Я 

Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

Лабораторна робота №2

Технології розроблення програмного забезпечення

«*Основи проектування*»

«10. VCS all-in-one»

Виконав
студент групи ІА–32:
Варивода К. С.

Перевірив:

Мягкий Михайло Юрійович

Київ 2025

Зміст

Зміст 2
Теоретичні відомості 2
Хід роботи 3
Діаграма варіантів 4
Сценарій 1 - Додавання репозиторію 5
Сценарій 2 - Вибір гілки та перегляд історії фіксацій 5
Сценарій 3 - Merge гілок 6
Діаграма класів 6
Проєктування БД 8
Висновки 14

Теоретичні відомості

UML є універсальною мовою графічного моделювання, яка призначена для опису, візуалізації, проєктування та документування компонентів програмного забезпечення, бізнес-процесів і різних типів систем. UML забезпечує строгий і потужний інструментарій для створення концептуальних, логічних і графічних моделей складних систем різного призначення. Вона об'єднала в собі напрацювання й переваги методів програмної інженерії, які успішно застосовувалися для моделювання масштабних і складних систем упродовж останніх років.

Діаграма – це графічне подання елементів моделі у вигляді пов'язаного графа, вершини та ребра якого мають визначене смислове навантаження. Основним засобом побудови моделей на основі UML є нотація стандартних діаграм.

Діаграма варіантів використання – це UML-діаграма, яка дозволяє у графічній формі подати вимоги до майбутньої системи. Вона виступає початковою концептуальною моделлю проєктованої системи та не деталізує її внутрішню структуру. Такі діаграми є вихідним пунктом у процесі збору вимог до програмного забезпечення та його реалізації. Вони створюються на аналітичному етапі розробки, допомагають бізнес-аналітикам сформувати повне уявлення про необхідний продукт і документувати його характеристики.

Діаграма варіантів використання складається з кількох основних елементів: прецедентів , акторів та відносин між ними.

Сценарії використання – це текстові описи процесів взаємодії користувачів із системою. Вони подаються як формалізовані покрокові інструкції, які описують

послідовність дій для досягнення певної мети. Такі сценарії однозначно визначають очікуваний результат.

Діаграми класів застосовуються в моделюванні програмних систем найчастіше. Вони належать до статичних моделей і відображають структуру системи з точки зору проєктування, показуючи класи, інтерфейси та взаємозв'язки між ними. При цьому діаграма класів не описує динамічної поведінки об'єктів.

Клас – це базовий елемент програмної системи. Він має пряме відображення в мовах програмування, що дозволяє автоматично генерувати програмний код або здійснювати реінжиніринг. Клас визначається назвою, атрибутами та операціями. На UML-діаграмі він зображається у вигляді прямокутника, розділеного на три секції: назва класу, його атрибути та операції, які може виконувати об'єкт класу.

Логічна модель бази даних – це структура, що містить таблиці, уявлення, індекси та інші логічні об'єкти бази даних, які забезпечують програмування й подальше використання БД. Створення логічної моделі є процесом проєктування бази даних, тісно пов'язаним із розробкою архітектури програмної системи, адже БД використовується для збереження даних, отриманих від програмних класів.

Хід роботи

Завдання

- Ознайомитись з короткими теоретичними відомостями.

- Проаналізувати тему та спроєктувати діаграму варіантів використання відповідно до обраної теми лабораторного циклу.

- Спроєктувати діаграму класів предметної області.

- Вибрати 3 варіанти використання та написати за ними сценарії використання.

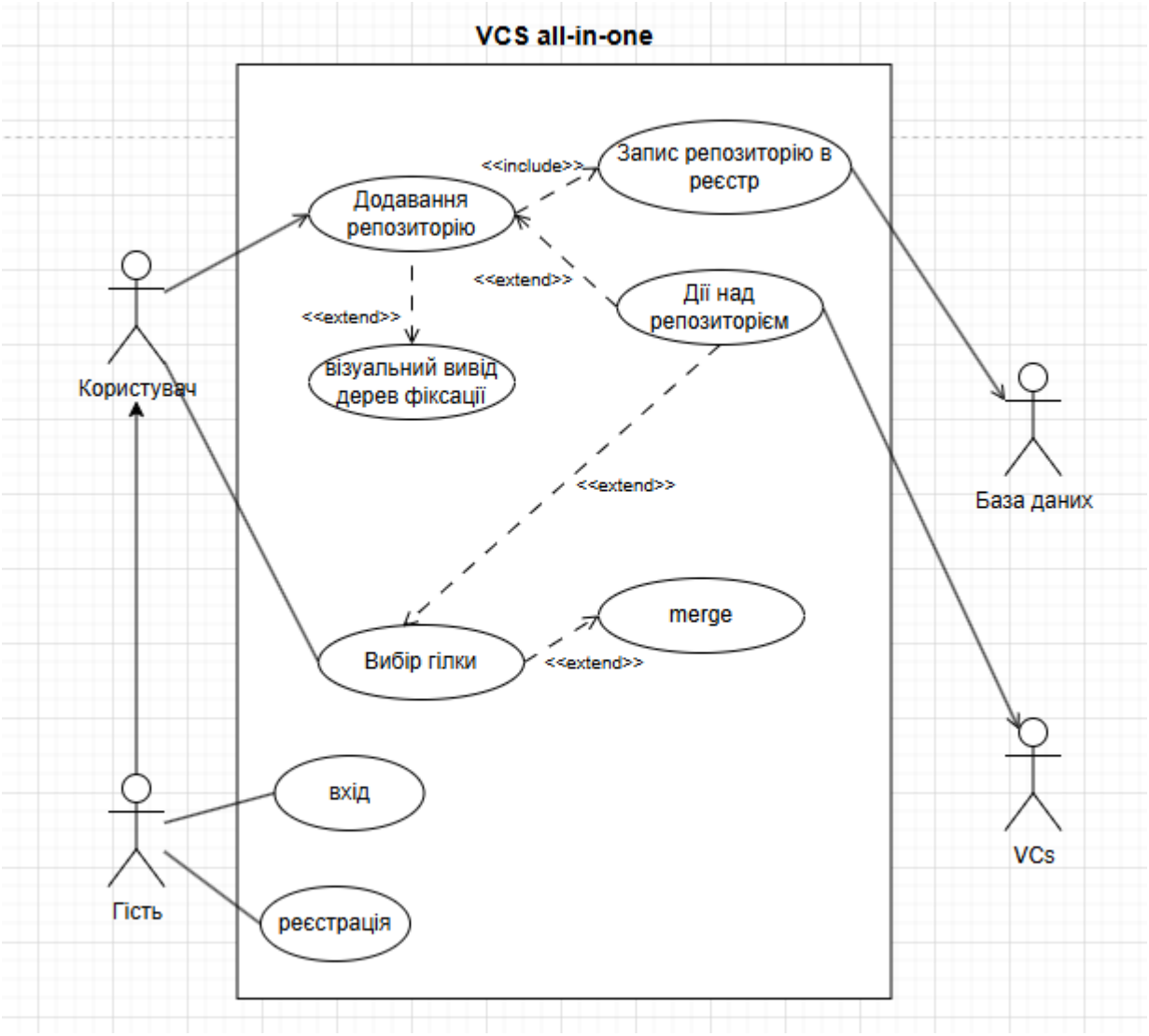
- На основі спроєктованої діаграми класів предметної області розробити основні класи та структуру бази даних системи. Класи даних повинні реалізувати шаблон Repository для взаємодії з базою даних.

- Нарисувати діаграму класів для реалізованої частини системи.

- Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт

повинен містити: діаграму варіантів використання відповідно, діаграму класів системи, вихідні коди класів системи, а також зображення структури бази даних.

Діаграма варіантів



Сценарій 1 - Додавання репозиторію

Актори: Користувач, VCS (Git/SVN/Mercurial), База даних

Передумови:

- Користувач увійшов у систему.
- Має доступ до існуючого репозиторію.

Основний перебіг подій:

1. Користувач обирає опцію **"Додати репозиторій"** у GUI.
2. Вказує шлях до локального репозиторію
3. Система визначає тип VCS (Git, SVN чи Mercurial).
4. Система виконує перевірку доступу.
5. Якщо доступ успішний, система зберігає інформацію про репозиторій у базі даних.
6. Система виводить репозиторій у списку проектів користувача.

Винятки:

- Якщо доступ до репозиторію неможливий → система показує повідомлення про помилку.
- Якщо користувач вказав неправильний тип→ запитати повторне введення.

Сценарій 2 - Вибір гілки та перегляд історії фіксацій

Актори: Користувач, VCS

Передумови:

- Репозиторій вже доданий у систему.

Основний перебіг подій:

1. Користувач відкриває потрібний репозиторій у GUI.
2. Обирає дію **"Вибрати гілку"**.
3. Система завантажує список доступних гілок з VCS.
4. Користувач вибирає потрібну гілку.
5. Система відображає **граф фіксацій** для цієї гілки.

Винятки:

- Якщо гілка не існує → повідомлення про помилку.
- Якщо немає доступу до VCS → повідомлення про неможливість отримати дані.

Сценарій 3 - Merge гілок

Актори: Користувач, VCS

Передумови:

- Користувач має доступ до репозиторію.
- Є хоча б дві гілки.

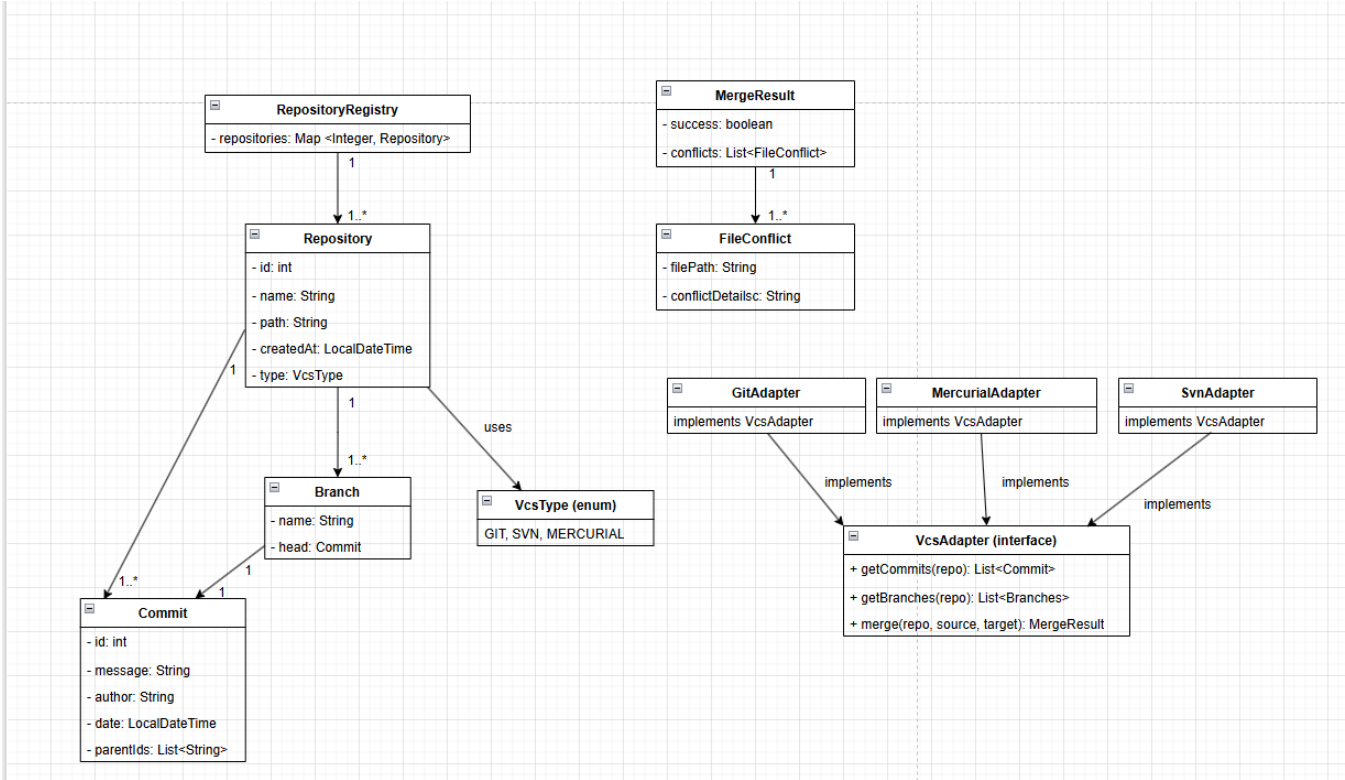
Основний перебіг подій:

1. Користувач у GUI обирає основну гілку та гілку для злиття.
2. Система надсилає команду merge до відповідного VCS.
3. Якщо конфліктів немає – система повідомляє про успішне злиття.
4. Якщо конфлікти є – система візуалізує їх у GUI (наприклад, підсвічування рядків у файлах).
5. Користувач вручну вирішує конфлікти та повторює merge.

Винятки:

- Якщо користувач не має прав на виконання merge → повідомлення про відмову.

Діаграма класів



1. **RepositoryFacade – IRepository**

- **Тип відносин:** Асоціація
- **Пояснення:** RepositoryFacade зберігає посилання на об’єкт типу IRepository (будь-який з Git/SVN/Mercurial). Це дозволяє викликати методи репозиторію без знання конкретної реалізації.

2. **RepositoryFacade – RepositoryFactory**

- **Тип відносин:** Агрегація
- **Пояснення:** RepositoryFacade використовує фабрику для створення конкретного репозиторію (Git, SVN або Mercurial). Фабрика може існувати самостійно.

3. **RepositoryFactory – IRepository**

- **Тип відносин:** Узагальнення (factory method)
- **Пояснення:** Фабрика створює об’єкти, які реалізують інтерфейс IRepository. Це інкапсулює процес створення і приховує від клієнта конкретний тип.

4. **GitRepository / SvnRepository / MercurialRepository – IRepository**

- **Тип відносин:** Узагальнення (implements)
- **Пояснення:** Кожен клас репозиторію реалізує спільний інтерфейс IRepository, забезпечуючи уніфікований контракт для операцій (commit, push, pull тощо).

5. **RepositoryIterator – IRepository**

- **Тип відносин:** Асоціація
- **Пояснення:** RepositoryIterator отримує дані з репозиторію через інтерфейс IRepository для ітерації по коммітах чи гілках. Це використання без "ціле-частина".

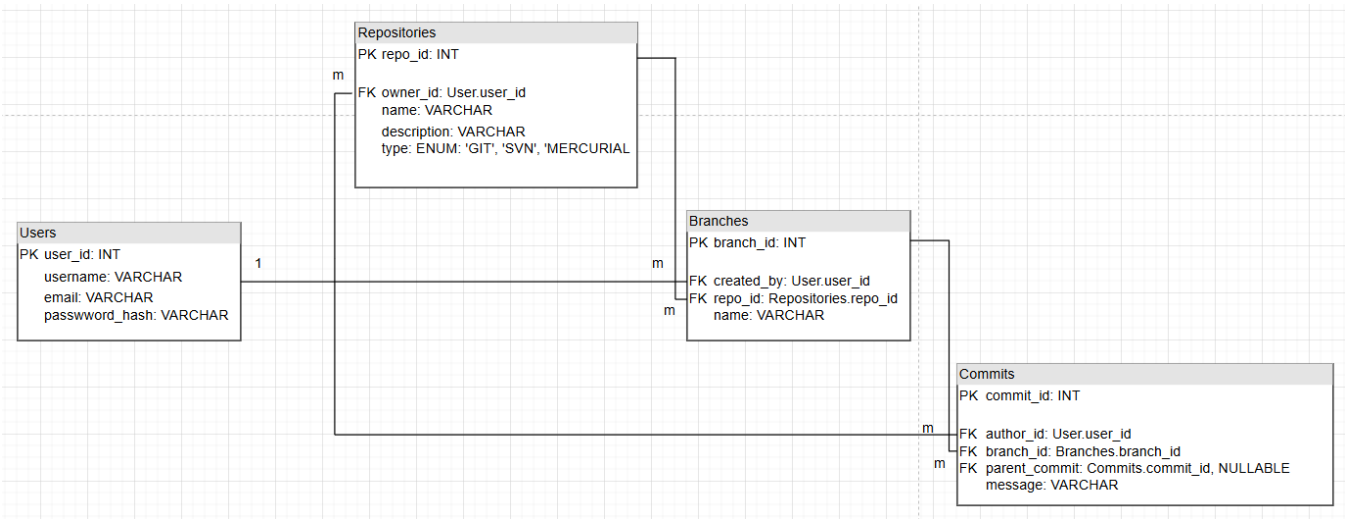
6. **RepositoryVisitor – IRepository**

- **Тип відносин:** Асоціація (патерн Visitor)
- **Пояснення:** RepositoryVisitor викликає методи IRepository, щоб "відвідати" його і виконати додаткові операції (наприклад, збір статистики чи перевірку коду).

7. **RepositoryIterator – Commit**

- **Тип відносин:** Агрегація
- **Пояснення:** Ітератор оперує множиною коммітів (Commit), але самі об’єкти коммітів можуть існувати незалежно.

Проєктування БД



Вихідні коди класів системи

```
public enum VcsType {
    GIT,
    SVN,
    MERCURIAL
}
```

```
public class Repository {
    private int id;
    private String name;
    private String path;
    private VcsType type;
    private LocalDateTime createdAt;

    public Repository(int id, String name, String path, VcsType type) {
        this.id = id;
```

```
        this.name = name;
        this.path = path;
        this.type = type;
        this.createdAt = LocalDateTime.now();
    }

    public int getId() { return id; }
    public String getName() { return name; }
    public String getPath() { return path; }
    public VcsType getType() { return type; }
    public LocalDateTime getCreatedAt() { return createdAt; }
}

public class Commit {
    private String id;
    private String message;
    private String author;
    private LocalDateTime date;
    private List<String> parentIds;

    public Commit(String id, String message, String author, LocalDateTime date,
List<String> parentIds) {
        this.id = id;
        this.message = message;
        this.author = author;
        this.date = date;
        this.parentIds = parentIds;
    }
```

```
    public String getId() { return id; }
    public String getMessage() { return message; }
    public String getAuthor() { return author; }
    public LocalDateTime getDate() { return date; }
    public List<String> getParentIds() { return parentIds; }
}

public class Branch {
    private String name;
    private Commit head;

    public Branch(String name, Commit head) {
        this.name = name;
        this.head = head;
    }

    public String getName() { return name; }
    public Commit getHead() { return head; }
}

public class FileConflict {
    private String filePath;
    private String conflictDetails;

    public FileConflict(String filePath, String conflictDetails) {
        this.filePath = filePath;
        this.conflictDetails = conflictDetails;
    }
```

```
    public String getFilePath() { return filePath; }
    public String getConflictDetails() { return conflictDetails; }
}

public class MergeResult {
    private boolean success;
    private List<FileConflict> conflicts;

    public MergeResult(boolean success, List<FileConflict> conflicts) {
        this.success = success;
        this.conflicts = conflicts;
    }

    public boolean isSuccess() { return success; }
    public List<FileConflict> getConflicts() { return conflicts; }
}


public interface VcsAdapter {
    List<Commit> getCommits(Repository repo);
    List<Branch> getBranches(Repository repo);
    MergeResult merge(Repository repo, Branch source, Branch target);
}


public class GitAdapter implements VcsAdapter {
    @Override
    public List<Commit> getCommits(Repository repo) {
```

```
        return List.of();
    }

    @Override
    public List<Branch> getBranches(Repository repo) {
        return List.of();
    }

    @Override
    public MergeResult merge(Repository repo, Branch source, Branch target) {
        return new MergeResult(true, List.of());
    }
}

public class SvnAdapter implements VcsAdapter {
    @Override
    public List<Commit> getCommits(Repository repo) {
        return List.of();
    }

    @Override
    public List<Branch> getBranches(Repository repo) {
        return List.of();
    }

    @Override
    public MergeResult merge(Repository repo, Branch source, Branch target) {
        return new MergeResult(true, List.of());
    }
```

```
}

public class MercurialAdapter implements VcsAdapter {
    @Override
    public List<Commit> getCommits(Repository repo) {
        return List.of();
    }

    @Override
    public List<Branch> getBranches(Repository repo) {
        return List.of();
    }

    @Override
    public MergeResult merge(Repository repo, Branch source, Branch target) {
        return new MergeResult(true, List.of());
    }
}


public class RepositoryRegistry {
    private Map<Integer, Repository> repositories = new HashMap<>();

    public void addRepository(Repository repo) {
        repositories.put(repo.getId(), repo);
    }

    public Repository getRepository(int id) {
        return repositories.get(id);
```

```
    }

    public List<Repository> getAllRepositories() {
        return List.copyOf(repositories.values());
    }
}
```

Висновки

під час виконання лабораторної роботи, ми навчилися основам проектування, обрали зручну систему побудови UML-діаграм та навчилися будувати діаграми варіантів використання для системи що проєктується, розробляти сценарії варіантів використання та будувати діаграми класів предметної області.

Контрольні питання

1. Що таке UML?

UML (Unified Modeling Language) - це уніфікована мова візуального моделювання, яка використовується для специфікації, візуалізації, проєктування та документування програмних систем. Вона надає стандартний спосіб представлення проєкту системи.

2. Що таке діаграма класів UML?

Це статична структурна діаграма, яка описує структуру системи, показуючи її класи, їх атрибути, методи та відносини між класами.

3. Які діаграми UML називають канонічними?

Канонічними діаграмами UML вважаються ті, що найчастіше використовуються для представлення різних аспектів системи. Зазвичай до них відносять діаграму варіантів використання, діаграму класів, діаграму послідовності, діаграму станів та діаграму діяльності.

4. Що таке діаграма варіантів використання?

Це діаграма, що описує взаємодію між користувачами та системою. Вона показує, які функції система надає акторам.

5. Що таке варіант використання?

Варіант використання - це опис послідовності дій, які виконує система у відповідь на запит актора для досягнення певної мети. Простими словами, це конкретна функція системи з погляду користувача.

6. Які відношення можуть бути відображені на діаграмі використання?

На діаграмі варіантів використання існують такі основні типи відношень:

Асоціація: зв'язок між актором та варіантом використання.

Включення: один варіант використання включає функціональність іншого.

Розширення: один варіант використання може розширювати функціональність іншого за певних умов.

Узагальнення: відношення спадкування між акторами або варіантами використання.

7. Що таке сценарій?

Сценарій - це конкретний екземпляр варіанта використання, що описує одну з можливих послідовностей подій. Наприклад, для варіанта використання "Оплата товару" сценаріями можуть бути "Успішна оплата" та "Не вдалося здійснити оплату".

8. Що таке діаграма класів?

Це статична діаграма, що ілюструє структуру системи, показуючи класи, їх атрибути, операції та відносини між ними. Це "креслення" програмної системи.

9. Які зв’язки між класами ви знаєте?

Основні типи зв'язків між класами:

Асоціація: загальний зв'язок між двома класами.

Агрегація: відношення "частина-ціле", де частина може існувати без цілого.

Композиція: сильна форма агрегації, де частина не може існувати без цілого.

Узагальнення/Спадкування: відношення "є різновидом".

Залежність: один клас використовує інший.

Реалізація: клас реалізує інтерфейс.

10. Чим відрізняється композиція від агрегації?

Основна відмінність полягає у часі життя та володінні.

Агрегація: Об'єкт-частина може існувати незалежно від об'єкта-контейнера.

Композиція: Об'єкт-частина не може існувати без об'єкта-контейнера. При знищенні контейнера знищуються і його частини.

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

На діаграмах класів вони позначаються лінією зі стрілкою у вигляді ромба з боку "цілого":

Агрегація: ромб незафарбований

Композиція: ромб зафарбований

12. Що являють собою нормальні форми баз даних?

Нормальні форми - це набір правил і вимог до структури реляційної бази даних, метою яких є зменшення надлишковості даних та усунення аномалій

13. Що таке фізична модель бази даних? Логічна?

Логічна модель даних: Описує дані та зв'язки між ними незалежно від конкретної системи управління базами даних. Вона фокусується на тому, що являють собою дані

Фізична модель даних: Описує конкретну реалізацію логічної моделі в обраній СУБД. Вона визначає як дані будуть зберігатися. Типи даних, індекси, таблиці, обмеження

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

Зазвичай існує пряма відповідність, яку описує технологія ORM (Object-Relational Mapping):

Клас відповідає таблиці в базі даних.

Об'єкт відповідає рядку в таблиці.

Атрибут класу відповідає стовпцю в таблиці.