічі бере участь у зв'язку «reports to»: перший раз як керівник, а другий раз як працівник, який перебуває під наглядом (підлеглий). Рекурсивні зв'язки іноді називають *унарними зв'язками*.

Зв'язкам можуть бути надані імена ролей, щоб указати на призначення, яке кожна сутність, що беруть участь у зв'язку, відіграє у зв'язку. Назви ролей можуть бути важливими для рекурсивних зв'язків, щоб визначити функцію кожного учасника. Використання імен ролей для опису рекурсивного зв'язку «reports to» показано на рис. 2.6. Першому учаснику сутності Professor у зв'язку «reports to» присвоєно рольове ім'я Supervisor, а другому учаснику – рольове ім'я Supervisee.

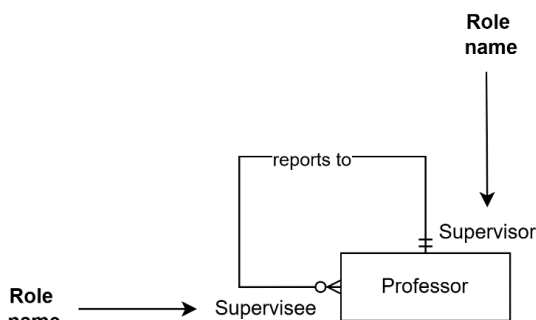


Рис. 2.6. Приклад рекурсивного зв'язку Professor (Supervisee) «reports to» Professor (Supervisor)

Назви ролей можуть також використовуватися, коли дві сутності пов'язані більш ніж одним зв'язком. Наприклад,

типи сутностей Professor і Department пов'язані двома різними зв'язками, які називаються «chairs» і «employs». Як показано на рис. 2.7, використання імен ролей пояснює призначення кожного зв'язку. Наприклад, у випадку зв'язку Professor «chairs» Department професор (сутність Professor) з іменем ролі Head керує кафедрою (сутність Department) з іменем ролі Office. Аналогічно, у випадку Department «employs» Professor, співробітник (сутність Professor) з назвою ролі Employee працює на кафедрі з назвою ролі Office.

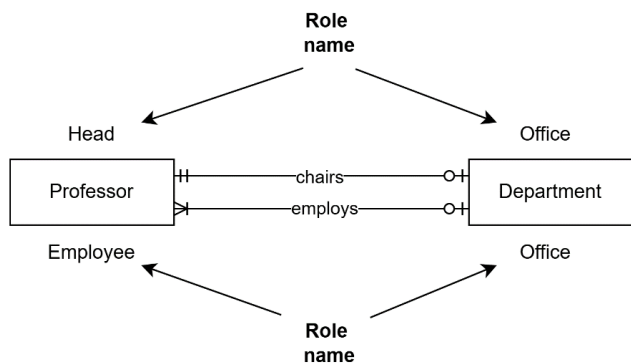


Рис. 2.7. Сутності з двома різними зв'язками «chairs» і «employs»

Назви ролей зазвичай не потрібні, якщо функції сутностей, що беруть участь у відносинах, є однозначними.

Структурні обмеження

Структурні обмеження повинні відображати обмеження зв'язків, як вони сприймаються у «реальному світі». Основний тип обмежень зв'язків називається *множинністю*.

Множинність – це кількість (або діапазон) можливих екземплярів сутності, які можуть стосуватися одного екземпляру пов'язаної сутності через певний зв'язок.

Множинність обмежує спосіб зв'язку між сутностями. Це представлення політик (або бізнес-правил), установлених користувачем або замовником.

Продемонструємо, як визначити множинність зв'язків і покажемо, як представити кожну з них на ER-діаграмі.

Кардинальність та обмеження участі

Множинність складається з двох окремих обмежень, відомих як **кардинальність** та **обов'язковість**.

Кардинальність описує максимальну кількість можливих взаємозв'язків для сутності, що бере участь у даному зв'язку.

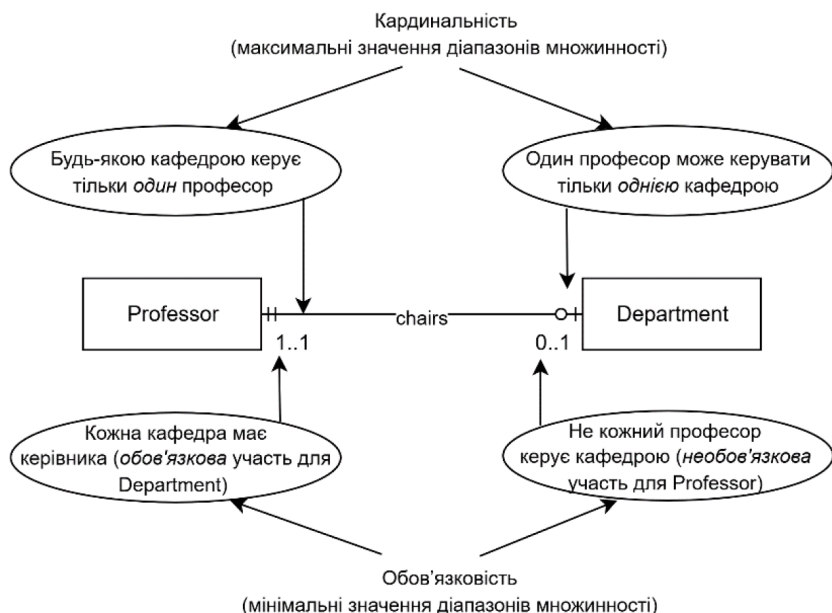


Рис. 2.8. Множинність, що описується як кардинальність та обмеження участі для зв'язку Professor «chairs» Department (1:1)

Кардинальність зв'язку відображається як максимальні значення діапазонів множинності по обидва боки зв'язку. Наприклад, для зв'язку Professor «chairs» Department, який показано на рис. 2.8, кардинальність представлено діапазонами множинності з максимальним значенням 1 по обидва боки зв'язку.

Обов'язковість – визначає, чи всі або тільки деякі екземпляри сутності беруть участь у зв'язку.

Обмеження участі відображає, чи всі екземпляри сутності беруть участь у певному зв'язку (що називається *обов'язковою* участю), чи тільки деякі (що називається *необов'язковою* участю). Участь сутностей у зв'язку відображається як мінімальні значення діапазонів множинності з обох боків зв'язку. Необов'язкова участь позначається мінімальним значенням 0, а обов'язкова участь – мінімальним значенням 1. Важливо зазначити, що участь певної сутності у зв'язку позначається мінімальним значенням на протилежному боці відношення, тобто мінімальним значенням множинності поруч із пов'язаною сутністю. Наприклад, на рис. 2.8 необов'язкова участь сутності Professor у зв'язку «chairs» показана як мінімальне значення 0 для множинності поруч із сутністю Department, а обов'язкова участь сутності Department у зв'язку «chairs» показана як мінімальне значення 1 для множинності поруч із сутністю Professor.

Визначення множинності

Визначення множинності зазвичай вимагає вивчення точних взаємозв'язків між даними, наведеними в описі предметної області, з використанням вибірових даних. Вибіркові дані можна отримати шляхом вивчення заповнених форм або звітів і, якщо можливо, у результаті обговорення з користувачами. Однак важливо підкреслити, що для отримання правильних висновків щодо обмеження необхідно,

щоб вибіркові дані, які вивчаються або обговорюються, були справжнім відображенням усіх даних, що моделюються.

Типи зв'язків

Як ми вже згадували раніше, найпоширенішим ступенем зв'язку є бінарний. Бінарні зв'язки зазвичай називають зв'язками «один до одного» (1:1), «один до багатьох» (1:*) або «багато до багатьох» (*:*). Ми розглянемо ці три типи зв'язків, використовуючи такі обмеження цілісності:

- професор (викладач) керує кафедрою: Professor «chairs» Department (1:1);
- на кафедрі працюють викладачі: Department «employs» Professor (1:*);
- викладачі викладають предмети Professor «teaches» Subject (*:*)

Зв'язок «один до одного» (1:1)

Розглянемо зв'язок «chairs», які пов'язують типи сутностей Professor і Department: професор (викладач) керує кафедрою: Professor «chairs» Department. Професор (викладач) може керувати або однією кафедрою, або взагалі не керувати жодною, а кожна кафедра керується тільки одним професором (викладачем).

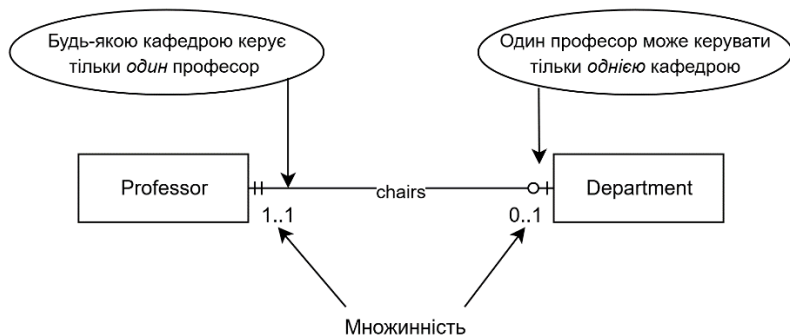


Рис. 2.9. Тип зв'язку «один до одного» (1:1)
Professor «chairs» Department

Оскільки для кожного професора (екземпляр сутності Professor), залученого до зв'язку «chairs» («керує»), існує максимум одна кафедра (екземпляр сутності Department), а для кожної кафедри (екземпляр сутності Department) – максимум один керівник (екземпляр сутності Professor). Цей тип зв'язку називається «один до одного», що зазвичай скорочується як (1:1).

Зв'язок «один до багатьох» (1:*)

Розглянемо зв'язок «employs», які пов'язують типи сутностей Department і Professor: професор (викладач) працює на кафедрі: Department «employs» Professor.

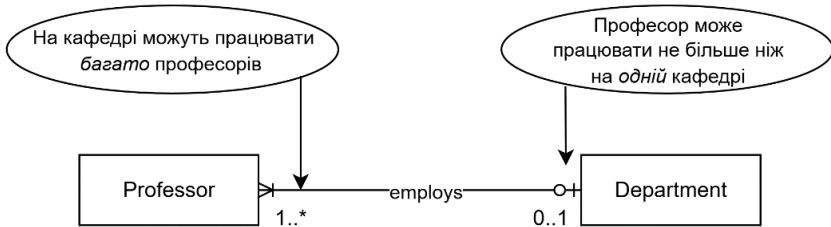


Рис. 2.10. Тип зв'язку «один до багатьох» (1:*)
Department «employs» Professor

Професор (викладач) може працювати тільки на одній кафедрі або може взагалі не працювати на кафедрі, а на кафедрі можуть працювати багато викладачів. Отже, для кожного викладача (екземпляр сутності Professor), який бере участь у зв'язку «employs», існує максимум одна кафедра, де він працює (екземпляр сутності Department), а для будь-якої кафедри (екземпляр сутності Department), що бере участь у зв'язку «employs», може бути багато викладачів (екземплярів сутності Professor). Цей тип зв'язку називається «один до багатьох», що зазвичай скорочується як (1:*)

Зв'язок «багато до багатьох» (*:*)

Розглянемо зв'язок «teaches», які пов'язують типи сутностей Professor і Subject: професор (викладач) викладає предмет: Professor «teaches» Subject. Професор (викладач) може викладати багато предметів, а один предмет (наприклад, «вища математика») можуть викладати різні викладачі (тобто багато викладачів). Отже, для кожного викладача (екземпляр сутності Professor), який бере участь у зв'язку «teaches», існує багато предметів, які він викладає (екземпляри сутності Subject), а для будь-якого предмету (екземпляр сутності Subject), що бере участь у зв'язку «teaches», може бути багато викладачів (екземплярів сутності Professor), які викладають цей предмет. Цей тип зв'язку називається «багато-до-багатьох», що зазвичай скорочується як (*:*)

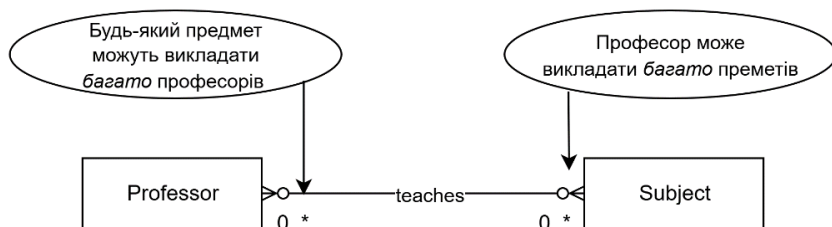


Рис. 2.11. Тип зв'язку «багато до багатьох» (*:*)
Professor «teaches» Subject

Діаграма ER, показана на рис. 2.12, є прикладом одного з можливих початкових результатів моделювання ER. Ця модель відображає взаємозв'язки між основними сутностями, описаними у специфікації вимог для прикладу DreamUniversity.

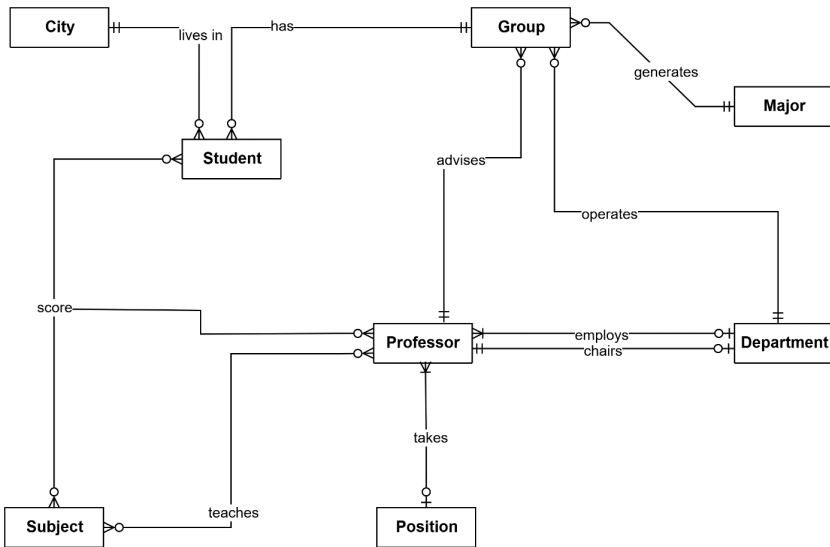


Рис. 2.12. Діаграма «сутність-зв'язок» (ER) першого розрізу інформаційної системи DreamUniversity

Атрибути

Атрибут – це властивість сутності або типу відношення.

Окремі властивості типів сутностей називаються *атрибутами*. Наприклад, тип сутності *Professor* може бути описаний атрибутами *professor_id*, *first_name*, *last_name*, *position_id* і *degree*. Атрибути містять значення, що описують кожний екземпляр сутності, і представляють основну частину даних, що зберігаються в базі даних.

Зв'язки, що пов'язують сутності, також можуть мати атрибути, подібні до атрибутів сутності.

Домен атрибуту – це набір допустимих значень для одного або декількох атрибутів.

Кожен атрибут пов'язаний з набором значень, який називається *доменом*. Домен визначає потенційні значення, які може мати атрибут. Наприклад, кількість кімнат, пов'язаних

з об'єктом нерухомості, становить від 1 до 15 для кожного екземпляра сутності. Тому ми визначаємо набір значень для атрибуту кількості кімнат (*rooms*) типу сутності *PropertyForRent* як набір цілих чисел від 1 до 15.

Атрибути можуть мати спільний домен. Наприклад, атрибути *first_name*, *last_name* сутностей *Professor* та *Student* мають спільний домен усіх можливих імен (для *first_name*) та прізвищ (для *last_name*). Домени також можуть складатися з доменів. Наприклад, домен для атрибуту адреси *street_address* сутності *Student* складається з піддоменів: вулиця, будинок та поштовий індекс.

Домен атрибуту *first_name* визначити складно, оскільки він складається з усіх можливих імен. Це, безумовно, символний рядок, але він може складатися не тільки з літер, а і з дефісів або інших спеціальних символів. Простіше визначити домен, який представляє собою, наприклад, діапазон чисел або дат.

Повністю розроблена модель даних включає в себе домени кожного атрибуту в моделі ER.

Як ми зараз пояснимо, атрибути можна класифікувати як прості або складені, однозначні або багатозначні, або похідні.

Прості та складені атрибути

Простий атрибут – це атрибут, що складається з одного компонента з незалежним існуванням.

Прості атрибути не можуть бути додатково поділені на менші компоненти. Прикладами простих атрибутів є *first_name* та *last_name* викладача (сутність *Professor*). Прості атрибути іноді називають *атомарними атрибутами*.

Складений атрибут – це атрибут, що складається з декількох компонентів, кожен з яких існує незалежно.

Деякі атрибути можна поділити на менші компоненти, які існують незалежно один від одного. Наприклад, атрибут

адреси *street_address* сутності *Student* можна поділити на атрибути вулиці, будинка та поштового індексу.

Рішення про моделювання атрибуту адреси як простого атрибуту або про поділ атрибуту на вулицю, місто та пошто-вий індекс залежить від того, чи користувач розглядає атрибут адреси як єдине ціле, чи як окремі компоненти.

Однозначний атрибут – атрибут, що містить одне значення для кожного входження типу сутності.

Більшість атрибутів є однозначними. Наприклад, кожний екземпляр сутності *Department* має одне значення для атрибуту *name* (назва кафедри), і тому атрибут *name* називається *однозначним*.

Багатозначний атрибут – це атрибут, який містить кілька значень для кожного випадку типу сутності.

Деякі атрибути можуть мати кілька значень для кожного екземпляру сутності. Наприклад, кожний екземпляр сутності *Department* може мати кілька значень для атрибута *telNo* (кафедра має декілька телефонних номерів), і тому атрибут *telNo* в цьому випадку є багатозначним.

Похідний атрибут – це атрибут, який представляє значення, яке можна вивести із значення пов'язаного атрибуту або набору атрибутів, не обов'язково в тому самому типі сутності.

У деяких випадках значення похідного атрибута виводиться з кількості екземплярів сутності в тій самій сутності або похідні атрибути також можуть включати в себе асоціацію атрибутів різних сутностей. Наприклад, атрибут *totalProfessor* (загальна кількість викладачів кожної кафедри) сутності *Department* можна обчислити, підрахувавши кількість екземплярів сутності *Professor* для кожної кафедри.

Ключи

Потенційний ключ – це мінімальний набір атрибутів, який однозначно ідентифікує кожне входження типу сутності.

Потенційний ключ – це мінімальна кількість атрибутів, значення яких однозначно ідентифікують кожне входження сутності. Наприклад, атрибут код кафедри (*department_id*) є потенційним ключем для типу сутності Department і має окреме значення для кожного входження сутності Department. Потенційний ключ повинен містити значення, які є унікальними для кожного екземпляра сутності. Це означає, що потенційний ключ не може містити значення *null*. Наприклад, кожна кафедра має унікальний номер кафедри, і ніколи не буде більше однієї кафедри з тим самим номером кафедри.

Первинний ключ – потенційний ключ, який вибирається для унікальної ідентифікації кожного входження екземпляру сутності.

Сутність може мати більше одного потенційного ключа. Розглянемо, що викладач має унікальний номер викладача (*professor_id*), який визначений закладом освіти, а також унікальний номер картки платника податків (*tax_card*), який використовується урядом. Отже, ми маємо два ключі-кандидати для сутності Professor, один з яких повинен бути обраний як первинний ключ.

Вибір первинного ключа для сутності базується на таких міркуваннях, як довжина атрибуту, мінімальна кількість необхідних атрибутів та майбутня впевненість в унікальності. Наприклад, номер співробітника, визначений закладом освіти, містить максимум, наприклад, п'ять символів, а номер картки платника податків містить максимум десять символів. Тому ми вибираємо *professor_id* як первинний ключ типу сутності Professor, а *tax_card* потім називається *альтернативним ключем*.

Складений ключ – потенційний ключ, що складається з двох або більше атрибутів.

У деяких випадках ключ сутності складається з декількох атрибутів, значення яких разом є унікальними для кожного входження сутності, але не окремо. Наприклад, розглянемо

сутність Professor з атрибутами *first_name* (ім'я) та *last_name* (прізвище). Багато людей мають однакові імена. Для унікальної ідентифікації кожного викладача сутності Professor необхідні обидва значення атрибутів *first_name* та *last_name*.

Таким чином, сутність Professor може мати складений первинний ключ, що складається з атрибутів *first_name* та *last_name* (за відсутністю більш простих потенційних ключів).

Представлення атрибутів на ER-діаграмі

Якщо сутність має бути відображена разом з її атрибутами, ми ділимо прямокутник, що представляє сутність, на дві частини. У верхній частині прямокутника відображається назва сутності, а в нижній частині перелічуються назви атрибутів. Наприклад, на рис. 2.13 показано ER-діаграму для сутностей Professor і Department та пов'язаних з ними атрибутів.

Першим атрибутом, що відображається, є первинний ключ для сутності, якщо він відомий. Імена атрибутів первинного ключа можуть позначатися тегом **ПК** (Primary Key). Як показано на рис. 2.13, первинним ключем сутності Professor є атрибут *professor_id*, а первинним ключем сутності Department є атрибут *department_id*.

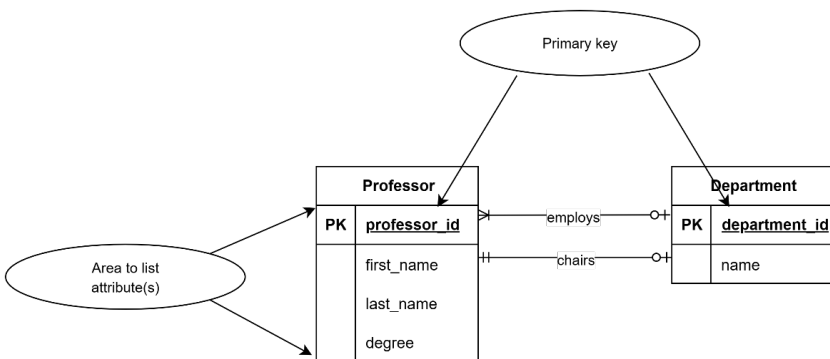


Рис. 2.13. Графічне представлення сутностей Professor і Department та їх атрибутів

Для простих однозначних атрибутів немає потреби використовувати теги, тому ми просто відображаємо імена атрибутів у списку під назвою «сутності». Для складених атрибутів ми перелічуємо назву складеного атрибуту, а нижче, з відступом праворуч, – назви його простих складових атрибутів.

У складних системах баз даних ми показуємо лише атрибут або атрибути, що утворюють первинний ключ кожної сутності (рис. 2.14). Коли в ER-діаграмі показані лише атрибути первинного ключа, ми можемо опустити тег PK.

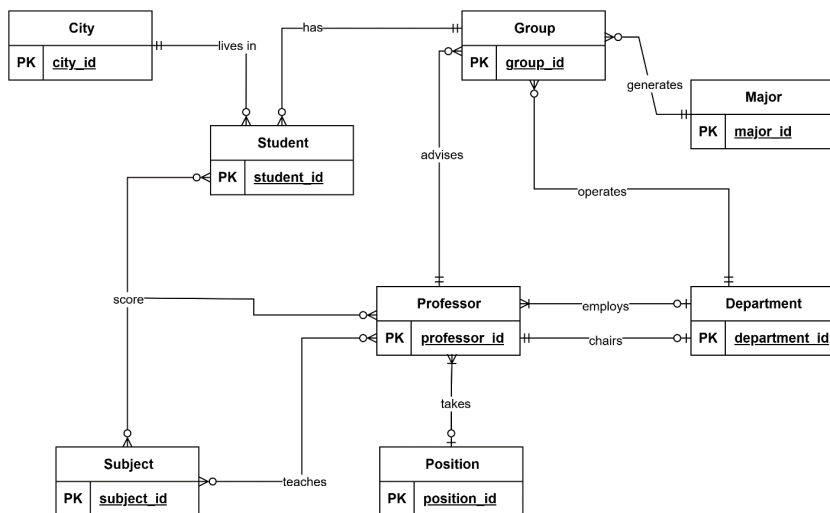


Рис. 2.14. ER-діаграма інформаційної системи DreamUniversity з доданими первинними ключами

Атрибути зв'язків

Атрибути також можуть бути приписані до зв'язків. Наприклад, розглянемо зв'язок «teaches», яке пов'язує сутності Professor і Subject, як показано на рис. 2.15. Щоб записати навчальний рік, коли професор викладає предмет, ми

пов'язуємо цю інформацію зі зв'язком «teaches» як атрибут під назвою *year*, а не із сутностями Professor або Subject.

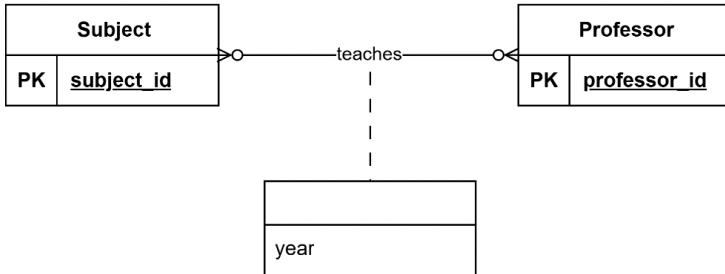


Рис. 2.15. Приклад зв'язку «teaches» з атрибутом *year*

Ми позначаємо атрибути, пов'язані зі зв'язком, тим самим символом, що й сутність. Однак, щоб відрізнити зв'язок з атрибутом від сутності, прямокутник, що позначає атрибут(и) зв'язку, пов'язується зі зв'язком за допомогою пунктирної лінії.

Ми використовуємо ER-діаграми, щоб простіше представити сутності, їх атрибути та взаємозв'язки між сутностями. Протягом усього етапу проектування бази даних ми рекомендуємо використовувати ER-діаграми, щоб скласти уявлення про ту предметну область, яку ми моделюємо.

Для більш повного представлення бази даних, що проектується, краще показати всі атрибути для кожної сутності в ER-діаграмі (рис. 2.16).

Концептуальна модель даних включає в себе діаграму ER та супровідну документацію, що описує модель даних. Якщо в моделі даних є якісь аномалії, ми повинні внести відповідні зміни, що може вимагати повторення попереднього кроку (кроків). Ми повторюємо цей процес, доки користувач не буде готовий «підписати» модель як «правдиве» відображення тієї предметної області, яку ми моделюємо.

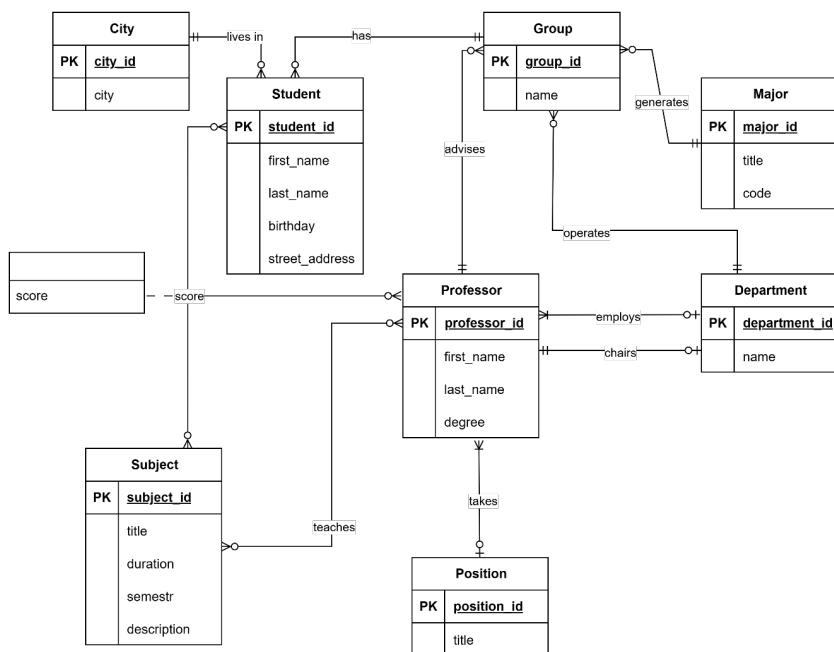


Рис. 2.16. Діаграма ER інформаційної системи DreamUniversity з доданими атрибутами

2.5. Побудова логічної моделі бази даних

На цьому етапі головна мета полягає в перетворенні концептуальної моделі даних, створеної на попередньому етапі, у логічну модель даних, що відповідає вимогам предметної області.

Спочатку ми виводимо набір відносин (реляційну схему) з концептуальної моделі даних, створеної на попередньому кроці. Структура реляційної схеми перевіряється за допомогою нормалізації. Далі логічна модель даних перевіряється користувачами, щоб переконатися, що вони вважають модель справжнім відображенням вимог предметної області.

Ми будемо демонструвати етап логічного проектування, використовуючи концептуальну модель даних, яка створена в попередньому розділі для прикладу DreamUniversity і представлена на рис. 2.16 у вигляді ER-діаграми.

Визначення відношень (таблиць) для логічної моделі даних

На цьому етапі ми визначаємо відношення для логічної моделі даних, щоб представити сутності та атрибути. Ми визначаємо склад кожного відношення за наступною схемою: спочатку вказуємо назву відношення, а потім список простих атрибутів відношення. Потім ми визначаємо первинний ключ та будь-які альтернативні та/або зовнішні ключі відношення.

Зв'язки, які існують між двома сутностями, представлені механізмом первинного ключа / зовнішнього ключа. При прийнятті рішення про те, де розмістити (або розмістити) атрибути зовнішнього ключа, ми повинні спочатку визначити «батьківську» і «дочірню» сутності, які беруть участь у зв'язку. *Батьківська сутність* – це сутність, яка розміщує копію свого первинного ключа у відношенні, що представляє дочірню сутність, для виконання функції зовнішнього ключа.

Тип бінарного зв'язку «один до багатьох» (1:*)

Для кожного бінарного зв'язку 1:* сутність на стороні «один» бінарного зв'язку визначається як «батьківська» сутність, а сутність на стороні «багато» визначається як дочірня суть. Щоб реалізувати цей зв'язок, ми розміщуємо копію атрибуту (атрибутів) первинного ключа «батьківської» сутності у відношенні, що представляє «дочірню» сутність, як зовнішній ключ.

Наприклад, відношення Department «employs» Professor, показане на рис. 2.17, є відношенням 1:*, оскільки на одній кафедрі можуть працювати багато викладачів. У цьому

прикладі Department знаходиться на стороні «один» і представляє «батьківську» сутність, а Professor знаходиться на стороні «багато» і представляє «дочірню» сутність. Зв'язок між цими сутностями встановлюється шляхом розміщення копії первинного ключа сутності Department («батьківської»), *department_id*, у відношенні Professor («дочірньої») як зовнішнього ключа. Поля зовнішнього ключа помічаються тегом **FK** (Foreign Key).

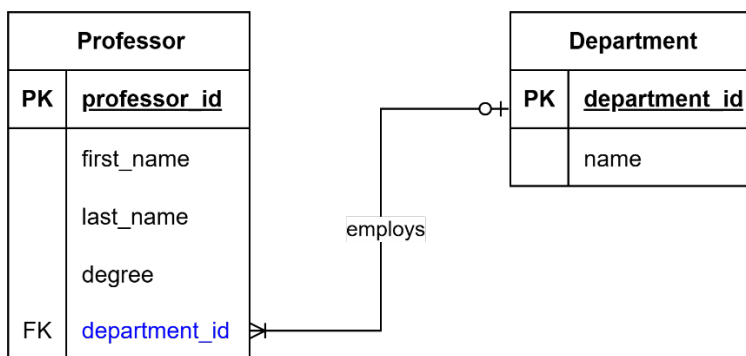


Рис. 2.17. Установлення бінарного зв'язку «один до багатьох» (1:*) за допомогою зовнішнього ключа (FK)

У випадку, коли зв'язок 1:* має один або більше атрибутів, ці атрибути також разом з атрибутами первинного ключа повинні бути розміщені у «дочірньому» відношенні. Наприклад, якщо зв'язок «employs» має атрибут *date_employment*, що позначає дату початку роботи викладача на кафедрі, цей атрибут також повинен бути розміщений у відношенні Professor разом із копією первинного ключа відношення Department, а саме *department_id*.

Тип бінарного зв'язку «один до одного» (1:1)

Створення відношень для представлення зв'язку 1:1 є дещо складнішим, оскільки кардинальність не може бути використана для ідентифікації батьківських та дочірніх

сутностей у зв'язку. Натомість, обмеження участі використовуються для визначення, чи краще представляти зв'язок шляхом об'єднання залучених сутностей в одне відношення, чи шляхом створення двох відношень та розміщення копії первинного ключа з одного відношення в інше. Розглянемо, як створити відносини для представлення таких обмежень участі:

- а) обов'язкова участь з обох сторін зв'язку 1:1;
- б) обов'язкова участь з однієї сторони зв'язку 1:1;
- с) необов'язкова участь з обох сторін зв'язку 1:1.

Обов'язкова участь обох сторін у зв'язку 1:1 (а)

У цьому випадку ми повинні об'єднати залучені сутності в одне відношення і вибрати один з первинних ключів оригінальних сутностей як первинний ключ нового відношення, а інший (якщо він існує) використовувати як альтернативний ключ.

У випадку, коли зв'язок 1:1 з обов'язковою участю обох сторін має один або кілька атрибутів, ці атрибути також повинні бути включені в об'єднане відношення.

Зверніть увагу, що об'єднати дві сутності в одне відношення можна лише в тому випадку, якщо між цими двома сутностями немає інших прямих зав'язків, які б цьому перешкоджали, таких як зв'язки 1:*. У такому випадку нам потрібно було б представити зв'язок 1:1 за допомогою механізму первинного ключа / зовнішнього ключа.

Обов'язкова участь в одній стороні зв'язку 1:1 (б)

У цьому випадку ми можемо ідентифікувати «батьківську» та «дочірню» сутності для зв'язку 1:1, використовуючи обмеження участі. Сутність, яка має необов'язкову участь у зв'язку, позначається як «батьківська» сутність, а сутність, яка має обов'язкову участь у зв'язку, позначається як «дочірня» суть. Як описано раніше, копія первинного ключа «батьківської» сутності розміщується у відношенні, що представляє «дочірню» сутність. Якщо зв'язок має один або

більше атрибутів, ці атрибути також розміщуються разом з копією первинного ключа у «дочірньому» відношенні.

Необов'язкова участь обох сторін у зв'язку 1:1 (с)

У цьому випадку призначення «батьківської» та «дочірньої» сутностей є довільним.

Тип бінарного зв'язку «багато до багатьох» (*:*)

Для кожного бінарного зв'язку *: * створить відношення, яке представляє цей зв'язок, і включить у нього всі атрибути, що є частиною цього зв'язку. Ми розміщуємо копію атрибутів первинного ключа сутностей, що беруть участь у зв'язку, у новому відношенні, щоб вони виконували функцію зовнішніх ключів. Ці зовнішні ключі також можуть утворювати складений первинний ключ нового відношення, можливо, у поєднанні з одним або кількома атрибутами відношення.

Наприклад, розглянемо зв'язок *: * Professor «teaches» Subject, показане на рис. 2.18. Щоб відобразити це, ми створюємо відношення для сутностей Professor і Subject, а також створюємо відношення SubjectProfessor, щоб відобразити відношення «teaches», що дає:

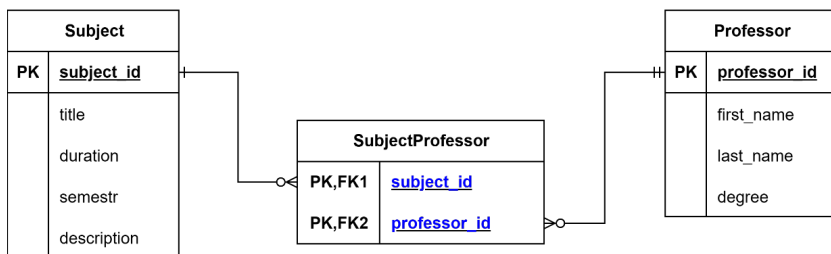


Рис. 2.18. Установлення бінарного зв'язку «багато до багатьох» (*:*) між сутностями Professor і Subject

У випадку, якщо зв'язок «багато до багатьох» має один або більше атрибутів, ці атрибути також розміщуються разом з копіями первинних ключів у додатковому відношенні.

Наприклад, якщо необхідно зберігати інформацію про навчальний рік, коли професор викладає предмет, ми пов'язуємо цю інформацію зі зв'язком «teaches» як атрибут під назвою *year* та додаємо його у відношення SubjectProfessor.

Складні зв'язки, які включають у себе більше двох сутностей

Для складного зв'язку створить відношення, яке представляє цей зв'язок, і включите усі атрибути, які є частиною цього зв'язку. Ми розміщуємо копію атрибутів первинного ключа сутностей, які беруть участь у складному зв'язку, у новому відношенні, щоб вони діяли як зовнішні ключі.

Розглянемо тернарний зв'язок «score» з трьома сутностями-учасниками: Professor, Student та Subject. На рис. 2.19 показано взаємозв'язок між трьома сутностями, який відображає оцінювання викладачем студента з певного предмету.

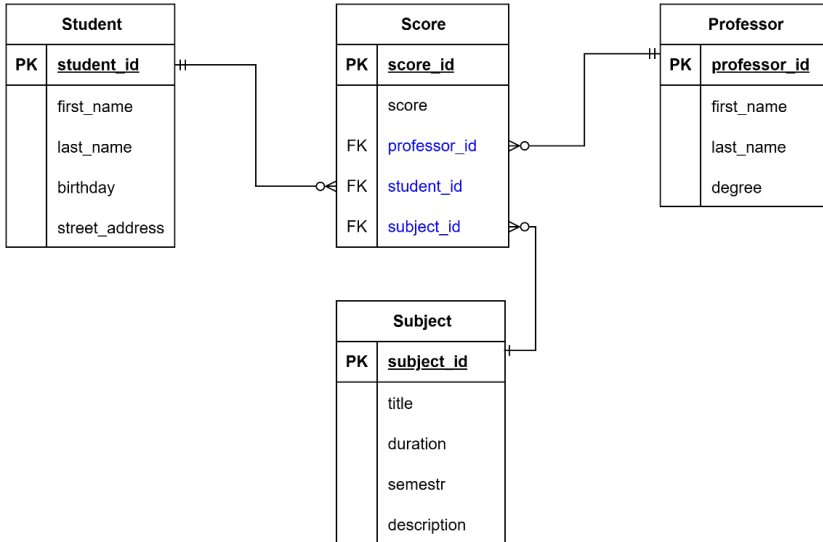


Рис. 2.19. Установлення тернарного зв'язку між сутностями Professor, Student та Subject

Щоб представити це, ми створюємо відношення для сутностей Professor, Student та Subject, а також створюємо відношення Score, щоб представити взаємозв'язок «score».

У випадку, якщо тернарний зв'язок має один або більше атрибутів, ці атрибути також розміщуються разом з копіями первинних ключів у додатковому відношенні, яке відображає зв'язок.

Багатозначні атрибути

Для кожного багатозначного атрибуту в сутності створіть нове відношення, яке представлятиме багатозначний атрибут, і включіть первинний ключ сутності в нове відношення, щоб він виконував функцію зовнішнього ключа. Якщо багатозначний атрибут сам по собі не є альтернативним ключем сутності, первинний ключ нового відношення є комбінацією багатозначного атрибуту та первинного ключа сутності.

Наприклад, у представленнях користувача для відображення ситуації, коли одна кафедра має до трьох телефонних номерів, атрибут *telNo* сутності Department визначено як багатозначний атрибут. Щоб відобразити це, ми створюємо відношення для сутності Department і створюємо нове відношення під назвою Telephone з атрибутами *telNo* – первинний ключ, *department_id* – зовнішній ключ.

Перегляд логічної моделі даних разом з користувачами

Побудовану логічну модель даних необхідно переглянути разом з користувачами, щоб переконатися, що вони вважають модель справжнім відображенням вимог предметної області.

Логічна модель даних має бути завершеною (рис. 2.20) та повністю задокументованою. Однак, щоб підтвердити це, користувачів просять переглянути логічну модель даних, щоб переконатися, що вони вважають модель справжнім відображенням вимог підприємства до даних. Якщо

користувачі незадоволені моделлю, може знадобитися повторення деяких попередніх кроків методології.

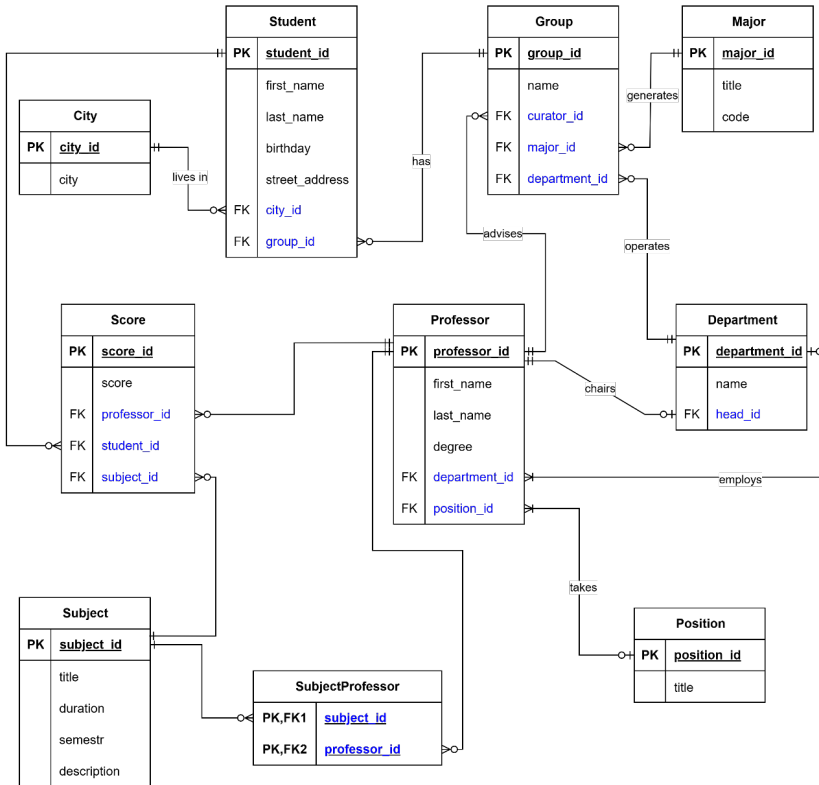


Рис. 2.20. Логічна модель даних інформаційної системи DreamUniversity

Перевірка відношень за допомогою нормалізації

Для представлення концептуальної моделі даних, створеної на попередньому кроці, на цьому етапі ми отримали набір відношень для логічної моделі даних. Далі необхідно перевірити групування атрибутів у кожному відношенні за допомогою правил нормалізації. Мета нормалізації полягає

в тому, щоб забезпечити, щоб набір відносин мав мінімальну, але достатню кількість атрибутів, необхідних для підтримки вимог предметної області. Крім того, відносини повинні мати мінімальну надмірність даних, щоб уникнути проблем аномалій оновлення, які будуть розглянути далі. Однак деяка надмірність є необхідною для об'єднання пов'язаних відносин.

2.6. Нормалізація

Основною метою проектування реляційної бази даних є групування атрибутів у відношення для мінімізації надлишковості даних. Якщо ця мета досягнута, то потенційні переваги реалізованої бази даних включають наступне:

- оновлення даних, що зберігаються в базі даних, досягається за допомогою мінімальної кількості операцій, таким чином зменшуючи можливості для неузгодженості даних, що виникають у базі даних;
- скорочення простору для зберігання файлів, необхідного для базових відношень, що дозволяє мінімізувати витрати.

Звичайно, реляційні бази даних також покладаються на існування певної надлишковості даних. Ця надлишковість існує у вигляді копій первинних ключів (або ключів-кандидатів), які виступають у ролі зовнішніх ключів у зв'язаних відношеннях, що дозволяє моделювати зв'язки між даними.

У цьому розділі ми проілюструємо проблеми, пов'язані з небажаною надлишковістю даних, порівнюючи відношення Student та Major, показані у табл. 2.1 та 2.2, з відношенням StudentMajor, показаним у табл. 2.3. Відношення StudentMajor є альтернативним форматом відношень Student і Major. Ці відношення мають наступний вигляд:

Таблиця 2.1. Student

student_id	sname	rating	major_id	funding_source
МК0101	Назар Романишин	85	F3	бюджет
МК0210	Аркадій Мазайло	73	F3	контракт
МК1101	Павло Яворівський	90	F2	бюджет
МК2121	Юрко Могилевський	65	F4	бюджет
МК1314	Петро Сідлецький	82	F6	контракт
МК1111	Борис Майборода	81	F6	контракт
МК1234	Сніжана Яцьків	79	F2	бюджет
МК0102	Максим Сірко	98	F3	контракт
МК1052	Віталіна Карась	87	F4	бюджет
МК1024	Олександр Ющик	60	F3	контракт

Таблиця 2.2. Major

major_id	mtitle
F2	Інженерія програмного забезпечення
F3	Комп'ютерні науки
F4	Системний аналіз та наука про дані
F6	Інформаційні системи і технології

Таблиця 2.3. StudentMajor

student_id	sname	rating	major_id	funding_source	mtitle
МК0101	Назар Романишин	85	F3	бюджет	Комп'ютерні науки
МК0210	Аркадій Мазайло	73	F4	контракт	Системний аналіз та наука про дані
МК1101	Павло Яворівський	90	F2	бюджет	Інженерія програмного забезпечення
МК2121	Юрко Могилевський	65	F4	бюджет	Системний аналіз та наука про дані
МК1314	Петро Сідлецький	82	F2	контракт	Інформаційні системи і технології

Продовж. табл. 2.3.

student_id	sname	rating	major_id	funding_source	mtitle
MK1111	Борис Майборода	81	F6	контракт	Інформаційні системи і технології
MK1234	Сніжана Яцьків	79	F2	бюджет	Інженерія програмного забезпечення
MK0102	Максим Сірко	98	F3	контракт	Комп'ютерні науки
MK1052	Віталіна Карась	87	F4	бюджет	Системний аналіз та наука про дані
MK1024	Олександр Ющик	60	F3	контракт	Комп'ютерні науки

У відношенні StudentMajor є надлишкові дані; інформація про спеціальність повторюється для кожного студента, який навчається за нею. На відміну від цього, у відношенні Major дані про спеціальність з'являються лише один раз для кожної спеціальності, а у відношенні Student повторюється лише номер спеціальності (major_id), щоб показати, за якою спеціальністю навчається студент. У відношеннях з надлишковими даними можуть виникати проблеми, які називаються *аномаліями оновлення*, що класифікуються як аномалії вставки, видалення або модифікації.

Аномалії вставки

Існує два основні типи аномалії вставки, які ми проілюструємо за допомогою відношення StudentMajor, показаного в табл. 2.3.

Щоб вставити дані про нових студентів у відношення StudentMajor, ми повинні включити дані про спеціальність, за якою буде навчатися студент. Наприклад, щоб вставити дані про нових студентів, які навчатимуться за спеціальністю під номером F3, ми повинні ввести правильні дані (назву) про спеціальність за номером F3 так, щоб дані про

спеціальність відповідали значенням для спеціальності F3 в інших кортежах відношення StudentMajor. Відношення, показані у табл. 2.1 та 2.2, не страждають від цієї потенційної неузгодженості, тому що ми вводимо тільки відповідний номер спеціальності для кожного студента у відношенні Student. Натомість дані про спеціальність з номером F3 записані в базі даних як єдиний кортеж у відношенні Major.

Щоб вставити дані про нову спеціальність, за якою наразі не навчається жодний студент, у відношення StudentMajor, необхідно ввести nulls в атрибути для студентів, такі як student_id. Однак, оскільки student_id є первинним ключем для відношення StudentMajor, спроба ввести nulls для student_id порушує цілісність сутності, і не допускається. Тому ми не можемо ввести кортеж для нової спеціальності у відношення StudentMajor із значенням null для student_id. Дизайн відношень, показаний у табл. 2.1 та 2.2, дозволяє уникнути цієї проблеми, оскільки дані спеціальності вводяться у відношення Major окремо від даних студента. Дані про студентів, які навчаються за спеціальністю, вводяться пізніше у відношення Student.

Аномалії видалення

Якщо ми видалимо кортеж з відношення StudentMajor, який представляє останнього студента, що навчається за спеціальністю, інформація про цю спеціальність також буде втрачена з бази даних. Наприклад, якщо ми видалимо кортеж для студента під номером МК1111 (Борис Майборода) з відношення StudentMajor, інформація про спеціальність під номером F6 буде втрачена з бази даних. Дизайн відношень у табл. 2.1 та 2.2 дозволяє уникнути цієї проблеми, оскільки кортежі спеціальностей зберігаються окремо від кортежів студентів у різних відношеннях, і лише атрибут major_id пов'язує ці два відношення. Якщо ми видалимо кортеж з номером студента МК1111 з відношення Student,

дані про спеціальність F6 залишаються незмінними у відношенні Major.

Аномалії модифікації

Якщо ми хочемо змінити значення одного з атрибутів певної спеціальності у відношенні StudentMajor – наприклад, назву для спеціальності з номером F4 – ми повинні оновити кортежі всіх студентів, які навчаються за цією спеціальністю. Якщо цю модифікацію не буде виконано для всіх відповідних кортежів відношення StudentMajor, база даних стане неузгодженою. У цьому прикладі спеціальність з номером F4 може мати різні назви в різних кортежах студентів.

Попередні приклади ілюструють, що відношення Student і Major у табл. 2.1 та 2.2 мають більш бажані властивості, ніж відношення StudentMajor у табл. 2.3. Це демонструє, що, хоча відношення StudentMajor піддається аномаліям оновлення. Ми можемо уникнути цих аномалій, застосувавши декомпозицію до відношення StudentMajor, та отримавши два окремі відношення Student і Major. Існують дві важливі властивості декомпозиції більшого відношення на менші відношення:

- Властивість «без втрат» гарантує, що будь-який екземпляр вихідного відношення може бути ідентифікований за відповідними екземплярами у менших відношеннях.
- Властивість збереження залежності гарантує, що обмеження на вихідне відношення можна зберегти, просто наклавши певне обмеження на кожне з менших відношень.

Іншими словами, нам не потрібно виконувати об'єднання над меншими відношеннями, щоб перевірити, чи не порушено обмеження на вихідне відношення.

Процес нормалізації. Основні концепції та етапи

Нормалізація баз даних – це процес організації даних у відношеннях, з метою зменшення надмірності та

запобігання аномаліям під час оновлення, додавання чи видалення даних. Нормалізація забезпечує структурованість даних, поліпшує цілісність та сприяє їх ефективній обробці.

Основні властивості нормалізації

Властивість «без втрат» гарантує, що будь-який екземпляр вихідного відношення може бути ідентифікований за відповідними екземплярами у менших відношеннях. Властивість збереження залежності гарантує, що обмеження на вихідне відношення можна зберегти, просто наклавши певне обмеження на кожне з менших відношень. Інакше кажучи, немає потреби виконувати об'єднання над меншими відношеннями, щоб перевірити, чи не порушено обмеження на вихідне відношення.

Розглянемо поняття функціональної залежності, що є фундаментальним для процесу нормалізації.

Функціональні залежності

Для обговорення функціональних залежностей припустимо, що реляційна схема має атрибути $(A, B, C, \dots, Z)$ і що база даних описується єдиним універсальним відношенням, яке називається $R = (A, B, C, \dots, Z)$. Це припущення означає, що кожен атрибут у базі даних має унікальне ім'я.

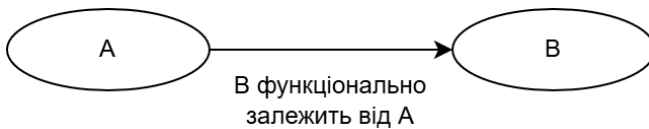


Рис. 2.21. Функціональна залежність між атрибутами у відношенні

Наприклад, якщо A і B – атрибути відношення R , то B є функціонально залежним від A (позначається $A \rightarrow B$), якщо кожному значенню A відповідає рівно одне значення B (A і B можуть складатися з одного або більше атрибутів).

Функціональна залежність – це властивість значення або семантики атрибутів у відношенні. Семантика вказує, як атрибути пов’язані один з одним, і визначає функціональні залежності між атрибутами. Коли присутня функціональна залежність, вона вказується як обмеження між атрибутами.

Розглянемо відношення з атрибутами А та В, де атрибут В функціонально залежить від атрибута А. Якщо ми знаємо значення А і досліджуємо відношення, яке містить цю залежність, ми знайдемо лише одне значення В у всіх кортежах, які мають дане значення А, у будь-який момент часу. Таким чином, коли два кортежі мають однакове значення А, вони також мають однакове значення В. Однак для одного значення В може існувати декілька різних значень А. Залежність між атрибутами А та В можна представити схематично, як показано на рис. 2.21.

Альтернативний спосіб описати залежність між атрибутами А і В – сказати, що «А функціонально визначає В». Іноді перевага віддається саме такому опису, оскільки він більш природно відповідає напрямку стрілки функціональної залежності між атрибутами.

Коли існує функціональна залежність, атрибут або група атрибутів зліва від стрілки називається *детермінантом*. Наприклад, на рис. 2.21, А є детермінантом В. Це буде продемонстровано в наступному прикладі.

Розглянемо атрибути *student_id* і *funding_source* відношення Student на рис. 2.22. Для конкретного *student_id* – наприклад, МК1111 – ми можемо визначити джерело оплати за навчання цього студента як «контракт». Іншими словами, *student_id* функціонально визначає джерело оплати, як показано на рис. 2.22,*а*. Однак рисунок 2.22,*б* ілюструє, що обернене твердження є хибним, оскільки джерело оплати функціонально не визначає *student_id*. Студент має одне джерело

оплати; однак може бути кілька студентів, що мають однакові джерела оплати за навчання.

Для кожного номера студента є лише одне джерело оплати. З іншого боку, існує декілька номерів студентів, пов'язаних з даним джерелом оплати. У цьому прикладі *student_id* є детермінантом цієї функціональної залежності. Для цілей нормалізації нас цікавить виявлення функціональних залежностей між атрибутами відношення, що є визначником у лівій частині, та атрибутом(ами) у правій частині залежності.

Під час визначення функціональних залежностей між атрибутами у відношенні важливо чітко розрізняти значення, яких набуває атрибут у певний момент часу, та множину всіх можливих значень, які атрибут може набувати в різні моменти часу. Іншими словами, функціональна залежність – це властивість схеми відношення, а не властивість конкретного екземпляра схеми, що проілюстровано в наступному прикладі.

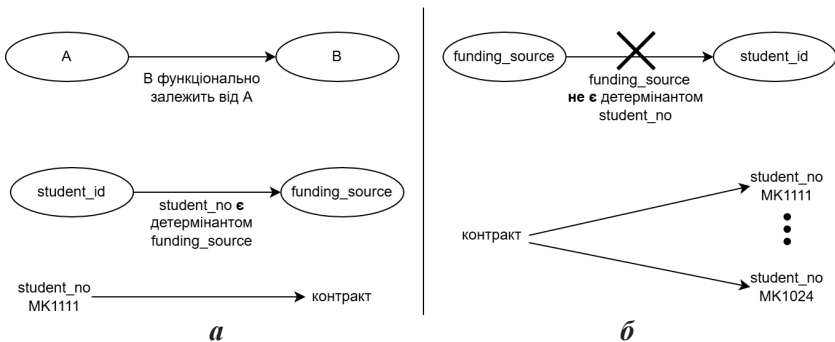


Рис. 2.22. Функціональні залежності

Номер студента функціонально визначає джерело оплати ($student_id \rightarrow funding_source$) (а); джерело оплати функціонально не визначає номер студента ($funding_source \nrightarrow student_id$) (б)

Додатковою характеристикою функціональних залежностей, яка є корисною для нормалізації, є те, що їхні детермінанти повинні мати мінімальну кількість атрибутів, необхідних для підтримки функціональної залежності з атрибутом (атрибутами) у правій частині. Ця вимога називається **повною функціональною залежністю**.

Повна функціональна залежність указує, що якщо A та B є атрибутами відношення, B є повністю функціонально залежним від A , якщо B є функціонально залежним від A , але не від будь-якої власної підмножини A .

Функціональна залежність $A \rightarrow B$ є повною функціональною залежністю, якщо видалення будь-якого атрибуту з A призводить до зникнення залежності. Функціональна залежність $A \rightarrow B$ є **частковою залежністю**, якщо існує якийсь атрибут, який можна видалити з A , але залежність все одно зберігається.

Існує тип функціональної залежності, який називається **транзитивною залежністю**, який ми повинні визнати, оскільки його наявність у відношенні може потенційно спричинити типи аномалій оновлення.

Умова, за якої A , B і C є атрибутами відношення, такі, що, якщо $A \rightarrow B$ і $B \rightarrow C$, то C транзитивно залежить від A через B (за умови, що A не є функціонально залежним від B або C).

Етапи нормалізації

Процес нормалізації включає в себе кілька етапів або нормальних форм, кожна з яких усуває певний тип надмірності або аномалій у даних.

Нормалізація – це формальна техніка аналізу відносин на основі їх первинного ключа (або ключів-кандидатів) та функціональних залежностей. Ця техніка включає в себе низку правил, які можна використовувати для перевірки окремих відносин, щоб база даних могла бути нормалізована

до будь-якого ступеня. Якщо вимога не виконується, відношення, що порушує вимогу, необхідно розкласти на відношення, які окремо відповідають вимогам нормалізації.

Спочатку було запропоновано три нормальні форми, які називалися першою нормальною формою (1NF), другою нормальною формою (2NF) і третьою нормальною формою (3NF). Згодом Р. Бойс і Е. Ф. Кодд запровадили більш суворе визначення третьої нормальної форми, яке називалося нормальною формою Бойса-Кода (BCNF). За винятком 1NF, усі ці нормальні форми базуються на функціональних залежностях між атрибутами відношення. Пізніше були введені вищі нормальні форми, що виходять за межі BCNF, такі як четверта нормальна форма (4NF) і п'ята нормальна форма (5NF). Однак ці пізніші нормальні форми стосуються ситуацій, які трапляються дуже рідко.

Нормалізація часто виконується як серія кроків. Кожен крок відповідає конкретній нормальній формі, яка має відомі властивості. У міру просування нормалізації відносини стають усе більш обмеженими (сильнішими) за форматом, а також менш вразливими до аномалій оновлення. Для реляційної моделі даних важливо усвідомлювати, що лише перша нормальна форма (1NF) є критичною для створення відносин; усі наступні нормальні форми є необов'язковими. Однак, щоб уникнути аномалій оновлення, зазвичай рекомендується переходити принаймні до третьої нормальної форми (3NF).

Перша нормальна форма (1NF)

Перша нормальна форма вимагає від відношення усунення мультизначних атрибутів, тобто кожне значення в таблиці повинно бути атомарним (неподільним). Це досягається шляхом розділення атрибутів на окремі записи.

Друга нормальна форма (2NF)

Відношення знаходиться у другій нормальній формі, якщо воно задовольняє умови першої нормальної форми та всі неключові атрибути функціонально залежать від

усього ключа. Це означає, що часткова залежність від ключа повинна бути усунута.

Третя нормальна форма (3НФ)

Щоб досягти третьої нормальної форми, відношення повинно знаходитись у другій нормальній формі та всі неключові атрибути повинні бути залежними тільки від суперключа, а не від інших неключових атрибутів. Це усуває транзитивні залежності.

Нормальна форма Бойса-Кодда (BCNF)

Нормальна форма Бойса-Кодда є посиленням третьої нормальної форми. Вона вимагає, щоб кожна функціональна залежність у відношенні була тривіальною або ліва частина залежності була суперключем.

Четверта нормальна форма (4НФ)

Четверта нормальна форма усуває багатозначні залежності. Відношення знаходиться у четвертій нормальній формі, якщо воно задовольняє умови Boyce-Codd нормальної форми та не має багатозначних залежностей.

П'ята нормальна форма (5НФ)

П'ята нормальна форма усуває надмірності, що виникають через багатозначні залежності. Вона досягається шляхом розбиття відношень на менші відношення таким чином, щоб зберігалася властивість «без втрат».

Використання нормалізації вимагає, щоб ми спочатку визначили функціональні залежності, що існують між атрибутами в кожному відношенні. Характеристики функціональних залежностей, що використовуються для нормалізації, які будуть розглянуті в наступному розділі, можуть бути визначені тільки в тому випадку, якщо добре зрозуміле значення кожного атрибуту. Функціональні залежності вказують на важливі зв'язки між атрибутами відношення. Саме ці функціональні залежності та первинний ключ для кожного відношення використовуються в процесі нормалізації.

Процес нормалізації передбачає виконання низки кроків для перевірки відповідності складу атрибутів у відношенні правилам певної нормальної форми, такої як 1NF, 2NF та 3NF. Щоб уникнути проблем, пов'язаних із надмірністю даних, рекомендується, щоб кожне відношення відповідало принаймні 3NF.

Процес виведення відношень з концептуальної моделі даних повинен давати логічну модель даних, яка складається з відношень, що вже відповідають 3NF. Однак, якщо ми виявляємо відношення, які не відповідають 3NF, це може свідчити про те, що частина логічної моделі даних та/або концептуальної моделі даних є неправильною, або що ми допустили помилку під час виведення відношень з концептуальної моделі даних. За потреби ми повинні реструктурувати проблемні відношення та/або моделі даних, щоб забезпечити правдиве відображення вимог предметної області. У процесі нормалізації важливо враховувати специфіку предметної області та вимоги до цілісності й ефективності зберігання даних. Як каже українське прислів'я: «Сім разів відмірай – один раз відріж», – тобто краще ретельно перевірити структуру відношень і функціональні залежності, ніж потім виправляти помилки в базі даних, які можуть призвести до втрати або дублювання інформації. Вдале визначення функціональних залежностей полегшує подальше масштабування та супровід інформаційної системи, а також забезпечує коректність виконання запитів і бізнес-логіки.

Після завершення побудови логічної моделі даних та узгодження її із замовником, якщо це потрібно, розробники можуть переходити до наступного етапу – побудування фізичної моделі даних. На цьому етапі обов'язково повинні бути враховані вимоги щодо масштабованості системи, засобів розгортання й обрана відповідна СУБД. Цей етап тісно пов'язаний із реалізацією застосунку із використанням однієї

із сучасних мов програмування та властивих для цієї мови бібліотек і фреймворків щодо роботи з даними. У наступному розділі посібника буде розглянуто процес реалізації такого застосунку на платформі Java.

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Опишіть три основні рівні архітектури БД.
2. Охарактеризуйте основні етапи проєктування БД (концептуальний, логічний, фізичний).
3. У чому полягає сутність та основна мета моделювання даних?
4. Назвіть ключові критерії вибору СУБД для проєкту.
5. Опишіть типові етапи процесу вибору СУБД.
6. Чим відрізняються реляційні та нереляційні (NoSQL) СУБД?
7. Які основні компоненти входять у проєктування застосунків?
8. Що таке реалізація БД і які інструменти (наприклад, SQL DDL) для цього використовуються?
9. Які складності виникають під час перетворення та завантаження даних?
10. Які основні сутності та зв'язки між ними були визначені в прикладі проєкту?
11. Назвіть дві основні стратегії проєктування БД («знизу-вгору» та «зверху-вниз») та поясніть їх відмінність.
12. Що таке сутність, атрибут, зв'язок в ER-моделі?
13. Які типи зв'язків існують між сутностями?
14. Що таке ідентифікатор (первинний ключ) сутності?
15. Яка мета побудови концептуальної моделі даних?
16. Яка основна мета нормалізації бази даних?
17. Дайте визначення першої нормальної форми (1НФ). Що таке атомарність значень?

18. За яких умов відношення знаходиться у другій нормальній формі (2НФ)?
19. Що таке транзитивна залежність і як її усунути для досягнення третьої нормальної форми (3НФ)?
20. У чому полягає сутність нормальної форми Бойса-Кодда (BCNF) та чим вона відрізняється від 3НФ?

РОЗДІЛ 3

РЕАЛІЗАЦІЯ ЗАСТОСУНКУ З БД НА ПЛАТФОРМІ JAVA

3.1. Аналіз специфікації Java Persistence API

Після аналізу підходів проєктування БД, розглянемо підходи до створення застосунків на платформі Java (мовами Java, Kotlin та ін.). Проаналізуємо основні й розширені функції та опишемо основні способи розробки нових застосунків із використанням Java Persistence API. Часто ці рекомендації не стосуються конкретно фреймворків Hibernate або Spring Data. Іноді це ідеї досвідчених розробників щодо найкращих способів роботи з постійними даними, пояснені в контексті Hibernate та Spring Data. Вибір підходу до управління постійними даними може бути ключовим рішенням під час проєктування багатьох програмних проєктів. Проблема Persistence завжди була гарячою темою для обговорення у Java-спільноті та взагалі у сегменті розробки корпоративних застосунків. Чи є Persistence проблемою, яка вже вирішена за допомогою SQL та розширень, таких як збережені процедури, чи це більш поширена проблема, яку необхідно вирішувати за допомогою спеціальних Java-фреймворків? Чи варто вручну кодувати навіть найпростіші операції CRUD (створення, читання, оновлення, видалення) у SQL і JDBC, чи слід цю роботу передати проміжному рівню? Яким чином можна досягти переносимості (portability), якщо кожна система управління базами даних має свій власний діалект SQL? Чи не слід повністю відмовитися від SQL і перейти на іншу технологію баз даних, таку як об'єктні системи баз даних або системи NoSQL? Ця дискусія, можливо, ніколи

не закінчиться, але на сучасному етапі розуміння концепцій розробки корпоративних застосунків рішення під назвою «об'єктно-реляційне відображення» (ORM) є найбільш поширеним. Це сталося значною мірою завдяки Hibernate, реалізації служби ORM з відкритим кодом, та фреймворку Spring Data, загальному проєкту з родини Spring, метою якого є уніфікація та спрощення доступу до різних типів сховищ даних, включаючи реляційні системи баз даних та бази даних NoSQL.

Однак перш ніж розглянути засоби роботи з Hibernate та Spring Data, необхідно зрозуміти основні проблеми збереження об'єктів та ORM [7].

Для більшого розуміння наступних понять і концепцій зауважимо, що імена таблиць БД, їхніх полів та інших об'єктів «зі світу БД» будемо записувати у одному (зазвичай – верхньому) регістрі, а у випадку, коли ім'я складається з декількох слів, розділяємо їх за допомогою символу «_» (підкреслення). Іноді такий стиль називають «SNAKE_CASE». У той же час імена класів, що є відповідними до цих таблиць, будемо записувати згідно з правилами іменування Java – перша літера та кожна літера на початку нового слова в імені – у верхньому регістрі, інші символи – у нижньому регістрі. Такий стиль часто називають «CamelCase».

Наприклад, STUDENT та SCORE – імена таблиць. Student та Score – імена персистентних класів, що відповідні до цих таблиць. Зауважимо, що більшість діалектів SQL сучасних СУБД не є чутливими до регістру літер в іменах, тому під час проєктування БД, щоб не створювати плутанини, краще все писати в одному регістрі.

Спочатку буде визначено поняття управління постійними даними в контексті програмних застосунків і обговоримо взаємозв'язки між SQL, JDBC і Java, базовими технологіями та стандартами, на яких базуються Hibernate і Spring Data.

Потім розглянемо так звану невідповідність об'єктно-реляційної парадигми та загальні проблеми, з якими розробники стикаються в об'єктно-орієнтованій розробці програмного забезпечення з базами даних SQL. Ці проблеми дають зрозуміти, які потрібні інструменти та шаблони, щоб мінімізувати час, що витрачається на код, пов'язаний із збереженням даних у застосунках, що розробляються.

Для більшості програм потрібні постійні дані. Персистентність (Persistence) є одним з основоположних понять у розробці застосунків. Якщо інформаційна система не зберігає дані при вимкненому живленні, то практичної користі від неї буде мало. Персистентність об'єктів означає, що окремі об'єкти можуть зберегтися навіть при завершенні програми, – вони можуть бути збережені у сховищі даних і знову створені в більш пізній момент часу. Коли розробники говорять про персистентність у Java, зазвичай мається на увазі відображення та зберігання екземплярів об'єктів у базі даних за допомогою SQL. З початку буде зроблено короткий огляд персистентності і того, як вона використовується у Java.

У попередніх розділах посібника були розглянуті підходи до проектування реляційних баз даних. Більшість розробників щодня стикаються з такими системами. Реляційні системи управління базами даних мають інтерфейси прикладного програмування на основі SQL, тому іноді сьогоднішні продукти реляційних баз даних SQL називають *системами управління базами даних* (СУБД) або, коли мова йде про конкретні системи, базами даних SQL.

Реляційна технологія є добре відомою технологією, і вже одне це є достатньою причиною для багатьох організацій вибрати її. Реляційні бази даних також є неймовірно гнучким і надійним підходом до управління даними. Завдяки добре вивченій теоретичній базі реляційної моделі

даних, реляційні бази даних можуть гарантувати й захищати цілісність даних, що зберігаються, поряд з іншими бажаними характеристиками. Основні поняття реляційних моделей були розглянуті раніше.

Щоб ефективно використовувати JPA, Hibernate та Spring Data для розробки прикладних програм потрібно добре розуміти реляційну та інформаційну модель, а також нормалізацію відношень, щоб гарантувати цілісність ваших даних, також вам потрібно буде використовувати знання SQL для налаштування продуктивності вашого застосунку. Питання проєктування моделей даних та нормалізації відношень розглядались у другому розділі. Мова SQL сама по собі є окремою темою вивчення і виходить за межі цього посібника. Звісно, окремі конструкції запитів SQL розглядатимуться далі у посібнику, але для більш докладного вивчення SQL рекомендується звернутись до [5].

Hibernate і Spring Data спрощують багато повторюваних завдань кодування, але знання технології персистентності повинні виходити за рамки самих фреймворків, якщо розробнику треба скористатися всіма можливостями сучасних баз даних SQL.

Коли Java-розробник працює з базою даних SQL у програмі Java, він пише код, що надсилає SQL-запити до бази даних через API Java DataBase Connectivity (JDBC). Незалежно від того, чи був SQL написаний уручну та вбудований у код Java чи згенерований на льоту за допомогою коду Java, буде використовуватись JDBC API для зв'язування аргументів під час підготовки параметрів запиту, виконання запиту, прокручування результатів запиту, отримання значень із набору результатів тощо. Це завдання низькорівневого доступу до даних. Але з точки зору розробників інформаційних систем, є більша зацікавленість у самій бізнес-проблемі, яка вимагає такого доступу до даних.

Те, що розробник дійсно хоче написати, це код, який зберігає і відновлює екземпляри класів застосунку, звільняючи його від цієї низькорівневої праці.

Є багато причин, чому бази даних SQL домінують в обчислювальній індустрії – реляційні системи управління базами даних є єдиною перевіреною загальною технологією управління даними, і вони майже завжди є вимогою в проєктах на Java.

У посібнику розглядаються проблеми зберігання та спільного використання даних у контексті об'єктно-орієнтованого застосунку, що використовує модель предметної області. Замість того, щоб безпосередньо працювати з рядками та стовпцями набору даних `java.sql.ResultSet`, бізнес-логіка програми буде взаємодіяти з об'єктно-орієнтованою моделлю домену конкретного додатку. Якщо у схемі бази даних SQL системи університету є, наприклад, таблиці `STUDENT` і `SCORE`, то застосунок Java визначає відповідні класи `Student` і `Score`. Замість того, щоб зчитувати й записувати значення конкретного рядка і стовпця за допомогою `API ResultSet`, застосунок завантажує та зберігає екземпляри класів `Student` і `Score`.

Тому під час виконання застосунків працює з екземплярами цих класів. Кожен екземпляр `Score` має посилання на `Student`, а кожен `Student` може мати колекцію посилань на екземпляри `Score`. Бізнес-логіка не виконується в базі даних (як процедура, що зберігається в SQL), вона реалізована на Java й виконується на рівні застосунків.

Це дозволяє бізнес-логіці використовувати складні об'єктно-орієнтовані поняття, такі як успадкування і поліморфізм. Наприклад, з'являється можливість використовувати добре відомі шаблони проєктування, такі як `Strategy`, `Mediator` і `Composite` [6], які залежать від викликів поліморфних методів.

Зауважимо, що не всі Java-застосунки працюють таким чином, та вони і не повинні обов'язково працювати саме так. Прості програми можуть бути набагато кращими без доменної моделі. Можна використовувати JDBC ResultSet, якщо це все, що потрібно. Можна викликати існуючі збережені процедури та прочитати їхні набори результатів SQL. Багатьом програмам потрібно виконувати процедури, які змінюють великі набори даних, близькі до даних. Розробник також може реалізувати деякі функції створення звітів за допомогою простих SQL-запитів і відображати результати безпосередньо на екрані. SQL та API JDBC ідеально підходять для роботи з табличним представленням даних, а JDBC RowSet робить операції CRUD ще простішими. Робота з таким представленням постійних даних проста й добре зрозуміла.

Але для додатків з нетривіальною бізнес-логікою перехід до доменної моделі допомагає значно покращити повторне використання коду та його супроводжуваність. На практиці обидві стратегії є загальними й необхідними.

Протягом кількох десятиліть розробники говорили про невідповідність парадигм. Ідеться про об'єктне моделювання та реляційне моделювання, або, більш практично, об'єктно-орієнтоване програмування та SQL. Ця невідповідність пояснює, чому кожен корпоративний проєкт витрачає стільки зусиль на проблеми, пов'язані з постійністю. За допомогою цієї концепції ви можете почати бачити проблеми – деякі добре зрозумілі, деякі менш вивчені – які повинні бути вирішені в додатку, що поєднує в собі об'єктно-орієнтовану модель предметної області й постійну реляційну модель. Розглянемо докладніше цю так звану невідповідність парадигм.

Невідповідність парадигми

Невідповідність об'єктно-орієнтованої та реляційної парадигм можна розбити на кілька частин, але ми розглянемо лише основні. Згадаємо приклад, бази даних, створеної

в попередньому розділі, але, для початку, розглянемо лише дві таблиці.

Розглянемо взаємозв'язок між Викладачем та Кафедрою, де він працює. У такому випадку потрібен клас для представлення інформації про Кафедру, а інший клас потрібен для представлення інформації про Викладача, як показано на рис. 3.1.

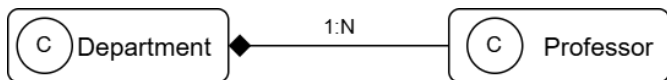


Рис. 3.1. Зв'язок між викладачем та кафедрою

На цій діаграмі ви можете побачити, що Кафедра має у штаті багато Викладачів. Це є композицією, що позначена зафарбованим ромбом. *Композиція* – це тип асоціації, при якій об'єкт (у нашому випадку Викладач) не може концептуально існувати без контейнера (у нашому випадку Кафедра). У реальному житті, звісно, викладач може тимчасово не бути «прив'язаним» до кафедри, але в рамках інформаційної системи університету ми вважаємо, що викладач завжди працює на деякій кафедрі. Орієнтуватися у взаєминах між класами можна в обох напрямках. Це означає, що ви можете перебирати колекції або викликати методи, щоб дістатися до «іншої» сторони відносин. Класи, що представляють ці сутності, можуть бути дуже простими:

```

public class Department {
    private int department_id;
    private String name;
    private String head;
    private Set<Professor> professors = new LinkedHashSet<>();
}

public class Professor {
    private int professor_id;
    private String name;
}
  
```

```
}  
    private String degree;  
    private Department department;  
}
```

Зверніть увагу, що нас цікавить лише стан сутностей, що зберігається, тому ми опустили реалізацію конструкторів, методів доступу та бізнес-методів.

Для цього випадку легко придумати дизайн SQL-схеми (синтаксис наступних запитів відповідає діалекту H2):

```
CREATE TABLE DEPARTMENT (  
    ID INTEGER NOT NULL PRIMARY KEY,  
    NAME VARCHAR(100) NOT NULL,  
    HEAD VARCHAR(150) NOT NULL  
);  
  
CREATE TABLE PROFESSOR (  
    ID INTEGER NOT NULL PRIMARY KEY,  
    NAME VARCHAR(50) NOT NULL,  
    DEGREE VARCHAR(100) NOT NULL,  
    DEPT_ID VARCHAR(100) NOT NULL,  
    FOREIGN KEY (DEPT_ID) REFERENCES DEPARTMENT(ID)  
);
```

Як легко побачити, таблиця DEPARTMENT (Кафедри) не містить інформації про викладачів, тому, щоб знайти всіх викладачів, які працюють на вказаній кафедрі, треба буде переглянути повністю всю таблицю PROFESSOR (Викладачі). У той же час, у Java-кодi, клас Department містить колекцію (Set) усіх викладачів.

Проблема асоціацій

Об'єктно-орієнтовані мови представляють асоціації за допомогою посилань на об'єкти, але в реляційному світі стовпець з обмеженням зовнішнього ключа представляє асоціацію з копіями значень ключів. *Обмеження* (Constraint) – це правило, яке гарантує цілісність об'єднання. Між цими двома механізмами є суттєві відмінності. Посилання на

об'єкти за своєю суттю є орієнтованими. Зв'язок відбувається від одного екземпляра до іншого. Вони є посиланнями. Якщо асоціація між екземплярами повинна бути навігаційною в обох напрямках, то необхідно визначити асоціацію двічі, по одному разу в кожному з асоційованих класів.

Проблема навігації по даних

Існує принципова різниця між тим, як розробник отримує доступ до даних у коді Java та в реляційній базі даних. У Java, коли треба отримати доступ до списку викладачів кафедри, Java-розробник пише код, подібний до такого:

```
someDepartment.getProfessors().iterator().next().
```

Або, починаючи з Java 8, це може виглядати таким чином:

```
someDepartment.getProfessors().stream()  
    .filter(someCondition)  
    .map(someMapping)  
    .forEach(p → { doSomething(p) } ).
```

Це найприродніший спосіб доступу до об'єктно-орієнтованих даних, і його часто описують як *ходіння по об'єктній мережі*. Програма виконує переходи з одного екземпляра до наступного. Звісно, так можна проходити по колекціях, слідуючи заздалегідь підготовленим посиланням між об'єктами. На жаль, це не найбільш ефективний спосіб отримання даних із бази даних SQL.

Єдина найважливіша річ, яку завжди можна зробити для підвищення продуктивності коду доступу до даних, це мінімізувати кількість запитів до бази даних. Найочевидніший спосіб зробити це – мінімізувати кількість SQL-запитів. (Звичайно, інші, більш складні способи, такі як широке кешування, ідуть як другий крок.)

Тому ефективний доступ до реляційних даних за допомогою SQL зазвичай вимагає об'єднання (join) між таблицями.

Кількість таблиць, що входять в об'єднання під час отримання даних, визначає глибину об'єктної мережі, по якій можна переміщуватися в пам'яті. Наприклад, якщо розробнику потрібно отримати кафедру, але його не цікавлять її викладачі, можна написати такий достатньо простий запит:

```
SELECT * FROM DEPARTMENT WHERE ID = 123
```

З іншого боку, якщо потрібно отримати кафедру, а потім переглянути кожного із пов'язаних з нею викладачів (скажімо, щоб дізнатись їхню освіту), необхідно написати інший, більш складний запит:

```
SELECT * FROM DEPARTMENT, PROFESSOR  
WHERE DEPARTMENT.ID = 123 AND  
DEPT_ID = DEPARTMENT.ID
```

Легко побачити, що для ефективного використання об'єднань, розробнику потрібно знати, до якої частини об'єктної мережі він планує отримати доступ. І це треба зробити перш, ніж він уже буде орієнтуватися в об'єктній мережі. Однак треба бути обережними: якщо запит у програмі отримає занадто багато даних (імовірно, більше ніж може знадобитися), виникнуть зайві витрати пам'яті на рівні програми. Крім того, це може перевантажити базу даних SQL величезними наборами результатів декартових добутоків. Уявіть, що програма отримує не тільки списки кафедр та імена їхніх викладачів в одному запиті, але і всі дані викладача (можливо, місто, адреса і т. д.).

Будь-яке рішення для збереження об'єктів дозволяє отримувати дані пов'язаних екземплярів лише тоді, коли асоціація вперше звертається до неї в коді Java. Це відомо як ліниве завантаження (*lazy loading*): отримання даних лише на вимогу. Такий фрагментарний стиль доступу до даних

принципово неефективний у контексті бази даних SQL, тому що вимагає виконання одного оператора для кожного вузла або колекції об'єктної мережі, до якої здійснюється доступ. Це проблема «select n+1». У нашому прикладі вам знадобиться один select, щоб отримати екземпляр Department, а потім n операторів select для кожного з n пов'язаних екземплярів Professor.

Ця невідповідність у способі доступу до даних у кодї Java та в реляційній базі даних є, мабуть, єдиним найпоширенішим джерелом проблем із продуктивністю в інформаційних системах Java. Уникнення задач декартового добутку та «select n+1» усе ще залишається проблемою для багатьох програмістів Java. Hibernate надає складні функції для ефективного та прозорого отримання мереж об'єктів з бази даних до програми, що отримує доступ до них.

3.2. Вибір фреймворку: ORM, JPA, Hibernate та Spring Data

Наведемо коротке визначення. *Об'єктно-реляційне відображення* (ORM) – це автоматизоване (і прозоре) збереження об'єктів у застосунку Java до таблиць у РСУБД (реляційній системі управління базами даних) з використанням метаданих, які описують відповідність між класами програми та схемою бази даних SQL. По суті, ORM працює шляхом перетворення (у обох напрямках) даних з одного представлення в інше. Програма, що використовує ORM, надає мета-інформацію про те, як відображати об'єкти з пам'яті в базу даних, а ефективне перетворення буде виконано ORM.

Існує думка про те, що однією з переваг ORM є те, що він захищає розробників від безладного SQL. Ця точка зору стверджує, що від об'єктно-орієнтованих розробників не слід очікувати заглиблення в SQL або реляційні бази даних. Навпаки, розробники Java повинні мати достатній рівень

знайомства з реляційним моделюванням та SQL, щоб працювати з Hibernate та Spring Data. *ORM* – це «просунута» техніка, яка використовується розробниками, які вже зробили це непростим шляхом.

JPA (Jakarta Persistence API, раніше Java Persistence API) – це специфікація, що визначає API, який управляє постійністю об'єктів та відображенням об'єктів та відношень. Hibernate є найпопулярнішою реалізацією цієї специфікації. Так, JPA вказує, що потрібно зробити, щоб об'єкти зберігалися, а Hibernate визначає, як це зробити. Spring Data Commons, як частина сімейства Spring Data, надає основні концепції SpringFramework, які підтримують усі модулі Spring Data. Spring Data JPA – один з проєктів сімейства Spring Data, який є додатковим шаром поверх реалізацій JPA (таких як Hibernate). Spring Data JPA не тільки може використовувати всі можливості JPA, але й додає власні можливості, такі як генерація запитів до бази даних з імен методів.

Щоб ефективно використовувати Hibernate, розробники повинні вміти переглядати та інтерпретувати SQL-вирази, які він видає, і розуміти їх наслідки для продуктивності. Щоб скористатися перевагами Spring Data, треба бути у змозі передбачити, як створюється шаблонний код і як генеруються запити.

Специфікація JPA визначає наступне:

- Можливість для визначення метаданих відображення – наскільки персистентні класи та їх властивості пов'язані зі схемою бази даних. JPA значною мірою покладесться на анотації Java у класах моделей домену, але ви також можете записувати відображення у XML-файлах.
- API для виконання базових операцій CRUD над екземплярами персистентних класів, найбільш помітним є `jakarta.persistence.EntityManager` для зберігання та завантаження даних.

- Мова та API для задання запитів, які посилаються на класи та властивості класів. Ця мова називається Jakarta Persistence Query Language (JPQL) і зовні схожа на SQL. Стандартизований API дозволяє програмно створювати запити критеріїв без маніпуляцій із рядками.

- Як механізм персистентності взаємодіє з транзакційними екземплярами для отримання асоціацій та інших функцій оптимізації. Специфікація JPA охоплює деякі основні стратегії кешування.

Hibernate є однією з найпоширеніших реалізацій JPA і підтримує всі стандартизовані відображення, запити та програмні інтерфейси. Розглянемо деякі переваги Hibernate:

- **Швидкість.** Hibernate усуває більшу частину повторюваної роботи (більше ніж ви очікували) і дозволяє вам зосередитися на бізнес-проблемі. Незалежно від того, якій стратегії розробки додатків ви віддаєте перевагу – зверху-вниз (починаючи з моделі домену) або знизу-вгору (починаючи з існуючої схеми бази даних) – Hibernate, що використовується разом з відповідними інструментами, значно скоротить час розробки.

- **Супроводжуваність.** Автоматизований ORM з Hibernate зменшує кількість рядків коду, роблячи систему більш зрозумілою та простішою для рефакторингу. Hibernate забезпечує буфер між моделлю домену та SQL-схемою, ізолюючи кожну модель від незначних змін в іншій.

- **Продуктивність.** Хоча продуктивність ручного кодування може бути швидшою в тому ж сенсі, що код асемблера може бути швидшим, ніж код Java, автоматизовані рішення, такі як Hibernate, дозволяють використовувати багато оптимізацій у будь-який час. Одним із прикладів є ефективне та легке у налаштуванні кешування даних на рівні програми. Це означає, що розробники можуть витратити більше енергії на ручну оптимізацію кількох

реальних вузьких місць, що залишилися, замість того, щоб передчасно оптимізувати все.

- **Незалежність постачальника.** Hibernate може допомогти зменшити деякі ризики, пов'язані з прив'язкою до постачальника. Навіть якщо ви плануєте ніколи не змінювати свій продукт СУБД, інструменти ORM, які підтримують кілька різних СУБД, забезпечують певний рівень портативності. Крім того, незалежність СУБД допомагає у сценаріях розробки, коли інженери використовують легку локальну базу даних, але розгортають для тестування та виробництва на іншій системі.

Spring Data робить реалізацію шару персистентності ще більш ефективною. Spring Data JPA, один з проєктів сімейства Spring Data, використовує можливості JPA. Spring Data JDBC, ще один проєкт сімейства, побудований навколо JDBC. Розглянемо деякі з переваг Spring Data:

- **Спільна інфраструктура.** Spring Data Commons – частина проєкту Spring Data, надає модель метаданих для збережених класів Java та технологічно нейтральних інтерфейсів репозиторіїв. Він надає свої можливості іншим проєктам Spring Data.

- **Видаляє реалізації DAO.** У реалізаціях JPA використовується *шаблон об'єкта доступу до даних* (DAO). Цей шаблон починається з ідеї абстрактного інтерфейсу до бази даних і відображає виклики застосунків на рівень персистентності, приховуючи деталі бази даних. Spring Data JPA дає можливість повністю видалити реалізації DAO, тому код буде коротшим.

- **Автоматичне створення класів.** Використовуючи Spring Data JPA, інтерфейс DAO повинен розширити специфічний для JPA інтерфейс репозиторію – JpaRepository. Spring Data JPA автоматично створить реалізацію для цього інтерфейсу – програмісту не доведеться про це дбати.

- **Реалізації за замовчуванням для методів.** Spring Data JPA генеруватиме реалізації за замовчуванням для кожного методу, визначеного інтерфейсами його репозиторію. Базові операції CRUD більше не потрібно реалізовувати. Це зменшує шаблонний код, прискорює розробку та усуває можливість появи помилок.

- **Згенеровані запити.** Ви можете визначити метод в інтерфейсі вашого репозиторію за шаблоном іменування. Немає потреби писати свої запити вручну. Spring Data JPA розбере ім'я методу та створить для нього запит. Такі методи отримали назву «Похідні методи» (Derived methods).

- **За потреби наближається до бази даних.** Spring Data JDBC може зв'язуватися безпосередньо з базою даних і уникати «магії» Spring Data JPA. Він дозволяє взаємодіяти з базою даних через JDBC, але видаляє шаблонний код за допомогою можливостей фреймворку Spring.

Розглянемо ці технології докладніше.

Hibernate

Як уже було написано раніше, *об'єктно-реляційне відображення* (ORM) – це техніка програмування для встановлення зв'язків між несумісними світами об'єктно-орієнтованих систем та реляційних баз даних. *Hibernate* – це великий проєкт, який має на меті надати повне рішення проблеми управління постійними даними на Java. Сьогодні *Hibernate* – це не тільки сервіс ORM, але й набір інструментів для управління даними, що виходять далеко за межі ORM [7]. Пакет проєктів *Hibernate* включає в себе наступне:

Hibernate ORM. *Hibernate ORM* складається з ядра, базової служби для роботи з базами даних SQL і власного пропрієтарного API. *Hibernate ORM* є основою для кількох інших проєктів у пакеті, і це найстаріший проєкт *Hibernate*. Розробники можуть використовувати *Hibernate ORM* окремо,

незалежно від будь-якого фреймворку або будь-якого конкретного середовища виконання з усіма JDK. Поки джерело даних доступне, існує можливість налаштувати його для Hibernate, і це працює.

Hibernate EntityManager. Це реалізація Hibernate стандартного Jakarta Persistence API. Це необов'язковий модуль, який можна додатково розмістити поверх Hibernate ORM. Рідні функції Hibernate є надмножиною функцій JPA.

Hibernate Validator. Hibernate надає еталонну реалізацію специфікації Bean Validation (JSR 303). Незалежно від інших проєктів Hibernate, він забезпечує декларативну валідацію для класів доменної моделі (або будь-яких інших).

Hibernate Envers. Envers призначений для аудиту, ведення журналу та зберігання кількох версій даних у базі даних SQL. Це допомагає додавати історію даних та контрольні журнали до програми, подібно до будь-яких систем контролю версій, з якими ви, можливо, уже знайомі, таких як Subversion або Git.

Hibernate Search. Hibernate Search підтримує в актуальному стані індекс даних моделі домену в базі даних Apache Lucene. Це дозволяє робити запити до цієї бази даних за допомогою потужного та природно інтегрованого API. Багато проєктів використовують Hibernate Search на додаток до Hibernate ORM, додаючи можливості повнотекстового пошуку. Якщо у вас в інтерфейсі користувача програми є форма вільного текстового пошуку, і ви хочете задоволених користувачів, використовуйте Hibernate Search. Hibernate Search не описаний у посібнику, але ви можете отримати інформацію про його використання з книги [8].

Hibernate OGM. Цей проєкт Hibernate забезпечує підтримку JPA для NoSQL-рішень, повторно використовуючи движок ядра Hibernate, але зберігаючи відображені сутності у сховищах даних, орієнтованих на ключі / значенні, документах або графах.

Hibernate Reactive. Це реактивний API для Hibernate ORM, який взаємодіє з базою даних у неблокуючий спосіб. Він підтримує неблокуючі драйвери баз даних. Hibernate Reactive в даному посібнику не розглядається.

Варто зауважити, що вихідний код Hibernate можна вільно завантажити з <https://github.com/hibernate>.

Spring Data

Spring Data – це сімейство проєктів, що належать до фреймворку Spring, метою яких є спрощення доступу як до реляційних, так і до NoSQL баз даних:

- **Spring Data Commons** – це частина проєкту Spring Data, яка надає модель метаданих для збережених класів Java та технологічно нейтральних інтерфейсів репозиторіїв.

- **Spring Data JPA** займається впровадженням репозиторіїв на основі JPA. Він забезпечує покращену підтримку рівнів доступу до даних на основі JPA шляхом зменшення шаблонного коду та створення реалізацій для інтерфейсів репозиторіїв.

- **Spring Data JDBC** займається впровадженням репозиторіїв на основі JDBC. Він забезпечує покращену підтримку рівнів доступу до даних на основі JDBC. Він не пропонує низку можливостей JPA, таких як кешування або відкладене завантаження, що призводить до простішого та обмеженого ORM.

- **Spring Data REST** займається експортом репозиторіїв Spring Data як RESTful ресурсів.

- **Spring Data MongoDB** займається доступом до бази даних документів MongoDB. Він спирається на рівень доступу до даних у стилі репозиторію та модель програмування POJO.

- **Spring Data Redis** займається доступом до бази даних ключів / значень Redis. Він ґрунтується на звільненні

розробника від управління інфраструктурою та надання високорівневих і низькорівневих абстракцій для доступу до сховища даних. Spring Data Redis та Spring Data MongoDB у посібнику не розглядаються.

Вихідний код Spring Data (разом з іншими проєктами Spring) можна вільно завантажити з <https://github.com/spring-projects>.

3.3. Прості проєкти з використанням JPA та Spring Data

Простий проєкт JPA

Для розроблення простого проєкту, що продемонструє ключові моменти роботи з JPA спочатку створимо порожній Java-проєкт на основі Maven. Початковий файл `pom.xml`, що описує основні властивості проєкту та підключає необхідні бібліотеки (dependencies) показаний нижче. Зверніть увагу, що на момент, коли Ви будете працювати з посібником, версії основних бібліотек можуть змінитись. Можна навіть із впевненістю сказати – зміняться. Урахуйте це під час створення свого проєкту.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ua.edu.nuos</groupId>
  <artifactId>dbdemo1</artifactId>
  <version>1.0-SNAPSHOT</version>
  <properties>
    <maven.compiler.source>21</maven.compiler.source>
    <maven.compiler.target>21</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <version>2.3.232</version>
```

```
</dependency>
<dependency>
  <groupId>org.hibernate.orm</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>7.0.4.Final</version>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.12.2</version>
  <scope>test</scope>
</dependency>
</dependencies>
</project>
```

Модуль `hibernate-core` включає в себе транзитивні залежності від інших модулів, які нам знадобляться та заглушки інтерфейсу JPA. Нам також потрібна залежність `junit-jupiter-engine` для запуску тестів за допомогою JUnit 5, а також залежність `h2`, яка є офіційним драйвером JDBC для H2.

Також для роботи нам знадобиться проста БД. Яку можна побачити із файлу `pom.xml`, для першої ознайомчої програми ми використаємо просту СУБД H2, як було рекомендовано раніше. Створимо БД із використанням H2 засобами IntelliJ IDEA. Для цього у вікні Database треба натиснути знак `+` і вибрати тип джерела даних – H2. У вікні, що відкриється, потрібно перевірити, чи співпадає версія СУБД з версією підключеного у `pom.xml` драйвера. У разі не співпадіння, варто оновити драйвер у IntelliJ IDEA. Інші параметри підключення можна заповнити власними значеннями (або скористатися зразком на рис. 3.2).

Після створення БД, можна додати до неї таблицю. Для того, під іменем бази знайдіть розділ PUBLIC та клацніть по ньому правою кнопкою миші. У меню, що відкриється, оберіть «`+ New`» і далі – Table (рис. 3.3).

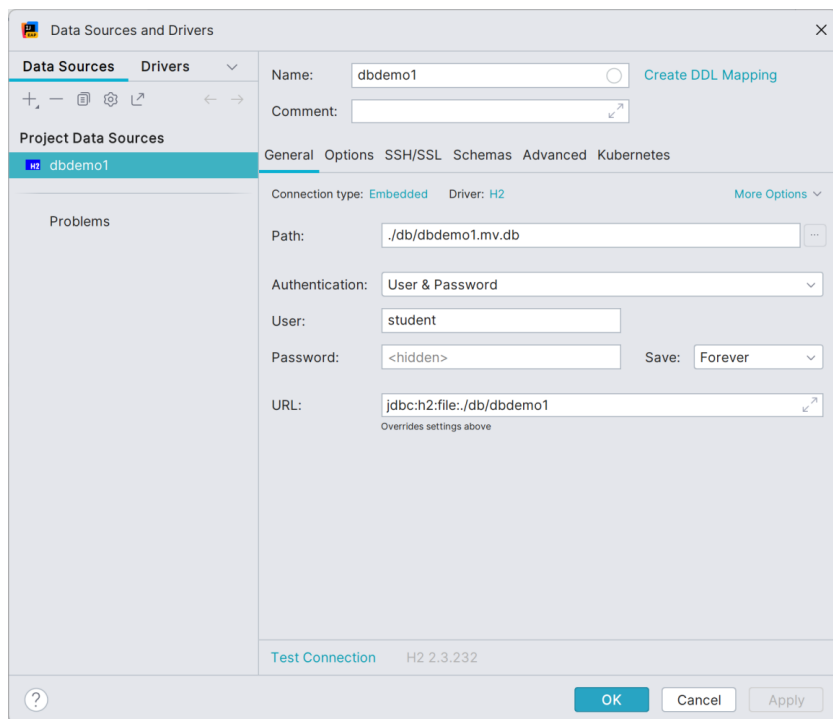


Рис. 3.2. Створення БД у СУБД H2 із використанням IntelliJ IDEA

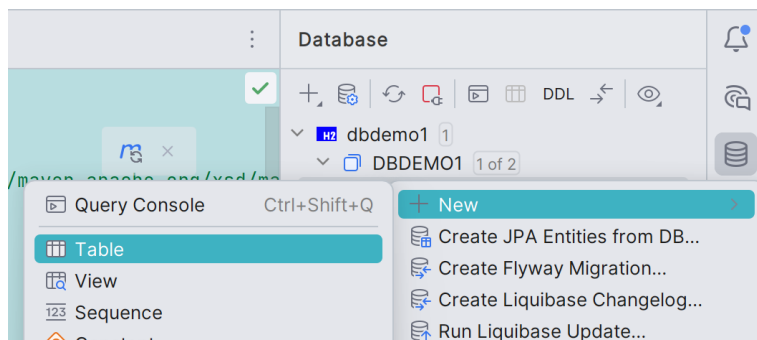


Рис. 3.3. Створення таблиці у БД

Після того, заповніть дані таблиці CITY з двома колонками: ID та NAME відповідно до рис. 3.4.

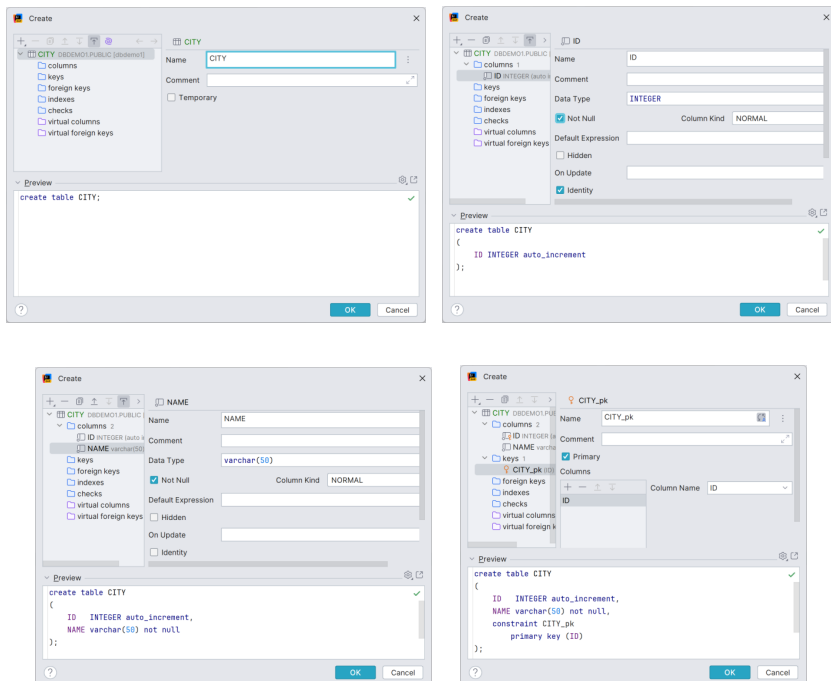


Рис. 3.4. Кроки створення таблиці

Звісно, також можна створити таблицю, виконавши у консолі БД IntelliJ IDEA скрипт DDL:

```
create table CITY
(
    ID    INTEGER auto_increment,
    NAME  varchar(50) not null,
    constraint CITY_pk
        primary key (ID)
);
```

Після того, як таблиця у БД створена, продовжимо створення застосунку.

Наша відправна точка в JPA – це одиниця персистентності (persistence unit). *Одиниця персистентності* (persistence unit) – це сполучення відображень класів нашої доменної моделі з підключенням до бази даних, а також деякими іншими налаштуваннями конфігурації. Кожна програма має принаймні одну одиницю постійності. Деякі програми мають кілька, якщо вони спілкуються з декількома (логічними або фізичними) базами даних. Отже, нашим першим кроком є налаштування одиниці персистентності в конфігурації нашого додатку.

Налаштування одиниці персистентності

Стандартний файл налаштувань для одиниць персистентності знаходиться у середині проєкту в директорії META-INF та має ім'я persistence.xml. Створіть наведений нижче файл конфігурації для програми, яка розробляється.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
             version="3.0">
    <!-- 1 -->
    <persistence-unit name="pu">
        <!-- 2 -->
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</
provider>
        <class>main.dbdemo1.entity.Person</class>
        <properties>
            <!-- 3 Database connection settings -->
            <property name="jakarta.persistence.jdbc.driver"
                value="org.h2.Driver"/>
            <property name="jakarta.persistence.jdbc.url"
                value="jdbc:h2:file:./db/dbdemo1"/>
            <property name="jakarta.persistence.jdbc.user"
value="student"/>
            <property name="jakarta.persistence.jdbc.password"
value="1234"/>

            <!-- 4 Hibernate properties -->
            <property name="hibernate.dialect"
                value="org.hibernate.dialect.H2Dialect"/>
            <property name="hibernate.show_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

```
<property name="hibernate.format_sql" value="true"/>
<property name="hibernate.hbm2ddl.auto" value="create"/>
</properties>
</persistence-unit>
</persistence>
```

1. Файл `persistence.xml` налаштовує принаймні одну одиницю персистентності. Кожна одиниця повинна мати унікальне ім'я.

2. Оскільки JPA – це лише специфікація, розробнику потрібно вказати реалізацію API для конкретного постачальника `PersistenceProvider`. Персистентність, яка визначається таким чином, буде реалізована із використанням провайдера `Hibernate`. Також необхідно вказати повні імена класів-сущностей.

3. У цьому ж файлі треба вказати такі властивості JDBC:

- драйвер;
- URL-адреса бази даних;
- ім'я користувача;
- пароль.

4. Також можна вказати параметри `Hibernate`:

- Діалект для «спілкування» з БД (H2, оскільки використовується база даних на СУБД H2).

- Параметри відображення SQL-запитів, що генеруються `Hibernate` (під час виконання ми бажаємо бачити SQL-код чи ні).

- Також можна налаштувати форматування SQL-коду у логах (наприклад, тут вимагається від `Hibernate` гарно формувати SQL і генерувати коментарі в рядку SQL, у такому випадку розробник буде знати, що `Hibernate` виконав SQL-запит.

- Також, у процесі розробки, зручно налаштувати застосунок так, щоб кожного разу під час запуску програми

на виконання база даних створювалась з нуля. Це ідеальний варіант для автоматизованого тестування, коли ми хочемо працювати з чистою базою даних для кожного тестового запуску. Звісно, для «виробничого» оточення – готової програми, цей параметр можна просто прибрати.

Розглянемо більш докладно як виглядає простий персистентний клас, як створюється відображення, і що саме ми можемо зробити з екземплярами персистентного класу в JPA.

```
package main.dbdemo1.entity;

import jakarta.persistence.*;

@Entity
@Table(name = "CITY")
public class City {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "CITY_ID", nullable = false)
    private Integer id;

    @Column(name = "NAME", nullable = false, length = 50)
    private String name;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Кожен персистентний клас сутностей повинен обов'язково мати анотацію `@Entity`. Hibernate відображає цей клас у таблицю під назвою `CITY`.

Кожен персистентний клас сутності повинен мати атрибут-ідентифікатор з анотацією `@Id`. Hibernate відображає цей атрибут у стовпець таблиці з первинним ключем.

Значення ідентифікаторів повинні генеруватись. Ця анотація дозволяє автоматично генерувати значення id.

Зазвичай ми реалізуємо звичайні атрибути персистентного класу з приватними полями та пари `public` методів `getter/setter`. Hibernate відображає такі атрибути у стовпці таблиці з відповідними іменами.

Необов'язкова анотація `@Column` налаштовує це відображення, визначаючи ім'я відповідного стовпця, та деякі інші його параметри: довжину, обов'язковість та ін.

Атрибут `id` персистентного класу дозволяє програмі отримати доступ до ідентифікатора рядка у таблиці бази даних – значення первинного ключа для персистентного екземпляра. Якщо два екземпляри «City» мають однакове значення цього ідентифікатора, вони представляють один і той самий рядок у базі даних. Цей приклад використовує `Integer` для атрибута типу ідентифікатора, але це не є обов'язковою вимогою.

Hibernate дозволяє використовувати практично будь-що для типу ідентифікатора, проте найчастіше як ідентифікатор використовують цілі числа різної розрядності залежно від очікуваної кількості рядків у таблиці.

Легко побачити, що атрибути класу `City` мають методи доступу до поля у стилі `JavaBeans`. Клас також має (за замовчуванням) конструктор за замовчуванням без параметрів. Персистентні класи, створення яких розглядається у прикладах посібника, зазвичай будуть виглядати приблизно таким же чином. Зауважте, що на відміну від старих специфікацій `EntityBeans`, тепер уже не потрібно реалізовувати якийсь конкретний інтерфейс або розширювати будь-який спеціальний суперклас.

Екземплярами класу `City` можна керувати (зробити їх персистентними) за допомогою `Hibernate`, але це не обов'язково. Оскільки об'єкт `City` не реалізує жодних специфічних для роботи з персистентністю класів або інтерфейсів, розробники можуть використовувати його так само, як і будь-який інший клас `Java`:

```
City city = new City();
city.setName("Kyiv");
System.out.println("Hello from " + city.getName());
```

Може здатися, що в наведеному прикладі все дуже спрощено. Але фактично тут демонструється важлива особливість, яка відрізняє `Hibernate` від деяких інших рішень персистентності. Розробники завжди можуть використовувати персистентний клас у будь-якому контексті виконання – для цього не потрібен спеціальний контейнер.

Тепер клас `City` готовий. `JPA` надає можливості зберігати екземпляри в прив'язаній базі даних і писати запити, щоб знову завантажити їх у пам'ять програми.

Зберігання та завантаження об'єктів `City`

Зрозуміло, що цей посібник присвячений роботі з БД, тому розглянемо, як відбувається робота `JPA` з `Hibernate`, і тепер збережемо новий об'єкт `City` в базу даних.

```
class MainTest {
    @Test
    void storeLoadUpdatePersonTest() {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("pu");           // 1
        try {
            EntityManager em = emf.createEntityManager();           // 2
            em.getTransaction().begin();                             // 3
            City city = new City();                                   // 4
            city.setName("Kyiv");
            em.persist(city);                                         // 5
            em.getTransaction().commit();                             // 6
            // INSERT INTO CITY (NAME) VALUES ('Kyiv')
```

```
em.getTransaction().begin(); // 7
List<Person> people = em.createQuery(
    "SELECT c FROM City c", City.class).getResultList(); // 8
// SELECT * FROM CITY

people.getLast().setName("Mykolaiv"); // 9
em.getTransaction().commit(); // 10
// UPDATE CITY SET NAME = 'Mykolaiv' where ID = 1

assertAll(
    () -> assertEquals(1, people.size()), // 11
    () -> assertEquals("Mykolaiv",
        people.getFirst().getName()) // 12
);

em.close(); // 13
} finally {
    emf.close(); // 14
}
}
```

1. Спочатку потрібно використати EntityManagerFactory для зв'язку з базою даних. Цей API представляє одиницю персистентності, і більшість застосунків мають один EntityManagerFactory для однієї налаштованої одиниці персистентності. Як тільки застосунок стартує, він повинен створити EntityManagerFactory. Фабрика є потокобезпечною, тому весь код у програмі, який отримує доступ до бази даних, повинен отримувати доступ до неї.

2. Для виконання запитів треба почати новий сеанс з базою даних, для цього створюється EntityManager. Це контекст для всіх операцій з персистентністю.

3. Наступним кроком треба отримати доступ до стандартного API транзакції та розпочати транзакцію на цьому потоці виконання.

4. Після старту транзакції необхідно створити новий екземпляр потрібного класу моделі домену City та встановити його властивості (наприклад, назву).

5. Новий, поки не збережений у БД екземпляр сутності треба додати до контексту персистентності. Таким чином він отримує статус персистентного. Тепер Hibernate знає, що розробник планує зберегти ці дані, але не обов'язково викликає збереження у базу даних негайно.

6. Для завершення операції треба підтвердити транзакцію. Hibernate автоматично перевіряє контекст персистентності та виконує необхідний оператор SQL INSERT. Для кращого розуміння того, як працює Hibernate, у файлі persistence.xml, що показаний вище, було зроблено налаштування виведення автоматично згенерованих та виконуваних SQL-виразів у логах і продубльовано у коментарях до вихідного коду, коли вони виникають. Hibernate вставляє рядок у таблицю City з автоматично згенерованим значенням для стовпця первинного ключа CITY_ID та значенням NAME.

7. Кожна взаємодія з базою даних повинна відбуватися в межах транзакцій. Навіть якщо потрібно лише зчитати дані, усе одно треба розпочати нову транзакцію. Будь-яка потенційна несправність, яка може з'явитися, з цього моменту не вплине на раніше здійснену транзакцію.

8. Виконуємо запит, щоб отримати всі екземпляри City з бази даних.

9. Розробник може змінити значення поля. Hibernate виявляє це автоматично, оскільки завантажений об'єкт City все ще прикріплений до контексту персистентності, яким він був завантажений.

10. Під час завершення транзакції Hibernate перевіряє контекст збереження на наявність «брудного» стану об'єкта та автоматично виконує SQL UPDATE для синхронізації об'єктів у пам'яті зі станом об'єктів бази даних.

11. Виконується перевірка розміру списку City, що був отриманий з бази даних.

12. Перевіряється, що дані, які були збережені, є в базі даних. Найкращий метод – скористатися методом, що визначений у фреймворку модульного тестування JUnit 5 – `assertAll`, який перевіряє всі твердження, які йому передаються, навіть якщо деякі з них «падають». Два твердження, які перевіряються у прикладі, концептуально пов'язані між собою.

13. Раніше створений `EntityManager` після завершення роботи з ним має бути закритим.

14. Також у прикладі був створений `EntityManagerFactory`, тому також потрібно його закрити.

Мова запитів, яку було використано в цьому прикладі, це не SQL, а Jakarta Persistence Query Language (JPQL). Хоча синтаксично різниці в цьому тривіальному прикладі немає, `City` в рядку запиту посилається не на ім'я таблиці бази даних, а на ім'я персистентного класу. З цієї причини ім'я класу `City` у запиті чутливе до регістру. Якщо буде потрібно прив'язати клас до іншої таблиці, запит усе одно працюватиме. Більш докладно синтаксис цієї мови буде розглянуто далі.

Крім того, зверніть увагу, як Hibernate виявляє зміну текстового поля `name` екземпляра `City` та автоматично оновлює базу даних. Ця функція називається «автоматичною перевіркою бруду JPA» і цей приклад надає можливість побачити її в роботі. Це позбавляє розробника від потреби явно просити менеджера персистентності оновити базу даних, коли стан екземпляра змінюється всередині транзакції.

Убудована конфігурація Hibernate

Хоча базова (і обширна) конфігурація стандартизована в JPA, розробник не зможе отримати доступ до всіх функцій конфігурації Hibernate з властивостями в `persistence.xml`. Зауважимо, для більшості програм, навіть достатньо складних, не потрібні такі спеціальні параметри конфігурації.

А отже, їм не потрібно звертатися до початкового API та вбудованих специфічних налаштувань Hibernate.

Простий проєкт із використанням Spring Data JPA

Після того, як було розібрано простий проєкт на основі JPA із використанням Hibernate, перейдемо до розгляду першого застосунку Spring Data JPA, який буде зберігати об'єкт City в базі даних, а потім отримувати його. Спочатку треба додати залежності Spring до конфігурації Apache Maven.

Файл pom.xml буде мати наступний вигляд:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>ua.edu.nuos</groupId>
  <artifactId>dbdemo2</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>21</maven.compiler.source>
    <maven.compiler.target>21</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <version>2.3.232</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate.orm</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>7.0.4.Final</version>
    </dependency>
    <dependency>
      <groupId>org.springframework.data</groupId>
      <artifactId>spring-data-jpa</artifactId>
      <version>3.5.1</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-test</artifactId>
      <version>6.2.7</version>
    </dependency>
  </dependencies>
</project>
```

```

    </dependency>
  </dependencies>
</project>

```

Залежність `spring-data-jpa` забезпечує підтримку репозиторіїв для JPA та включає в себе транзитивні залежності від інших модулів, які нам знадобляться, таких як `spring-core` та `spring-context`. Нам також потрібна залежність `spring-test` для запуску тестів.

Стандартний конфігураційний файл для Spring Data JPA – це клас на Java, який створює та налаштовує біни, необхідні Spring Data. Конфігурацію можна описати за допомогою XML-файлу або коду Java. У нашому прикладі обираємо другий варіант. Створюємо наступний файл конфігурації для програми «Hello Spring Data».

```

@Configuration
@EnableJpaRepositories(basePackages = "main.repositories") // 1
class SpringDataConfiguration {

    @Bean // 2
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.h2.Driver"); // 3
        dataSource.setUrl("jdbc:h2:file:./db/dbdemo2"); // 4
        dataSource.setUsername("student"); // 5
        dataSource.setPassword("1234"); // 6
        return dataSource;
    }

    @Bean // 7
    public JpaTransactionManager transactionManager(EntityManagerFactory emf){
        return new JpaTransactionManager(emf);
    }

    @Bean // 8
    public JpaVendorAdapter jpaVendorAdapter() {
        HibernateJpaVendorAdapter jpaVendorAdapter =
            new HibernateJpaVendorAdapter();
        jpaVendorAdapter.setDatabase(Database.H2); // 9
        jpaVendorAdapter.setShowSql(true); // 10
        return jpaVendorAdapter;
    }

    @Bean // 11
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
        LocalContainerEntityManagerFactoryBean
            localContainerEntityManagerFactoryBean =

```

```
        new LocalContainerEntityManagerFactoryBean();
        LocalContainerEntityManagerFactoryBean.setDataSource(dataSource());
                                                                    //12
        Properties properties = new Properties();
        properties.put("hibernate.hbm2ddl.auto", "create");           // 13
        LocalContainerEntityManagerFactoryBean.setJpaProperties(properties);
        LocalContainerEntityManagerFactoryBean
            .setJpaVendorAdapter(jpaVendorAdapter());               // 14
        LocalContainerEntityManagerFactoryBean.setPackagesToScan("main");
                                                                    // 15
        return localContainerEntityManagerFactoryBean;
    }
}
```

1. Анотація `@EnableJpaRepositories` дозволяє шукати репозиторії Spring Data за допомогою сканування пакета (і всіх його підпакетів), отриманий як аргумент для анотації.

2. Створення джерела даних.

3. Підключення властивості JDBC драйвера.

4. Визначення URL-адреси бази даних.

5. Визначення імені користувача для доступу до БД.

6. Визначення пароля доступу до БД.

7. Створення біна – менеджера транзакцій на основі фабрики менеджера сутностей. Кожна взаємодія з базою даних повинна відбуватися в межах транзакцій, і Spring Data потребує менеджера транзакцій.

8. Створення адаптера JPA, необхідного JPA для взаємодії з Hibernate.

9. Налаштування створеного адаптера постачальника для доступу до бази даних H2.

10. Показувати SQL-код під час його виконання.

11. Створення біна `LocalContainerEntityManagerFactoryBean`. Це фабричний бін, який створює `EntityManagerFactory` відповідно до стандартного контракту на початкове завантаження контейнерів JPA.

12. Підключення джерела даних.

13. Налаштування бази даних, що буде створюватися з нуля при кожному виконанні програми (так зручно для тестування).

14. Установлення адаптера постачальника.

15. Налаштування пакетів для сканування на наявність класів сутностей. Оскільки сутність City розташована в пакеті main.entities, ми встановлюємо цей пакет для сканування.

Spring Data JPA забезпечує підтримку рівнів доступу до даних на основі JPA шляхом зменшення шаблонного коду та створення реалізацій для інтерфейсів репозиторіїв. Нам потрібно лише визначити власний інтерфейс репозиторію, щоб розширити один з інтерфейсів Spring Data.

```
public interface CityRepository extends CrudRepository<City, Integer> { }
```

Інтерфейс CityRepository розширює стандартний інтерфейс Spring Data JPA CrudRepository<City, Integer>. Це означає, що це репозиторій сутностей City з ідентифікатором (первинним ключем) типу Integer. Пам'ятайте, що клас City має поле id, анотоване як @Id типу Integer. Ми можемо безпосередньо викликати такі методи, як save(), findAll() або findById(), які успадковані від CrudRepository, і ми можемо використовувати їх без будь-якої іншої додаткової інформації для виконання звичайних операцій над базою даних. Spring Data JPA створить проксі-клас, що реалізує інтерфейс CityRepository та реалізує його методи (рис. 3.5).

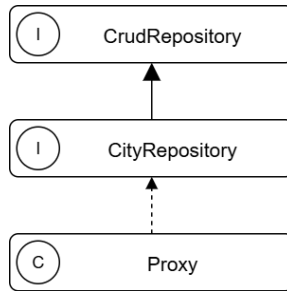


Рис. 3.5. Ієрархія інтерфейсів та класів Repository

Збережемо об'єкт City в базу даних за допомогою Spring Data JPA.

```

@ExtendWith(SpringExtension.class) // 1
@ContextConfiguration(classes = {SpringDataConfiguration.class}) // 2
public class SpringDataJPATest {

    @Autowired
    private CityRepository cityRepository; // 3

    @Test
    public void storeLoadMessage() {
        City city = new City(); // 4
        city.setName("Kyiv");
        cityRepository.save(city);

        List<City> cities = (List<City>) cityRepository.findAll();
        assertEquals(1, cities.size());
        assertEquals("Kyiv", cities.getFirst().getName());
    }
}
    
```

1. Підключаємо до тесту анотацію SpringExtension. Це розширення використовується для інтеграції контексту весняного тесту з тестом JUnit 5 Jupiter.

2. Контекст тесту Spring налаштовується за допомогою бінів, визначених у раніше представленому класі SpringDataConfiguration.

3. Бін `CityRepository` вводиться Spring за допомогою автозв'язування. Це можливо, оскільки пакет `main.repositories`, де знаходиться `MainRepository`, був використаний як аргумент анотації `@EnableJpaRepositories` у класі конфігурації. Якщо ми викличемо `cityRepository.getClass()`, ми побачимо, що він повертає щось на кшталт `com.sun.proxy.$Proxy41` – проксі, згенерований Spring Data, як показано на рис. 3.5.

4. Створюємо новий екземпляр класу моделі домену `City` та встановлюємо його властивість `name`.

5. Зберігаємо об'єкт `City` (`city`). Метод `save()` успадковується від інтерфейсу `CrudRepository`, а його тіло буде згенеровано Spring Data JPA під час створення класу проксі. Він просто збереже сутність `City` в базу даних.

6. Отримаємо всі об'єкти `City` зі сховища. Метод `findAll` успадкований від інтерфейсу `CrudRepository`, а його тіло буде згенеровано Spring Data JPA під час створення класу проксі. Він просто поверне всі сутності, що належать до класу `City`.

7. Перевіримо розмір списку `cities`, отриманих з бази даних, і пересвідчимося, що об'єкт `city`, який ми зберігали, є в базі даних.

8. Використаємо метод `JUnit 5 assertAll`, який перевіряє всі твердження, які йому передаються, навіть якщо деякі з них «падають». Два твердження, які ми перевіряємо, концептуально пов'язані між собою.

Легко побачити, що код тестового класу Spring Data JPA значно коротший, ніж код тестового класу на JPA. Це пов'язано з тим, що шаблонний код був видалений – більше немає явного створення об'єктів або явного контролю над транзакціями. Реалізується об'єкт репозиторію, і він надає згенеровані методи проксі-класу. Тепер навантаження зі створення конфігурації є більшим, але це слід робити лише один раз для кожного застосунку.

Порівняння підходів персистентності

У двох прикладах, що були розглянуті, було реалізовано прості застосунки, які взаємодіють з базою даних і використовують, як альтернативи «чистий» JPA на основі Hibernate і Spring Data JPA. Мета цих прикладів полягала в тому, щоб проаналізувати обидва підходи і побачити, як змінюється конфігурація та код. У таблиці 3.1 узагальнено характеристики цих підходів.

Таблиця 3.1. Характеристики підходів персистентності

Framework	Характеристики
JPA	• Використовує загальний API JPA і вимагає наявності постачальника персистентності. • Існує можливість перемикатися між постачальниками персистентності у конфігурації. • Потрібне чітке управління EntityManagerFactory, EntityManager та транзакціями. • Конфігурація та кількість коду, який потрібно написати, схожі на нативний підхід Hibernate. • Для тих, хто знайомий з нативною конфігурацією Hibernate, існує можливість перейти на підхід JPA, побудувавши EntityManagerFactory з нативної конфігурації Hibernate
Spring Data JPA	• Потрібні додаткові залежності Spring Data у проєкті. • Конфігурація також подбає про створення бінів, необхідних для проєкту, включаючи менеджер транзакцій. • Інтерфейс репозиторію потрібно лише оголосити, а Spring Data створить для нього реалізацію у вигляді проксі-класу зі згенерованими методами, які взаємодіють з базою даних. • Необхідний repository реалізується фреймворком, а не явно створюється програмістом. • Такий підхід вимагає найменшої кількості коду для написання, оскільки конфігурація бере на себе більшу частину навантаження

Приклади двох програм, що реалізують базові операції, у цьому розділі продемонстрували підходи Spring Data та JPA, але для розуміння вимог реальних застосунків зі

складними моделями даних цього недостатньо. Далі ми повернемося до задачі, що була сформульована й описана у вигляді моделей даних – застосунку DreamUniversity, систему управління університетом, для демонстрації JPA, Hibernate, а також Spring Data.

Реалізацію Java-застосунку DreamUniversity треба починати з розгляду представлення стандартної багаторівневої архітектури. Далі будуть розглянуті підходи створення об'єктів предметної області, вони будуть реалізовані на Java через створення персистентних класів-сутностей. Будуть розглянуті особливості того, що повинно бути в цих класах і яке місце вони займають у типовій багаторівневій архітектурі застосунків. Крім того, будуть розглянуті особливості персистентних класів-сутностей і те, як вони впливають на дизайн і реалізацію програм.

Наступним кроком буде опис основних параметрів метаданих співставлення – способи, якими розробник може повідомити Hibernate як персистентні класи та їхні властивості пов'язані з таблицями та стовпцями бази даних. Це може бути так само просто, як додавання анотацій безпосередньо у вихідний код класів Java або написання XML-документів, які в кінцевому підсумку розгортаються разом із скомпільованими класами Java, до яких Hibernate звертається під час виконання.

3.4. Відображення персистентних класів

Розглянемо основні параметри відображення та принципи зіставлення класів сутностей із SQL-таблицями більш докладно. Знання та розуміння цих параметрів необхідно для структурування класів у застосунку, незалежно від того, чи працюєте ви з Hibernate, Spring Data JPA або якимось іншим фреймворком persistence, що реалізує специфікацію JPA. Продемонструємо та проаналізуємо, яким чином можна обробляти ідентифікатори бази даних та первинні ключі, а також як можна використовувати різноманітні

налаштування метаданих для Hibernate або Spring Data JPA, використовуючи Hibernate як постачальника персистентності. Буде показано, як відбувається завантаження та збереження екземплярів класів моделі домену.

Spring Data JPA, як абстракція доступу до даних, побудована навколо провайдера JPA (наприклад, Hibernate) з метою значно скоротити шаблонний код, необхідний для взаємодії з базою даних. Саме тому, після того, як будуть зроблені відображення персистентних класів, їх можна буде використовувати як із Hibernate, так із Spring Data JPA. Наші приклади продемонструють це, і всі наші приклади відображення використовуватимуть анотації JPA. Однак перш ніж будуть розглянуті відображення, буде визначено суттєву різницю між сутностями та типами значень і пояснено, як найкраще слід підійти до об'єктного / реляційного відображення вашої моделі домену. Роль інженера-розробника полягає в тому, щоб установити зв'язок між доменом *програми*, що є середовищем задачі, яку система повинна вирішити, і доменом *рішення*, який є програмним забезпеченням і технологіями, що будуть будувати цю систему.

Відображення сутностей з ідентичністю

Відображення сутностей з ідентичністю вимагає розуміння того, як у Java розрізняються тотожність та рівність. Після докладного розуміння принципів перевірки на тотожність та рівність треба розглянути приклади класів сутностей та їхніх відображень, обговорити такі терміни, як *ідентичність у базі даних*, і подивитися, як JPA управляє ідентичністю. Після цього з'явиться можливість більш глибоко розібратися зі структурою таблиць та зв'язаних з ними персистентних класів-сутностей і вибрати первинний ключ, налаштувати генератори ключів і, нарешті, пройти по стратегіям генератора ідентифікаторів.

Розуміння тотожності та рівності Java

Розробники, що використовують мову програмування Java, розуміють різницю між тотожністю та рівністю об'єктів Java. Тотожність об'єктів (==) – це поняття, яке визначає віртуальна машина Java. Два посилання ідентичні, якщо вони вказують на одне й те ж місце пам'яті.

З іншого боку, рівність об'єктів – це поняття, яке визначається методом класу equals(), який іноді також називають перевіркою *еквівалентності*. Еквівалентність означає, що два різні (неідентичні) екземпляри мають однакове значення – один і той самий стан. Якщо у вас є стіс свіжонадрукованих книг одного виду, і вам потрібно вибрати одну з них, це означає, що вам доведеться вибрати один з декількох не ідентичних, але рівноцінних предметів.

Два різні екземпляри String є рівними, якщо вони представляють одну й ту ж послідовність символів, навіть якщо кожен з них має своє розташування в просторі пам'яті віртуальної машини.

Персистентність ускладнює цю картину. При об'єктній / реляційній персистентності постійний екземпляр – це представлення в пам'яті певного рядка (або рядків) таблиці (або таблиць) бази даних. Поряд з тотожністю та рівністю Java ми визначаємо *ідентичність бази даних*. Тепер у вас є три методи розрізнення посилань:

Ідентичність об'єктів. Об'єкти ідентичні, якщо вони займають одну й ту саму ділянку пам'яті в JVM. Це можна перевірити за допомогою оператора `a == b`. Це поняття відоме як *об'єктна тотожність*.

Об'єктна рівність. Об'єкти є рівними, якщо вони мають однаковий стан, як визначено методом `a.equals(Object b)`. Класи, які явно не перевизначають цей метод, успадковують реалізацію, визначену `java.lang.Object`, який порівнює ідентичність об'єкта за допомогою перевірки на тотожність

`==`. Це поняття відоме як *об'єктна рівність*. Як відомо зі специфікації мови програмування Java, об'єктна рівність, яку самостійно описує розробник, повинна мати властивості рефлексивності, симетричності та транзитивності. З цього випливає, що якщо `a == b`, то і `a.equals(b)`, і `b.equals(a)` повинні бути істинними.

Ідентичність у базі даних. Об'єкти, що зберігаються в реляційній базі даних, ідентичні, якщо вони належать до однієї таблиці та мають однакові значення первинного ключа. Ця концепція, яку необхідно відобразити в просторі Java, відома як *ідентичність у базі даних*.

Тепер потрібно подивитися, як ідентичність бази даних пов'язана з ідентичністю об'єкта і як розробник може виразити ідентичність бази даних у метаданих відображення. Як приклад, зіставимо сутність моделі домену.

Перший клас сутності та відображення

Анотації `@Entity` недостатньо для відображення постійного класу. Також знадобиться анотація `@Id`, як показано далі.

Сутності в JPA – це не що інше, як POJO (Plain Old Java Objects – звичайні об'єкти Java), що представляють дані, які можуть зберігатися в базі даних. Сутність становить собою таблицю, що зберігається в базі даних. Кожен екземпляр сутності представляє рядок у таблиці.

Анотація `@Entity`

Припустимо, у нас є POJO під назвою `Student` (у моделі БД, що розглянута в розділі 2 є така таблиця, але тут деякі властивості з тієї таблиці не будуть використані), який представляє дані студента, і ми хотіли б зберегти їх у базі даних:

```
public class Student {  
    // fields, getters and setters  
}
```

Для цього ми повинні визначити сутність так, щоб JPA знав про неї.

Отже, давайте визначимо його за допомогою анотації `@Entity`. Ми повинні вказати цю анотацію на рівні класу. Ми також повинні переконатися, що сутність має конструктор без аргументів та первинний ключ:

```
@Entity
public class Student {
    // fields, getters and setters
}
```

Ім'я сутності за замовчуванням дорівнює імені класу. Ми можемо змінити його ім'я за допомогою елемента `name`:

```
@Entity(name = "STUDENT")
public class Student {
    // fields, getters and setters
}
```

Оскільки різні реалізації JPA намагатимуться створювати підкласи нашої сутності, щоб забезпечити свою функціональність, класи сутностей не повинні оголошуватися як `final`.

Анотація `@Id`

Кожна сутність JPA повинна мати первинний ключ, який однозначно її ідентифікує. Анотація `@Id` визначає первинний ключ. Ми можемо генерувати ідентифікатори різними способами, які задаються анотацією `@GeneratedValue`.

Ми можемо вибрати одну з чотирьох стратегій генерації `id` з елементом стратегії. Значенням може бути `AUTO`, `TABLE`, `SEQUENCE` або `IDENTITY`:

```
@Entity(name = "STUDENT")
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
```

```
private String name;  
    // getters and setters  
}
```

Налаштування генераторів ключів

GenerationType.AUTO. Hibernate (або Spring Data JPA, що використовує Hibernate як провайдера персистентності) вибирає відповідну стратегію, запитуючи діалект SQL вашої налаштованої бази даних, що краще. Це еквівалентно `@GeneratedValue()` без будь-яких налаштувань.

GenerationType.SEQUENCE. Hibernate (або Spring Data JPA, що використовує Hibernate як провайдера персистентності) очікує (і створює, якщо ви використовуєте інструменти) послідовність з іменем `HIBERNATE_SEQUENCE` у вашій базі даних. Послідовність буде викликатися окремо перед кожною операцією `INSERT`, генеруючи послідовні числові значення.

GenerationType.IDENTITY. Hibernate (або Spring Data JPA, що використовує Hibernate як провайдера персистентності) очікує (і створює в таблиці `DDL`) спеціальний стовпець первинного ключа з автозбільшенням, який автоматично генерує числове значення при `INSERT` у базі даних.

GenerationType.TABLE. Hibernate (або Spring Data JPA, що використовує Hibernate як провайдера персистентності) використовуватиме додаткову таблицю у схемі вашої бази даних, яка містить наступне числове значення первинного ключа, з одним рядком для кожного класу сутності. Ця таблиця буде прочитана та оновлена перед операцією `INSERT`. Ім'я таблиці за замовчуванням – `HIBERNATE_SEQUENCES` зі стовпцями `SEQUENCE_NAME` та `NEXT_VALUE`.

Хоча `AUTO` здається зручним, іноді розробник потребує більше контролю над тим, як створюються ідентифікатори, тому зазвичай замість цього вам слід явно налаштувати

стратегію генерації первинного ключа. Більшість програм працюють з послідовностями баз даних, але ви можете захопити налаштувати ім'я та інші параметри послідовності бази даних. Тому, замість того, щоб вибирати одну зі стратегій JPA, ви можете зіставити ідентифікатор з `@GeneratedValue` (`generator = «ID_GENERATOR»`). Це *генератор іменованих ідентифікаторів*; тепер ви можете налаштувати конфігурацію `ID_GENERATOR` незалежно від ваших класів сутностей.

JPA має дві вбудовані анотації, які призначені для використання при налаштуванні іменованих генераторів: `@jakarta.persistence.SequenceGenerator` і `@jakarta.persistence.TableGenerator`. За допомогою цих анотацій надається можливість створення генератора імен із власною послідовністю та іменами таблиць. Як зазвичай з анотаціями JPA, їх, на жаль, можна використовувати лише в заголовку (можливо, в іншому випадку порожнього) класу, а не у файлі `package-info.java`.

Якщо анотувати поля сутності, постачальник JPA використовуватиме ці поля для отримання та встановлення стану сутності. Окрім доступу до полів, також можна здійснювати доступ до властивостей або змішаний доступ, що дозволить використовувати доступ до полів і властивостей в одній сутності.

Анотація `@Table`

У більшості випадків ім'я таблиці в базі даних та ім'я сутності не повинні збігатися. У цих випадках просто можна вказати ім'я таблиці за допомогою анотації `@Table`:

```
@Entity
@Table(name = "STUDENT")
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
```

```
private String name;  
// getters and setters  
}
```

Також можна вказати схему за допомогою елемента `schema`:

```
@Table(name = "STUDENT", schema = "PUBLIC")
```

Назва схеми допомагає відрізнити один набір таблиць від іншого.

Якщо анотацію `@Table` не використовувати, то ім'я таблиці буде співпадати з іменем сутності. При цьому постачальник персистентності бере на себе завдання трансляції імен та перетворення регістру літер у іменах.

Анотація `@Column`

Так само, як і з анотацією `@Table`, існує можливість використовувати анотацію `@Column` для указання деталей стовпця в таблиці.

Анотація `@Column` має багато елементів, таких як `name`, `length`, `nullable` та `unique`:

```
@Entity  
@Table(name="STUDENT")  
public class Student {  
    @Id  
    @GeneratedValue(strategy=GenerationType.AUTO)  
    @Column(name="STUDENT_ID", nullable=false, unique=true)  
    private Long id;  
  
    @Column(name="STUDENT_NAME", length=50, nullable=false,  
unique=false)  
    private String name;  
  
    // other fields, getters and setters  
}
```

Елемент `name` вказує ім'я стовпця в таблиці. Елемент `length` вказує на його довжину. Елемент `nullable` визначає, чи може стовпець містити значення `null`, а елемент `unique` визначає, чи є значення у стовпці унікальними.

Якщо ми не вкажемо цю анотацію, то ім'я стовпця в таблиці буде співпадати з назвою поля.

Анотація `@Transient`

Іноді може бути так, що є потреба **зробити поле неперсистентним**. Для цього може використовуватись анотація `@Transient`. Вона вказує, що поле не буде зберігатися.

Наприклад, вік студента можна розрахувати за датою його народження.

Отже, у коді треба анотувати поле віку студента – `age` за допомогою `@Transient`:

```
@Entity
@Table(name="STUDENT")
public class Student {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    @Column(name="STUDENT_NAME", length=50, nullable=false)
    private String name;

    @Transient
    private Integer age;

    // other fields, getters and setters
}
```

У результаті поле `age` не буде зберігатися в таблиці.

Анотація `@Temporal`

Достатньо часто є потреба у зберіганні значення часу або дати в таблиці бази даних. Традиційно для цього використовувалась анотація `@Temporal`:

```
@Entity
@Table(name="STUDENT")
public class Student {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    @Column(name="STUDENT_NAME", length=50, nullable=false,
unique=false)
    private String name;

    @Transient
    private Integer age;

    @Temporal(TemporalType.DATE)
    private Date birthDate;

    // other fields, getters and setters
}
```

Однак, починаючи із JPA 3.1, також підтримується збереження даних «нових» типів дати та часу, що вперше з'явились у Java 8: `java.time.LocalDate`, `java.time.LocalDateTime`, `java.time.OffsetTime` і `java.time.OffsetDateTime`. Для таких класів анотація `@Temporal` не потрібна.

Анотація `@Enumerated`

Іноді виникає потреба зберегти в базі даних тип переліку Java.

Для вирішення цієї задачі може використовуватись анотація `@Enumerated`, щоб указати, чи слід зберігати перелік за іменем чи за порядковим номером (за замовчуванням):

```
public enum Gender {
    MALE,
    FEMALE
}

@Entity
@Table(name="STUDENT")
public class Student {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;
```

```
@Column(name="STUDENT_NAME", length=50, nullable=false, unique=false)
private String name;

@Transient
private Integer age;

@Temporal(TemporalType.DATE)
private Date birthDate;

@Enumerated(EnumType.STRING)
private Gender gender;

// other fields, getters and setters
}
```

Розробнику не потрібно вказувати анотацію `@Enumerated`, якщо він використовує для збереження `Gender` за порядковим номером у переліку.

Однак, щоб зберегти `Gender` за іменем у переліку, треба використати налаштування анотацією `@Enumerated` з параметром `EnumType.STRING`.

3.5. Відображення типів Java та SQL за допомогою конверторів

Можна вважати, що `Hibernate` або `Spring Data JPA` за допомогою `Hibernate` вибере правильний тип SQL, коли буде відображати рядкову властивість типу `java.lang.String`. Але що таке правильне відображення між типами Java та SQL і як розробник може його контролювати? Сформулюємо відповідність між цими типами в міру того, як заглибимося у специфіку.

Убудовані типи

Будь-який провайдер JPA повинен підтримувати мінімальний набір перетворень типів Java-to-SQL. `Hibernate` та `Spring Data JPA` за допомогою `Hibernate` підтримують усі ці відображення, а також деякі додаткові адаптери, які не є стандартними, але корисними на практиці. Для початку розглянемо примітиви Java та їхні SQL-еквіваленти.

Примітивні та числові типи

Убудовані типи, наведені в таблиці 3.2, показують, як будуть відображатися примітивні типи мови Java та їхні типи-обгортки у відповідні стандартні типи мови SQL. Крім того, у мові Java існує два числові типи для роботи з великими числами: `BigInteger` та `BigDecimal`. Вони також наведені в таблиці. Імена у стовпчику «Назва» специфічні для Hibernate. Вони будуть використані пізніше під час налаштування відображення типів.

Таблиця 3.2. Прості типи даних

Ім'я	Тип Java	Тип ANSI SQL
<code>integer</code>	<code>int, java.lang.Integer</code>	<code>INTEGER</code>
<code>long</code>	<code>long, java.lang.Long</code>	<code>BIGINT</code>
<code>short</code>	<code>short, java.lang.Short</code>	<code>SMALLINT</code>
<code>byte</code>	<code>byte, java.lang.Byte</code>	<code>TINYINT</code>
<code>float</code>	<code>float, java.lang.Float</code>	<code>FLOAT</code>
<code>double</code>	<code>double, java.lang.Double</code>	<code>DOUBLE</code>
<code>boolean</code>	<code>boolean, java.lang.Boolean</code>	<code>BOOLEAN</code>
<code>big_decimal</code>	<code>java.math.BigDecimal</code>	<code>NUMERIC</code>
<code>big_integer</code>	<code>java.math.BigInteger</code>	<code>NUMERIC</code>

Іноді трапляється так, що СУБД, що використовується, не підтримує деякі з перерахованих типів SQL. Ці імена типів SQL є стандартними іменами типів ANSI. Більшість постачальників СУБД ігнорують цю частину стандарту SQL зазвичай тому, що їх застарілі системи типів передували стандарту. Однак JDBC забезпечує часткову абстракцію специфічних для постачальника типів даних, дозволяючи Hibernate працювати зі стандартними типами ANSI під час виконання операторів DML, таких як `INSERT` і `UPDATE`. Для генерації схеми для конкретного продукту Hibernate переходить зі стандартного типу ANSI на відповідний тип для конкретного постачальника, використовуючи налаштований

діалект SQL. Це означає, що розробнику зазвичай не потрібно турбуватися про типи даних SQL, якщо він покладається на можливість Hibernate створити схему бази даних.

Якщо ж розробник має існуючу схему або йому потрібно знати нативний тип даних для СУБД, що використовується, він може подивитися на джерело нашого налаштованого діалекту SQL. Наприклад, H2Dialect, який поставляється з Hibernate, містить таке відображення з типу ANSI NUMERIC на специфічний для постачальника тип DECIMAL: `registerColumnType(Types.NUMERIC, «decimal($p,$s)»)`. Тип NUMERIC SQL підтримує налаштування десяткової точності та масштабу. Стандартним значенням точності та масштабу для властивості `BigDecimal`, наприклад, є `NUMERIC(19, 2)`. Щоб перевизначити це для генерації схеми, можна застосувати анотацію `@Column` до властивості та встановити її параметри точності й масштабу. Далі розглянемо типи, які відображаються на рядкові дані в базі даних.

Таблиця 3.3. Типи даних, що відображаються рядковими типами у БД

Name	Java type	ANSI SQL type
String	<code>java.lang.String</code>	VARCHAR
character	<code>char[]</code> , <code>Character[]</code> , <code>java.lang.String</code>	CHAR
yes_no	<code>boolean</code> , <code>java.lang.Boolean</code>	CHAR(1), 'Y' or 'N'
true_false	<code>boolean</code> , <code>java.lang.Boolean</code>	CHAR(1), 'T' or 'F'
class	<code>java.lang.Class</code>	VARCHAR
locale	<code>java.util.Locale</code>	VARCHAR
timezone	<code>java.util.Timezone</code>	VARCHAR
currency	<code>java.util.Currency</code>	VARCHAR

Система типів Hibernate вибирає тип даних SQL залежно від оголошеної довжини рядкового значення: якщо властивість `String` анотується `@Column(length = ...)` або `@Length`

у Bean Validation, Hibernate вибирає правильний тип даних SQL для заданого розміру рядка. Цей вибір також залежить від налаштованого діалекту SQL. Наприклад, для MySQL, коли схема генерується Hibernate, довжина до 65535 створює звичайний стовпець VARCHAR(length). Для довжини до 16777215 створюється специфічний для MySQL тип даних MEDIUMTEXT, а для ще більшої довжини використовується LONGTEXT. Довжина за замовчуванням для всіх властивостей java.lang.String при використанні Hibernate становить 255, тому без будь-якого додаткового відображення, властивість String відображається на стовпець VARCHAR(255). Проте, варто зауважити, що розробник може налаштувати вибір типів, розширивши клас для використання потрібного діалекту SQL. Інформацію про нього можна отримати з документації СУБД, що використовується.

База даних зазвичай дозволяє інтернаціоналізувати текст за допомогою використання набору символів (UTF-8) за замовчуванням для всієї бази даних або принаймні цілих таблиць. Це специфічний параметр СУБД. Якщо розробнику потрібен більш точний контроль і він хоче перейти на націоналізовані варіанти типів даних символів (наприклад, NVARCHAR, NCHAR або NCLOB), треба використовувати анотації відображення @org.hibernate.annotations.Nationalized.

Також існують убудовані деякі спеціальні перетворювачі для застарілих баз даних або СУБД із системами обмеженого типу, таких як Oracle. СУБД Oracle навіть не має типу даних з логічним значенням – один із типів даних, які вимагає реляційна модель. Тому багато існуючих схем Oracle представляють логічні значення із символами Y/N або T/F. Або, як це зроблено за замовчуванням у діалекті Oracle Hibernate, – очікується та генерується стовпець типу NUMBER(1,0). Знову ж таки, якщо потрібна інформація щодо конкретного

діалекту SQL СУБД, її треба шукати в офіційній документації конкретного постачальника.

У наступній таблиці наведено типи, що відповідні даті й часу в базі даних.

Таблиця 3.4. Типи даних дати і часу

Name	Java type	ANSI SQL type
date	java.util.Date, java.sql.Date	DATE
time	java.util.Date, java.sql.Time	TIME
timestamp	java.util.Date, java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE
duration	java.time.Duration	BIGINT
instant	java.time.Instant	TIMESTAMP
localdatetime	java.time.LocalDateTime	TIMESTAMP
localdate	java.time.LocalDate	DATE
localtime	java.time.LocalTime	TIME
offsetdatetime	java.time.OffsetDateTime	TIMESTAMP
offsettime	java.time.OffsetTime	TIME
zoneddatetime	java.time.ZonedDateTime	TIMESTAMP

У моделі домену ми можемо представити дані дати й часу у вигляді або `java.util.Date`, або підкласів `java.util.Date`, визначених у пакеті `java.sql`, або класів, що вперше з'явилися у Java 8 у пакеті `java.time`. Найкращим рішенням на даний момент є використання Java 8 API з пакетом `java.time`. Ці класи можуть представляти дату, час, дату з часом, або вони навіть можуть включати в себе зсув до зони UTC (`OffsetDateTime` і `OffsetTime`). Jakarta Persistence 3.2 (і JPA 2.2) офіційно підтримує класи дати і часу Java 8.

Поведінка Hibernate для властивостей `java.util.Date` спочатку може здивувати: під час зберігання `java.util.Date`, Hibernate не поверне `java.util.Date` після завантаження. Він поверне об'єкт типу `java.sql.Date`, `java.sql.Time` або `java.sql.Timestamp`, залежно від того, чи зіставлено

властивість з `TemporalType.DATE`, `TemporalType.TIME`, або `TemporalType.TIMESTAMP`.

Hibernate повинен використовувати підклас `JDBC` під час завантаження даних з бази даних, оскільки типи баз даних мають більш високу точність, ніж `java.util.Date`. Як відомо, `java.util.Date` має точність до мілісекунд, але позначка часу `java.sql.Time` включає в себе інформацію про наносекунди, яка може бути присутня в базі даних. Hibernate не обрізатиме цю інформацію, щоб умістити значення у `java.util.Date`, що може призвести до проблем під час спроби порівняти значення `java.util.Date` з методом `equals()`; він не симетричний з методом `equals()` підкласу `java.sql.Timestamp`.

Існує багато різних проблем обробки дати й часу. Про це є багато інформації в офіційній документації та специфічних тьюторіалах від розробників. Усі ці проблеми зникнуть, якщо ви вирішите представляти дані дати й часу за допомогою класів `Java 8` `LocalDate`, `LocalTime`, `LocalDateTime`. Проте, оскільки розробники все ще можуть стикатися з великою кількістю коду, який використовує старі класи, потрібно знати про проблеми, які можуть виникнути.

Далі розглянемо типи, які відображають двійкові дані та великі значення в базі даних. Одразу ж зауважимо, що такі відображення дуже сильно залежать від виробника СУБД.

Якщо властивість у постійному класі `Java` має тип `byte[]`, Hibernate відображає її на стовпець `VARBINARY`. Реальний тип даних SQL буде залежати від діалекту; наприклад, у `PostgreSQL` тип даних – `BYTEA`, а у СУБД `Oracle` – `RAW`. У деяких діалектах довжина, установлена за допомогою `@Column`, також впливає на обраний нативний тип; наприклад, `LONG RAW` використовується для довжини 2000 і більше в `Oracle`. У `MySQL` типом даних SQL за замовчуванням буде `TINYBLOB`. Залежно від довжини, установленної

в анотації `@Column`, це може бути BLOB, MEDIUMBLOB або LONGBLOB.

Таблиця 3.5. Бінарні дані та великі значення

Name	Java type	ANSI SQL type
binary	byte[], java.lang.Byte[]	VARBINARY
text	java.lang.String	CLOB
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB
serializable	java.io.Serializable	VARBINARY

Властивість `java.lang.String` відображається на стовпець SQL VARCHAR, і те саме справедливо для `char[]` і `Character[]`. Як ми вже обговорювали, деякі діалекти реєструють різні нативні типи залежно від заявленої довжини.

Hibernate ініціалізує значення властивості відразу, коли завантажується екземпляр сутності, який містить змінну властивості. Це незручно, коли вам доводиться мати справу з потенційно великими значеннями, тому ви зазвичай захочете перевизначити це відображення за замовчуванням. Специфікація JPA має зручну анотацію ярликів для цієї мети, `@Lob`:

```
@Entity
public class Document {
    @Id
    private Integer id;
    @Lob
    private byte[] image;
    @Lob
    private String description;
    // ...
}
```

Це відображає `byte[]` з типом даних SQL BLOB, а `String` – з CLOB. На жаль, лінивого завантаження з такою конструкцією у вас усе одно не вийде. Hibernate або Spring Data JPA за допомогою Hibernate повинні були б перехоплювати доступ до полів і, наприклад, завантажувати байти зображення при

виклику `someItem.getImage()`. Цей підхід вимагає інструментарію байт-коду класів після компіляції, для введення додаткового коду.

Як варіант, можна змінити тип властивості у класі Java. JDBC підтримує великі об'єкти (LOB) безпосередньо. Якщо властивість Java має вигляд `java.sql.Clob` або `java.sql.Blob`, ви отримаєте ліниве завантаження без інструментарію байт-коду:

```
@Entity
public class Document {
    @Id
    private Integer id;
    @Lob
    private java.sql.Blob image;
    @Lob
    private java.sql.Clob description;
    // ...
}
```

Примітка. Що означає BLOB/CLOB?

BLOB інтерпретується як двійковий великий об'єкт: двійкові дані (зазвичай мультимедійний об'єкт – зображення, відео або аудіо) зберігаються як єдине ціле. CLOB означає Character Large Object – дані про символи, що зберігаються в окремому місці, на яке таблиця лише посилається.

Ці класи JDBC включають у себе поведінку для завантаження значень на вимогу. Коли завантажувється екземпляр сутності-власника, значення властивості є заповнювачем, а реальне значення не матеріалізується відразу. Як тільки ви отримуєте доступ до властивості в рамках однієї транзакції, значення матеріалізується або навіть передається безпосередньо (клієнту) без використання тимчасової пам'яті:

```
Document document = em.find(Document.class, docId);
InputStream imageDataStream = document.getImageBlob().
getBinaryStream();
ByteArrayOutputStream outStream = new ByteArrayOutputStream();
StreamUtils.copy(imageDataStream, outStream);
byte[] imageBytes = outStream.toByteArray();
```

Недоліком є те, що модель домену тоді прив'язана до JDBC; у модульних тестах ви не можете отримати доступ до властивостей LOB без з'єднання з базою даних. Щоб створити та встановити значення Blob або Clob, Hibernate пропонує деякі зручні методи. Цей приклад зчитує байти `byteLength` з `InputStream` безпосередньо в базу даних, не споживаючи тимчасову пам'ять:

```
Session session = em.unwrap(Session.class);
Blob blob = session.getLobHelper().createBlob(imageInputStream,
byteLength);
someItem.setImageBlob(blob);
em.persist(someItem);
```

Нарешті, Hibernate забезпечує резервну серіалізацію для будь-якого типу власності, який є `java.io.Serializable`. Це відображення перетворює значення властивості в потік байтів, що зберігається у стовпці `VARBINARY`. Серіалізація та десеріалізація відбуваються, коли екземпляр сутності-власника зберігається та завантажується. Зауважимо, що використовувати цю стратегію слід з особливою обережністю, тому що дані живуть довше, ніж застосунки. Одного разу ніхто не дізнається, що означають ці байти в базі даних. Серіалізація іноді корисна для тимчасових даних, таких як налаштування користувача, дані сеансу входу тощо.

Hibernate підбере правильний тип адаптера залежно від типу Java властивості. Якщо вам не подобається прив'язка за замовчуванням, завжди можна реалізувати конвертер даних. Як це зробити, можна прочитати в керівництві розробника Hibernate.

Оскільки цей посібник ставить на меті дати лише основні моменти роботи з БД у Java із використанням Hibernate, питання розширення механізмів відображення користувальницьких типів тут не розглядатимуться.

3.6. Шаблони роботи з персистентними даними

Далі необхідно спроектувати структуру застосунку з БД. Існують загальноприйняті шаблони роботи з персистентними даними.

У цьому розділі представлені найактуальніші з цих шаблонів, але є й інші, які не розглядаються в посібнику, більшість з них докладно описані на вебсайті Мартіна Фаулера [11].

Data Access Object (DAO)

Шаблон «Об'єкт доступу до даних» (DAO) призначений для зберігання та отримання інформації з баз даних. Це дозволяє розробникам ізолювати бізнес-рівень для шару персистентності, який у більшості випадків пов'язаний з базою даних, але може бути будь-чим, що відповідає за доступ до інформації.

Цей шаблон приховує складність виконання всіх операцій CRUD у базі даних, тому ви можете змінювати або вносити модифікації без значного впливу на всі інші шари. Цей шаблон є поширеним у стандартних реалізаціях Jakarta EE, але не часто зустрічається у Spring Data, оскільки більшість розробників використовують інший шаблон – шаблон репозиторію. Але деякі випадки, пов'язані з продуктивністю або операціями, не підтримуються для цього шаблону, тому шаблон DAO, у такому випадку може стати у нагоді.

Щоб продемонструвати цей шаблон, згадаємо таблицю CITY нашої моделі. Для її відображення створимо персистентний клас-сутність:

```
@Entity
@Table(name = "CITY")
public class City {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID", nullable = false)
    private Integer id;
```

```

@Column(name = "NAME", nullable = false, length = 50)
private String name;

public Integer getId() {
    return id;
}

public void setId(Integer id) {
    this.id = id;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}
}

```

Для доступу до таблиці в БД та роботі з цим об'єктом (або колекцією таких об'єктів) можна скористатися шаблоном DAO.

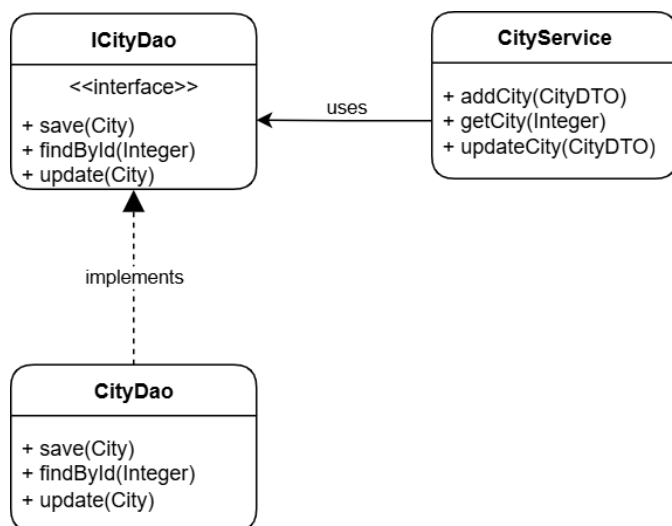


Рис. 3.6. Шаблон Data Access Object

Коли розробники починають використовувати цей шаблон, більшість з них створює загальний інтерфейс з усіма можливими методами і, на основі нього, створюють один клас для доступу до конкретної таблиці та зменшення кількості файлів, пов'язаних з доступом до бази даних. Лістинг показує різні методи, які можуть бути у типового DAO. Звісно, ви можете створити інші, але більшість фреймворків реалізують ці методи.

```
public interface CommonDao<T> {  
    Optional<T> get(long id);  
    List<T> getAll();  
    void save(T t);  
    void delete(T t);  
}
```

Тепер, коли вже розроблено загальний інтерфейс, реалізація конкретних класів DAO буде більш-менш схожою у всіх випадках. Різниця стосується лише імені таблиці та запитів для доступу до бази даних, але всі DAO дотримуються одного й того ж формату.

Наступний список показує можливу реалізацію інтерфейсу, який був наведений у попередньому лістингу. Звісно, треба врахувати, що це – не єдиний можливий варіант реалізації, існують також інші способи зробити це.

```
public class CityDao implements CommonDao<City> {  
    private List<City> cities = new ArrayList<>();  
    public CityDao() {  
        cities.add(new City(1, "Kyiv"));  
        cities.add(new City(2, "Mykolaiv"));  
    }  
    public Optional<City> get(long id) {  
        return Optional.ofNullable(cities.get((int) id));  
    }  
    public List<City> getAll() {  
        return cities;  
    }  
    public void save(City city) {  
        cities.add(city);  
    }  
}
```

```

public void delete(City city) {
    cities.remove(city);
}
}

```

***Примітка.** Орієнтуючись тільки на приклад шаблону, у пам'яті застосунку немає реального зв'язку між базою даних і створенням міст, але концепція та сама.*

Шаблон Repository

Шаблон Repository звертається до бази даних для отримання певної інформації, уводячи абстракцію для решти шарів. Сутність шаблону Repository полягає в емуляції або посередництві між доменом і відображенням даних за допомогою інтерфейсів для доступу до об'єктів домену.

Основна відмінність між шаблонами DAO і Repository полягає в тому, що Repository знаходяться поруч з бізнес-логікою і намагаються приховати всю логіку, пов'язану із запитами до бази даних. Найбільш поширеним застосуванням є створення інтерфейсу, який містить певні методи. Такі фреймворки, як Spring Boot і Quarkus, перетворюють методи в конкретні запити до бази даних. На противагу цьому, DAO реалізують усю логіку, використовуючи запити для доступу до бази даних. У таблиці 3.6 наведено порівняння основних відмінностей між підходами DAO та Repository.

Таблиця 3.6. Основні відмінності між DAO та Repository

DAO	Repository
Шаблон ближче до бази даних, тому що працює із запитами та таблицями	Шаблон ближче до рівня бізнес / домен, оскільки використовує абстракцію, щоб приховати реалізацію
DAO не можуть містити репозиторії, оскільки вони знаходяться на різних рівнях	Репозиторії можуть містити або реалізовувати DAO
Абстракція даних	Абстракція колекції об'єктів

Spring Data JPA пропонує розробникам базовий інтерфейс для реалізації логіки пагінації (розбиття результатів запиту на сторінки) та операцій CRUD без необхідності додаткової роботи. Наступний фрагмент коду показує приклад інтерфейсу, який слідує за шаблоном репозиторію.

```
public interface CityRepository {
    Optional<City> get(long id);
    List<City> getAll();
    void save(City city);
    void delete(City city);
}
```

За потреби, розробник може використовувати шаблон репозиторію з одним або декількома репозиторіями, щоб приховати логіку доступу до інформації. Наприклад, можна використати інтерфейс, що наведений вище, і створити конкретний клас, який використовує його:

```
public class CityRepositoryImpl implements CityRepository {
    private CityDao cityDao;
    public CityRepositoryImpl(CityDao cityDao) {
        this.cityDao = cityDao;
    }
    public Optional<City> get(long id) {
        return cityDao.get(id);
    }
    public List<City> getAll() {
        return cityDao.getAll();
    }
    public void save(City city) {
        cityDao.save(city);
    }
    public void delete(City city) {
        cityDao.delete(city);
    }
}
```

Таке поєднання зустрічається дуже рідко, тому що більшість розробників використовують або шаблон DAO, або шаблон Repository. Spring Data пропонує величезний набір елементів для створення рішень на основі шаблону Repository, використовуючи тільки інтерфейси репозиторіїв.

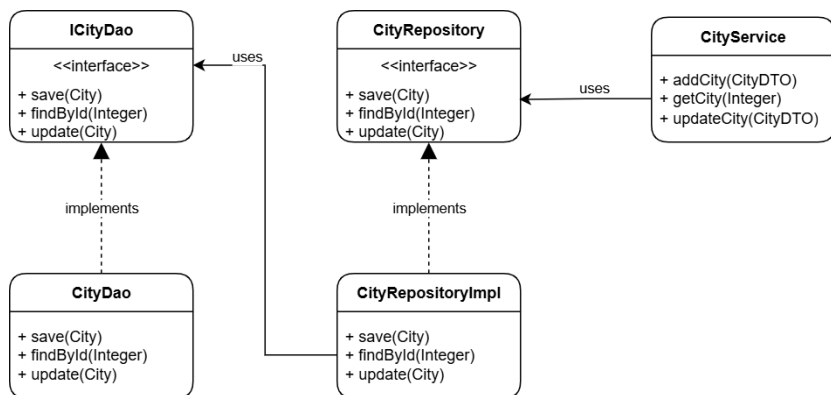


Рис. 3.7. Структура комбінування двох шаблонів

Data Transfer Object (DTO)

Шаблон об'єкта передачі даних (DTO) використовується не тільки у Java, а ще в декількох мовах програмування, агрегує та інкапсулює дані для передачі між шарами вашої програми або програм. Ви можете думати про цей шаблон як про структуру даних, яка не містить жодної бізнес-логіки і може представляти комбінацію структур.

Цей шаблон не є чимось, що з'явилося останнім часом. Мартін Фаулер уперше описав його ще у 2002 році у своїй книзі «Patterns of Enterprise Application Architecture». Про це також ідеться на його вебсторінці [11].

Цей шаблон допомагає скоротити кількість звернень до різних кінцевих точок або процесів для отримання всієї інформації, необхідної для виконання певного завдання, що може бути дорого з точки зору часу або розміру інформації для передачі.

Слід зауважити, що цей шаблон з'являється у випадках, коли класи передають сутності, що використовуються для зберігання інформації, між базою даних та іншими прикладними шарами.

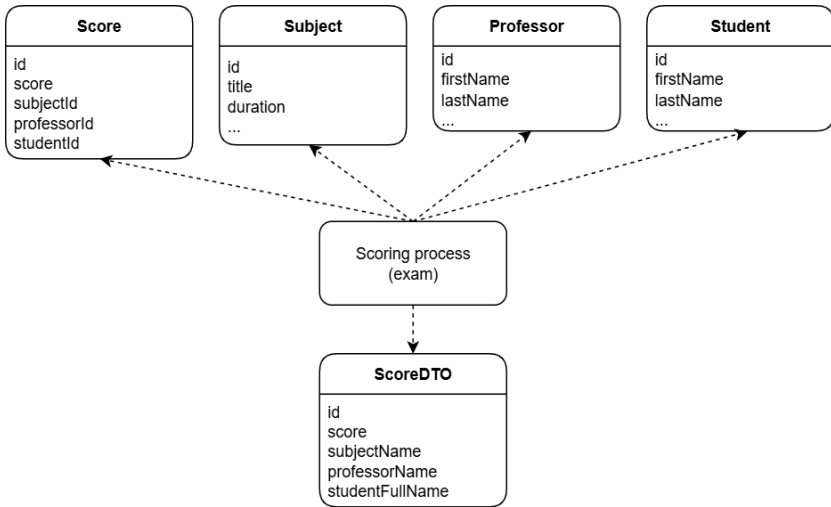


Рис. 3.8. Структура шаблону DTO

Шаблон Specification

Шаблон Specification фільтрує або вибирає підмножину об'єктів на основі критеріїв. Наприклад, ви хочете знати всіх студентів, які мають принаймні дві оцінки, збережені в системі, тому ви створюєте клас, який приймає кожного студента та перевіряє, чи відповідає цей студент умовам чи ні.

Мартін Фаулер уперше описав шаблон Specification більше 20 років тому. Сучасне трактування цього шаблону наведене на його сайті [11]. Але під час використання сучасних підходів, що реалізовані у Spring Data JPA, цей шаблон безпосередньо пов'язаний із шаблоном репозиторію, який використовується у Spring Data JPA для отримання даних з таблиці БД та фільтрації результатів запитів.

Ми розглянули типові шаблони роботи з персистентними даними. Звісно, існує набагато більше таких шаблонів, які розв'язують конкретні проблеми, наприклад, фільтрація результатів або збереження історичних даних.

Але розглянутих шаблонів буде цілком достатньо для розуміння рішень, що будуть використані далі у посібнику.

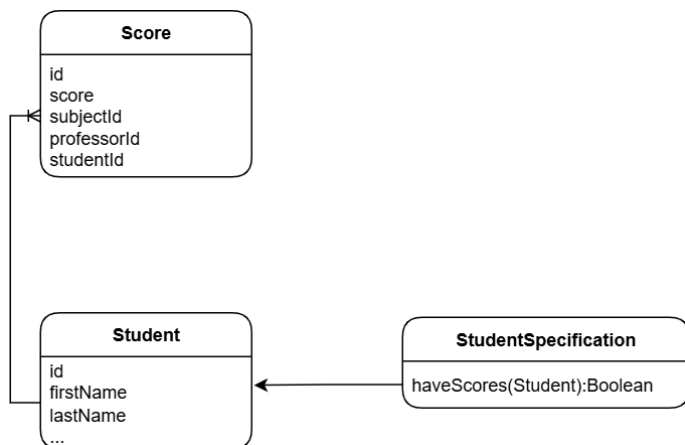


Рис. 3.9. Шаблон Specification

3.7. Створення інфраструктури застосунку Java для роботи з БД

Для початку треба створити каркас Spring Boot застосунку мовою Java.

Ініціалізація проєкту

Для створення нового застосунку більшість розробників рекомендують скористатися інтеграцією IntelliJ IDEA із використанням генератора Spring Initializr. На етапі ініціалізації доцільно задати наступні параметри:

- **Project SDK:** Java 17+ (на момент написання посібника, це – Java 25).
- **Group:** наприклад, ua.edu.nuos.
- **Artifact:** фактично – ім'я проєкту, наприклад, dreamuniversity.
- **Type:** Maven або Gradle відповідно до вподобань розробника.

До списку залежностей варто включити щонайменше: Spring Web – для реалізації вебфункціоналу; Spring Boot DevTools, Spring Data JPA, H2 Database або MySQL JDBC Driver.

Структура проєкту

Після генерації проєкту утворюється стандартна структура директорій:

- src/main/java – містить вихідний код застосунку;
- src/main/resources – призначено для конфігураційних файлів, зокрема application.properties, а також для розміщення статичних та динамічних компонентів web-застосунку;
- DreamUniversityApplication.java – головний клас, що містить точку входу в застосунок.

Реалізація контролера для пробного запуску

Як приклад, наведено базовий контролер, що обробляє HTTP-запити:

```
@RestController
public class HelloDemoController {
    @GetMapping("/hello")
    public String sayHello(
        @RequestParam(defaultValue = "World") String name) {
        return "Hello " + name + "!";
    }
}
```

Цей фрагмент демонструє використання REST-архітектури через анотації Spring Framework.

Запуск та тестування

Застосунок можна запустити безпосередньо з IntelliJ IDEA через виконання класу DreamUniversityApplication. У разі успішного запуску, його функціональність перевіряється за адресою: <http://localhost:8080/hello>.

Додаткові можливості

За потреби, можна одразу створити HTML-файл index.html та додати його до каталогу src/main/resources/static/, реалізуючи таким чином стартову сторінку застосунку.

Налаштування параметрів підключення до БД та JPA

У проєкті Spring Boot, який був згенерований автоматично, уже присутній файл налаштувань властивостей застосунку `application.properties`. Це звичайний текстовий файл, який треба доповнити записами про налаштування механізму персистентності та підключення до БД:

```
#JPA configuration
spring.jpa.generate-ddl=true
spring.jpa.show-sql=true
#Datasource configuration
spring.datasource.url=jdbc:mysql://localhost/univer?useSSL=false
spring.datasource.username=student
spring.datasource.password=1234
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

Звісно, налаштування URL бази даних, імені та пароля користувача можуть відрізнятися і повинні співпадати з відповідними параметрами підключення до СУБД, що встановлена на комп'ютері розробника або доступна з нього за URL.

3.8. Розробка компонентів для роботи із БД

Як уже було показано у п. 3.3, спочатку треба створити класи сутностей для БД. Існує два підходи до створення класів-сутностей та таблиць БД:

1. **Database first** – спочатку створюється схема БД з усіма таблицями та зв'язками між ними, а потім, використовуючи інструменти середовища розробки, створити відповідні до них класи-сутності.

2. **JPA first** – спочатку створюються класи-сутності, які моделюють предметну область, а застосунок налаштовується таким чином, щоб під час його старту, автоматично створювалась необхідна схема БД з усіма таблицями та зв'язками.

Розглянемо перший підхід – «Database first». Оскільки логічна модель БД уже створена, на основі неї створимо фізичну модель БД, указавши для полів таблиць типи даних відповідної СУБД. У нашому випадку – MySQL.

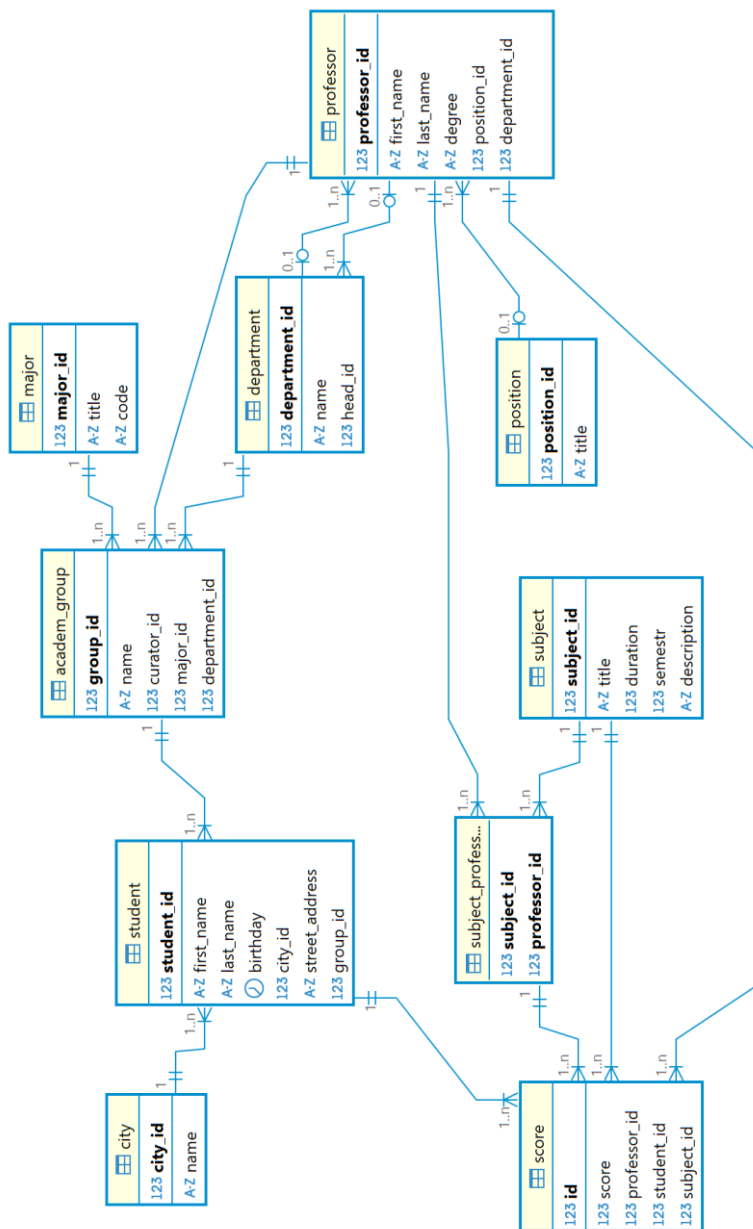


Рис. 3.10. Фізична модель даних

Далі, за допомогою інструмента JPA Buddy створимо класи сутностей.

Наша БД містить 10 таблиць, одна з яких є такою, що використовується як «Join Table» для реалізації зв'язку «багато до багатьох» між таблицями PROFESSOR та SUBJECT. Тому, треба буде створити 9 класів сутностей:

```
City  
Major  
Position  
Professor  
Department  
AcademGroup  
Student  
Subject  
Score
```

Для розміщення цих класів, у структурі проєкту треба створити пакет для них. Наприклад, **data**:

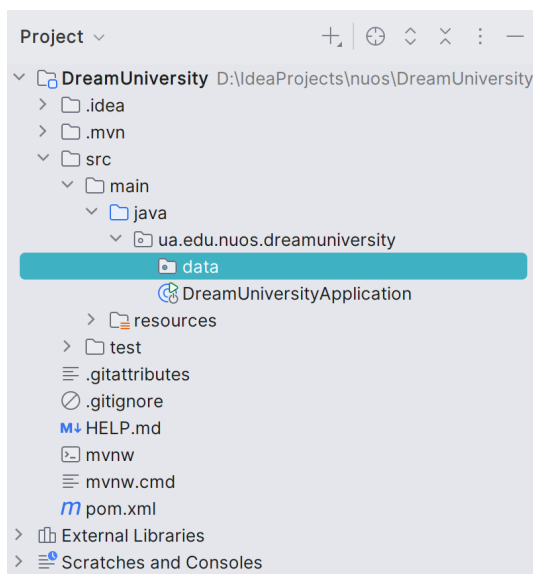


Рис. 3.11. Структура проєкту з пакетом data

Далі, треба за допомогою плагіна JPA Buddy створити й налаштувати класи-сутностей для бази даних:

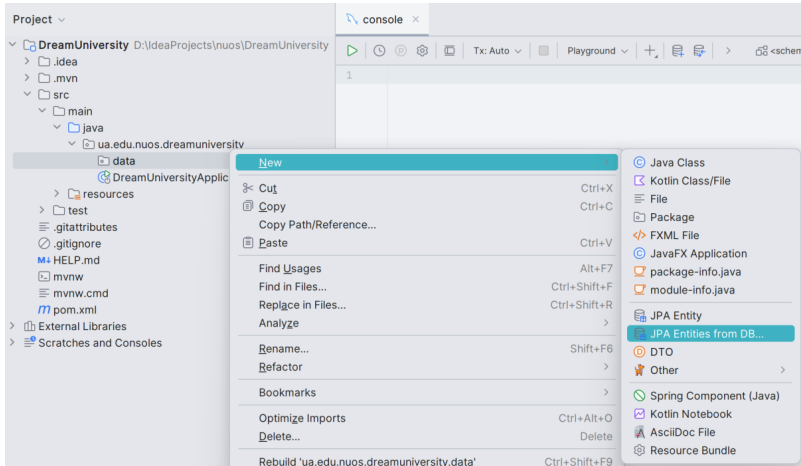


Рис. 3.12. Додавання класів сутностей

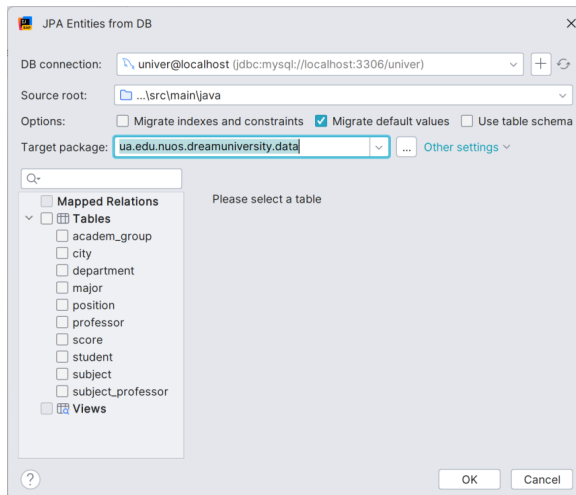


Рис. 3.13. Обрання таблиць для генерації класів-сутностей

У прикладі, що розглядається, треба обрати створення сутностей для всіх таблиць, крім таблиці `subject_professor`. У результаті будуть створені класи-сутності, які після відповідних налаштувань, будуть мати такий вигляд:

```
//                                Клас City
@Getter
@Setter
@Entity
@NoArgsConstructor
@Table(name = "city")
public class City {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "city_id", nullable = false)
    private Integer id;

    @Column(name = "name", nullable = false, length = 125)
    private String name;

    @OneToMany(mappedBy = "city", orphanRemoval = true)
    private Set<Student> students = new LinkedHashSet<>();

    public City(String name) {
        this.name = name;
    }
}
```

```
//                                Клас Major
@Getter
@Setter
@Entity
@NoArgsConstructor
@Table(name = "major")
public class Major {
    @Id
    @Column(name = "major_id", nullable = false)
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(name = "title", length = 120)
    private String title;

    @Column(name = "code", length = 5)
    private String code;

    @OneToMany(mappedBy = "major", orphanRemoval = true)
    private Set<AcademGroup> academGroups = new LinkedHashSet<>();
}
```

```
public Major(String title, String code) {
    this.title = title;
    this.code = code;
}
}
```

```
//                                     Клас Position
@Getter
@Setter
@Entity
@NoArgsConstructor
@Table(name = "position")
public class Position {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "position_id", nullable = false)
    private Integer id;

    @Column(name = "title", nullable = false, length = 125)
    private String title;

    @OneToMany(mappedBy = "position")
    private Set<Professor> professors = new LinkedHashSet<>();

    public Position(String title) {
        this.title = title;
    }
}
```

```
//                                     Клас Professor
@Getter
@Setter
@Entity
@Table(name = "professor")
@NoArgsConstructor
public class Professor {
    @Id
    @Column(name = "professor_id", nullable = false)
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(name = "first_name", nullable = false, length = 20)
    private String firstName;

    @Column(name = "last_name", nullable = false, length = 40)
    private String lastName;

    @Column(name = "degree", length = 100)
    private String degree;
}
```

```

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "position_id")
    private Position position;

    @ManyToMany(mappedBy = "professors")
    private Set<Subject> subjects = new LinkedHashSet<>();

    @OneToMany(mappedBy = "curator", orphanRemoval = true)
    private Set<AcademGroup> academGroups = new
LinkedHashSet<>();

    @OneToOne(mappedBy = "head", orphanRemoval = true)
    private Department departmentHeaded;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;

    public Professor(
        String firstName,
        String lastName,
        String degree,
        Position position,
        Department department) {

        this.firstName = firstName;
        this.lastName = lastName;
        this.degree = degree;
        this.position = position;
        this.department = department;
    }
}

```

```

//                                     Клас Department
@Getter
@Setter
@Entity
@NoArgsConstructor
@Table(name = "department")
public class Department {
    @Id
    @Column(name = "department_id", nullable = false)
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(name = "name", nullable = false, length = 200)
    private String name;

    @OneToMany(mappedBy = "department", orphanRemoval = true)
    private Set<AcademGroup> academGroups = new LinkedHashSet<>();
}

```

```

@OneToOne
@JoinColumn(name = "head_id")
private Professor head;

@OneToMany(mappedBy = "department", orphanRemoval = false)
private Set<Professor> professors = new LinkedHashSet<>();

public Department(String name) {
    this.name = name;
}
}

```

```

//                                     Клас AcademGroup
@Getter
@Setter
@Entity
@NoArgsConstructor
@Table(name = "academ_group")
public class AcademGroup {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "group_id", nullable = false)
    private Integer id;

    @Column(name = "name", nullable = false, length = 10)
    private String name;

    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "curator_id", nullable = false)
    private Professor curator;

    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "major_id", nullable = false)
    private Major major;

    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "department_id", nullable = false)
    private Department department;

    @OneToMany(mappedBy = "group", orphanRemoval = false)
    private Set<Student> students = new LinkedHashSet<>();

    public AcademGroup(
        String name,
        Major major,
        Professor professor,
        Department department) {

        this.name = name;
        this.major = major;
        this.curator = professor;
    }
}

```

```
        this.department = department;
    }
}
```

```
//                                     Клас Student
@Getter
@Setter
@Entity
@NoArgsConstructor
@Table(name = "student")
public class Student {
    @Id
    @Column(name = "student_id", nullable = false)
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(name = "first_name", nullable = false, length = 30)
    private String firstName;

    @Column(name = "last_name", nullable = false, length = 40)
    private String lastName;

    @Column(name = "birthday")
    private LocalDate birthday;

    @Column(name = "street_address", length = 200)
    private String streetAddress;

    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "group_id", nullable = false)
    private AcademGroup group;

    @ManyToOne
    @JoinColumn(name = "city_id")
    private City city;

    public Student(
        String firstName,
        String lastName,
        LocalDate date,
        City city,
        String address,
        AcademGroup group) {

        this.firstName = firstName;
        this.lastName = lastName;
        this.birthday = date;
        this.city = city;
        this.streetAddress = address;
        this.group = group;
    }
}
```

```
//                                     Клас Subject
@Getter
@Setter
@Entity
@NoArgsConstructor
@Table(name = "subject")
public class Subject {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "subject_id", nullable = false)
    private Integer id;

    @Column(name = "title", nullable = false, length = 100)
    private String title;

    @Column(name = "duration")
    private Integer duration;

    @Column(name = "semester")
    private Integer semester;

    @Column(name = "description", length = 200)
    private String description;

    @ManyToMany
    @JoinTable(name = "subject_professor",
        joinColumns = @JoinColumn(name = "subject_id"),
        inverseJoinColumns = @JoinColumn(name = "professor_
id"))
    private Set<Professor> professors = new LinkedHashSet<>();

    public Subject(
        String title,
        String description,
        Integer semester,
        Integer duration) {

        this.title = title;
        this.description = description;
        this.semestr = semester;
        this.duration = duration;
    }
}
```

```
//                                     Клас Score
@Getter
@Setter
@Entity
@Table(name = "score")
public class Score {
    @Id
```

```
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "score_id", nullable = false)
private Integer id;

@Column(name = "score")
private Integer score;

@ManyToOne(fetch = FetchType.LAZY, optional = false)
@JoinColumn(name = "student_id", nullable = false)
private Student student;

@ManyToOne(fetch = FetchType.LAZY, optional = false)
@JoinColumn(name = "subject_id", nullable = false)
private Subject subject;

@ManyToOne(fetch = FetchType.LAZY, optional = false)
@JoinColumn(name = "professor_id", nullable = false)
private Professor professor;
}
```

За допомогою інструменту Database tool, що є складовою частиною середовища IntelliJ IDEA, побудуємо діаграму класів-сутностей та проаналізуємо ці класи та зв'язки між ними.

Проведемо аналіз цих класів. Для зменшення кількості коду було задіяно бібліотеку Lombok. За рахунок використання анотацій `@Getter` та `@Setter` вдалось позбавитись від «спагетті-коду» – написаних уручну геттерів та сеттерів для полів класів. Тепер вони будуть генеруватися під час компіляції проекту. Крім того, анотація `@NoArgsConstructor` дозволяє автоматично створити конструктор без параметрів для тих класів, де за правилами бізнес-логіки потрібен конструктор з параметрами. Важливо пам'ятати, що відповідно до специфікації JPA, клас сутності (що має анотацію `@Entity`) повинен обов'язково мати конструктор без параметрів. Анотація `@Table` дозволяє встановити зв'язок класу-сутності, що створюється, з таблицею, до якої ми його прив'язуємо.

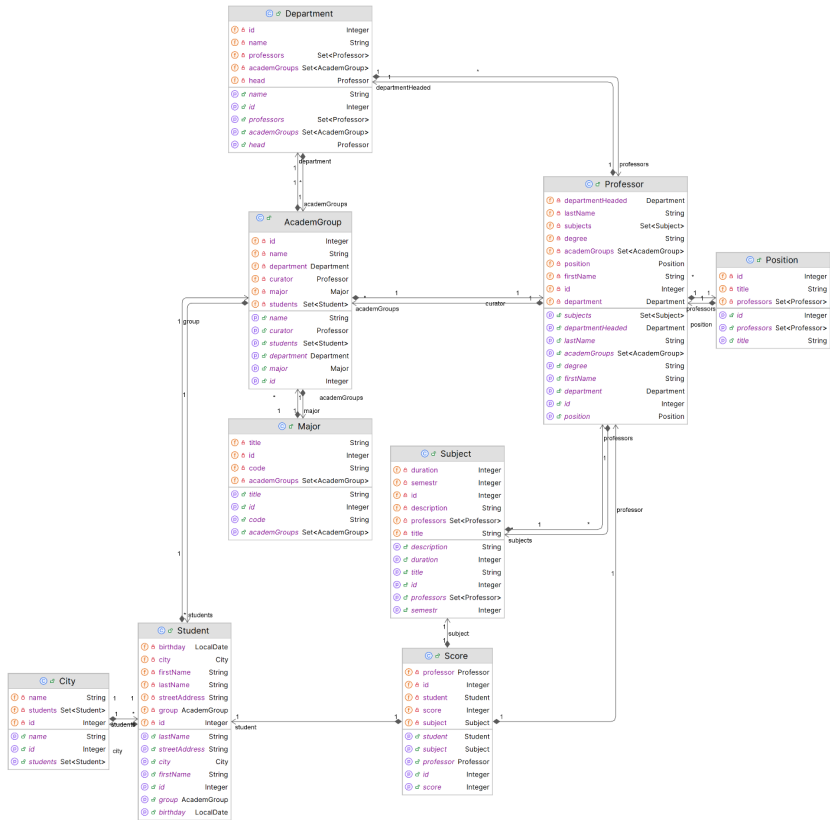


Рис. 3.14. Діаграма класів-сутностей

Тепер розглянемо анотації полів. Як уже було зазначено у п. 3.3, анотація `@Id` визначає поле, що буде відображатись у колонку первинного ключа таблиці. Анотація `@GeneratedValue(strategy = GenerationType.IDENTITY)` указує, що значення поля первинного ключа буде генеруватись, як автоінкрементне, оскільки така можливість підтримується в СУБД, що використовується (MySQL).

Анотація `@Column` установлює зв'язок поля класу-сутності з іменем стовпця у таблиці в БД а також указує

додаткові обмеження: довжина поля та його обов'язковість (чи може бути null).

Анотації `@OneToMany`, `@ManyToOne` та `@ManyToMany` використовуються для описів зв'язків та обмежень. Розглянемо їх докладніше.

Анотація `@OneToMany`

Розглянемо використання анотації `@OneToMany` на прикладі зв'язку між сутностями `Position` та `Professor`. Очевидно, що для кожної посади можна визначити список викладачів, які обіймають її. Тому ця анотація потрібна для того, щоб асоціація була двонаправленою. У цьому випадку нам також потрібно встановити параметр `mappedBy`. Аргументом є назва властивості на «іншій стороні» – у класі `Professor`. За замовчуванням, стратегію автоматичного отримання даних зі зв'язаної таблиці бази буде встановлено як `LAZY`:

```
@OneToMany(mappedBy = "position")
private Set<Professor> professors = new LinkedHashSet<>();
```

Анотація `@ManyToOne`

Очевидно, що для кожного викладача треба визначити властивість, що буде встановлювати зв'язок з його посадою. Таким чином, клас-сутність `Professor` буде містити змінну типу `Position` із відповідною анотацією `@ManyToOne`:

```
@ManyToOne
@JoinColumn(name = "department_id")
private Department department;
```

Разом з анотацією `@ManyToOne` використовується анотація `@JoinColumn` з параметром `name`, що визначає ім'я колонки таблиці, яка буде містити значення зовнішнього ключа для зв'язку з таблицею з «іншого боку».

Анотація @ManyToMany

Анотація @ManyToMany дозволяє описати зв'язки, у яких кожен об'єкт однієї сутності може відповідати багатьом об'єктам іншої, і навпаки. Для цього створюється проміжна таблиця, яка зберігає пари ідентифікаторів пов'язаних об'єктів. Наприклад, така анотація часто використовується для зв'язку між сутностями Professor і Subject, оскільки викладач може викладати кілька предметів, а предмет також може читатися багатьма викладачами. У цьому випадку для визначення проміжної таблиці використовується анотація @JoinTable, яка дозволяє явно задати імена таблиці та її колонок.

Завдяки гнучкості анотацій JPA можна детально налаштувати відображення зв'язків у моделях даних, контролювати каскадне оновлення або видалення пов'язаних записів, а також визначати стратегії отримання пов'язаних об'єктів (LAZY або EAGER). Це дає змогу ефективно моделювати складні структури баз даних відповідно до вимог реального застосування.

Особливо важливою перевагою JPA-анотацій є можливість уникнути надмірного дублювання коду: більшість аспектів зв'язків між сутностями можна централізовано описати через конфігураційні параметри у відповідних анотаціях. Наприклад, можна налаштувати тип каскадування (CascadeType), щоб автоматично зберігати або видаляти пов'язані об'єкти, або встановити, чи допускається можливість установлення null для зовнішнього ключа.

Крім того, завдяки параметрам fetch та cascade, можна тонко керувати продуктивністю застосунку та послідовністю змін у базі даних. Правильне налаштування цих аспектів дозволяє уникнути зайвих запитів до бази та забезпечити цілісність даних навіть у складних доменних моделях.

Для оптимізації роботи з великими обсягами даних часто застосовують додаткові параметри, наприклад,

orphanRemoval, який дозволяє автоматично видаляти об'єкти, що більше не пов'язані із батьківською сутністю:

```
@OneToMany(mappedBy = "department", orphanRemoval = true)
private Set<AcademGroup> academGroups = new LinkedHashSet<>();
```

Наведений фрагмент коду із класу-сутності Department визначає, що при видаленні кафедри, академічні групи, які за нею закріплені, будуть видалені (розформовані).

```
@OneToMany(mappedBy = "group", orphanRemoval = false)
private Set<Student> students = new LinkedHashSet<>();
```

У той же час, при видаленні групи, видаляти студентів не треба. Бізнес-логіка застосунку може або забороняти таку дію, або пропонувати адміністратору спочатку виконати переведення студентів до інших груп.

Такі гнучкі налаштування роблять застосування JPA максимально ефективним для розробників, які будують складну бізнес-логіку на основі реляційних баз даних.

Наступним етапом, після створення персистентних класів-сутностей, є створення компонентів для доступу до даних – інтерфейсів Repository. Як уже було описано вище, використовуючи Spring Data JPA, розробнику достатньо створити самі репозиторії та описати у них методи, за допомогою яких будуть виконуватися запити у БД. А реалізації цих методів Spring Data створить автоматично під час компіляції проекту. Зауважимо, що для створення найпростішого репозиторію треба лише знати тип самого персистентного класу-сутності, для якого створюється репозиторій та тип його первинного ключа. Навіть порожній інтерфейс, що успадкований від CrudRepository, як впливає з його назви, уже дає можливості виконання базових CRUD-операцій.

Використовуючи Spring Data JPA, розробник може створювати власні інтерфейси репозиторіїв на основі одного із стандартних інтерфейсів. Розглянемо їх тут:

- `CrudRepository` – найпростіший інтерфейс, що дозволяє виконувати основні CRUD-операції (створення, читання, оновлення, видалення) над сутностями. Його достатньо для більшості базових сценаріїв роботи з даними.
- `PagingAndSortingRepository` розширює можливості `CrudRepository`, додаючи підтримку сортування та пагінації. Це особливо корисно під час роботи з великими наборами даних, коли необхідно отримувати дані частинами та впорядковувати результати за певними критеріями.
- `JpaRepository` – ще більш функціональний інтерфейс, який, окрім можливостей попередніх двох, пропонує додаткові методи для роботи з JPA, зокрема пакетне збереження, видалення списком та інші зручні інструменти для оптимізації доступу до даних.

Завдяки цим інтерфейсам розробники можуть швидко створювати репозиторії для будь-яких сутностей, не витрачаючи час на написання шаблонного коду. Крім того, Spring Data JPA надає гнучкий механізм генерації запитів на основі імен методів. Наприклад, метод `findByLastName(String lastName)` автоматично буде інтерпретований як пошук усіх записів із відповідним прізвищем.

Розглянемо клас-сутність `Subject`, який представляє навчальну дисципліну. У відповідному репозиторії для нього, крім стандартних CRUD-методів, можна визначити різноманітні додаткові методи репозиторію.

```
Optional<Subject> findByTitle(String title);
```

Повертає об'єкт `Optional<Subject>`, який містить об'єкт `Subject`, назва якого точно співпадає з наведеною назвою.

Повертає порожній об'єкт Optional, якщо збіг не знайдено. Пошук відбувається з урахуванням реєстру.

```
List<Subject> findByDurationGreaterThanOrEqualTo(Integer minDuration);
```

Повертає список предметів, обсяг яких (у годинах) перевищує вказаний мінімум. Повертає порожній список, якщо всі предмети мають обсяг, що не перевищує вказаний.

```
List<Subject> findByProfessorsDepartmentNameContainsIgnoreCase(String name);
```

Шукає предмети, які «читають» викладачі вказаної кафедри. Пошук відбувається без урахування реєстру. Повертає список відповідних предметів.

```
List<Subject> findByTitleContaining(String pattern);
```

Повертає список предметів, заголовки яких містить заданий зразок. Пошук відбувається з урахуванням реєстру. Повертає порожній список, якщо не знайдено збігів.

```
List<Subject> findByProfessorsLastNameLikeIgnoreCase(String lastName);
```

Знаходить предмети, які викладають професори, прізвища яких відповідають зразку. Пошук відбувається без урахування реєстру. Використовує зіставлення зі зразком за правилами оператора SQL LIKE.

```
boolean existsByTitle(String title);
```

Повертає true, якщо існує предмет із назвою, що точно співпадає з указаною. В іншому випадку повертає false. Перевірка виконується з урахуванням реєстру.

```
Long countByProfessorsDepartment(Department department);
```

Рахує, скільки предметів викладають викладачі зазначеної кафедри. Повертає count у вигляді значення типу long.

```
List<Subject> findBySemester(Integer semester);
```

Повертає список усіх предметів, що вивчаються у вказаному семестрі. Повертає порожній список, якщо збігів не знайдено.

```
List<Subject> findBySemesterAndDurationBetween(Integer semester, Integer minDuration, Integer maxDuration);
```

Знаходить предмети в конкретному семестрі з тривалістю в межах заданого діапазону. Використовується закритий діапазон, тобто такий, що включає в себе кінцеві значення. Повертає відповідні теми у вигляді списку.

```
List<Subject> findTop3ByOrderByDurationDesc();
```

Знаходить до 3 предметів із найбільшим обсягом. Результати впорядковані за тривалістю в порядку спадання. Повертає список з максимальною кількістю у 3 елементи.

```
List<Subject> findByDurationBetween(Integer minDuration, Integer maxDuration);
```

Повертає предмети з тривалістю в межах вказаного діапазону. Діапазон включає в себе кінцеві значення. Повертає результати у вигляді списку.

```
List<Subject> findByProfessorsDepartmentAndSemester(Department department, Integer semester);
```

Знаходить предмети, які викладають викладачі зазначеної кафедри в даному семестрі. Повертає відповідні

предмети у вигляді списку. Повертає порожній список, якщо не знайдено збігів.

Таким чином, інтерфейс `SubjectRepository` буде мати такий вигляд:

```
public interface SubjectRepository extends JpaRepository<Subject, Integer> {
    Optional<Subject> findByTitle(String title);

    List<Subject> findByDurationGreaterThan(Integer minDuration);

    List<Subject> findByProfessorsDepartmentNameContainsIgnoreCase(String name);

    List<Subject> findByTitleContaining(String pattern);

    List<Subject> findByProfessorsLastNameLikeIgnoreCase(String lastName);

    boolean existsByTitle(String title);

    long countByProfessorsDepartment(Department department);

    List<Subject> findBySemester(Integer semester);

    List<Subject> findBySemesterAndDurationBetween(Integer semester, Integer minDuration, Integer maxDuration);

    List<Subject> findTop3ByOrderByDurationDesc();

    List<Subject> findByDurationBetween(Integer minDuration, Integer maxDuration);

    List<Subject> findByProfessorsDepartmentAndSemester(Department department, Integer semester);
}
```

Можливість розширення та налаштування під потреби розробника

Якщо стандартних методів недостатньо, розробник може додати власні методи до інтерфейсу репозиторію. Це можна зробити за допомогою утворення імен методів на основі спеціальних конструкцій, як було показано вище. У випадку, коли написати ім'я метода складно, або неможливо, або з метою покращення читаності коду, можна

скористатися анотацією `@Query` для написання власних JPQL- чи SQL-запитів.

Розглянемо деякі типові запити мовою JPQL та порівняємо їх із запитами, що написані «магічними» словами у іменах методів.

```
@Query("select s from Subject s where s.title = ?1")
Optional<Subject> findByTitle(String title);

@Query("select s from Subject s where s.duration > ?1")
List<Subject> findByDurationGreaterThan(Integer minDuration);

@Query("""
    select s from Subject s inner join s.professors professors
    where upper(professors.department.name) like
    upper(concat('%', ?1, '%'))""")
List<Subject> findByProfessorsDepartmentNameContainsIgnoreCase
(String name);

@Query("select s from Subject s where s.title like concat('%', ?1,
'%')")
List<Subject> findByTitleContaining(String pattern);

@Query("select s from Subject s inner join s.professors professors
where upper(professors.lastName) like upper(?1)")
List<Subject> findByProfessorsLastNameLikeIgnoreCase(String
lastName);

@Query("select (count(s) > 0) from Subject s where s.title = ?1")
boolean existsByTitle(String title);

@Query("select count(s) from Subject s inner join s.professors
professors where professors.department = ?1")
Long countByProfessorsDepartment(Department department);

@Query("select s from Subject s where s.semester = ?1")
List<Subject> findBySemester(Integer semester);

@Query("select s from Subject s where s.semester = ?1 and s.duration
between ?2 and ?3")
List<Subject> findBySemesterAndDurationBetween(Integer semester,
Integer minDuration, Integer maxDuration);

@Query("select s from Subject s where s.duration between ?1 and ?2")
List<Subject> findByDurationBetween(Integer minDuration, Integer
maxDuration);
```

Як можна побачити, майже кожний запит, що сформульований як похідний метод (derivative method) і записаний «магічними» словами, можна переформулювати у вигляді JPQL-запиту. Це може бути зручним, якщо ім'я методу виходить не дуже зрозумілим для читання та супроводження. У такому випадку, кращим рішенням може бути додавання відповідного запиту у вигляді анотації до метода.

Розглянувши наступний метод та його анотацію, можна помітити, що запит «Визначити усі навчальні дисципліни, що викладаються вказаною кафедрою у визначеному семестрі», виконує складний запит до чотирьох таблиць у базі даних, що не є очевидним просто з імені метода. Таким чином, додавши анотацію, розробник може покращити розуміння наступного коду для читання та модифікації у майбутньому.

```
@Query("""
    select s from Subject s inner join s.professors professors
    where professors.department = ?1 and s.semester = ?2""")
List<Subject> findByProfessorsDepartmentAndSemester(Department
department, Integer semester);
```

Звісно, існують такі похідні методи, для яких неможливо написати відповідну анотацію із запитом JPQL.

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. У чому полягає основна проблема «невідповідності парадигм» між ООП та реляційними БД?
2. Які проблеми виникають під час відображення асоціацій між об'єктами на зв'язки між таблицями?
3. Опишіть проблеми навігації по даних у контексті ORM.
4. Що таке ORM і які його основні переваги?
5. Яке співвідношення між JPA, Hibernate та Spring Data?
6. У чому полягає основна ідея Spring Data для спрощення роботи з персистентністю?

7. Що таке «єдиниця персистентності» (Persistence Unit) у JPA та як вона налаштовується?
8. Які є два основні способи конфігурації JPA/Hibernate?
9. Назвіть переваги та недоліки використання «чистого» JDBC, JPA/Hibernate та Spring Data JPA.
10. Які анотації є обов'язковими для оголошення класу сутності в JPA?
11. Яка різниця між поняттями «ідентичність» (identity) та «рівність» (equality) об'єкта в Java та чому це важливо для JPA?
12. Для чого використовуються анотації @Table та @Column?
13. Яку роль відіграють анотації @Transient, @Temporal та @Enumerated?
14. Яка основна мета шаблону DAO та чим він відрізняється від шаблону Repository?
15. Що таке DTO та у яких випадках його використання є доцільним?
16. Для вирішення яких завдань призначений шаблон Specification?
17. Які ключові компоненти необхідні для організації інфраструктури доступу до даних у Java-застосунку?
18. Які анотації використовуються для відображення зв'язків «один до багатьох» та «багато до одного»?
19. Як організується двосторонній зв'язок між сутностями і які правила (mappedBy) для цього існують?
20. Опишіть основні способи реалізації зв'язку «багато до багатьох» за допомогою анотації @ManyToMany.

СПИСОК ЛІТЕРАТУРИ

1. Elmasri, R., Navathe, S. Fundamentals of database systems (7th ed.). Pearson, 2022. 1272 p.
2. Silberschatz, A., Korth, H. F., Sudarshan, S. Database system concepts (7th ed.). McGraw-Hill, 2022. 1376 p.
3. Coronel, C., Morris, S. Database systems: Design, implementation, & management (14th ed.). Cengage Learning, 2024. 688 p.
4. Petrov, A. Database internals: A deep dive into how distributed data systems work (2nd ed.). O'Reilly Media, 2021. 368 p.
5. C Mains, David. SQL in Action: Hands-On Techniques and Real-World Case Studies for Building, Querying, and Optimizing Fast, Scalable Databases. Independently Published, 2025. 226 p.
6. Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994. 395 p.
7. Tudose, C. Java persistence with Spring Data and Hibernate. Manning Publications, 2023. 616 p.
8. Cosmina, L., Deinum, M. Spring Data and Hibernate for enterprise applications. Apress, 2025. 460 p.
9. Deinum, M., Long, J. Spring Boot persistence best practices. Apress, 2022. 380 p.
10. Sharma, S. Mastering Spring Data JPA. Packt Publishing, 2024. 350 p.
11. Fowler, M. (n.d.). *Martin Fowler*. Retrieved November 25, 2025, from <https://martinfowler.com/>.

Навчальне видання

КНИРІК Наталя Ромуальдівна
БЕРКУНСЬКИЙ Євген Юрійович

**БАЗИ ДАНИХ У СУЧАСНИХ
КОРПОРАТИВНИХ ЗАСТОСУНКАХ**

Навчальний посібник

Комп'ютерна правка й коректура *О. Г. Костенко*
Комп'ютерне верстання *Ю. С. Семенченко*

Формат 60×84/16. Ум. друк. арк. 10,58. Тираж 100 прим. Вид. № 1. Зам. № 1409-78.

Видавець і виготівник Національний університет кораблебудування
імені адмірала Макарова

просп. Героїв України, 9, м. Миколаїв, 54007

E-mail : publishing@nuos.edu.ua

Свідоцтво суб'єкта видавничої справи ДК № 6402 від 19.09.2018 р.



ISBN 978-966-321-509-9



9 789663 215099



ВИДАВНИЦТВО
НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ
КОРАБЛЕБУДУВАННЯ
ІМ. АДМІРАЛА МАКАРОВА