

1. Customer's Request (Yêu cầu của khách hàng): Khởi đầu từ khách hàng gửi yêu cầu qua HTTP đến máy tính (Computer).

2. Computer: Nhận yêu cầu từ khách hàng và chuyển tiếp đến Controller.

3. Controller:

- Nhận yêu cầu (Step 1).

- Gửi yêu cầu dữ liệu đến Model (Step 2).

- Nhận dữ liệu từ Model (Step 3).

- Gửi kết quả đến View (Step 4).

4. Model:

- Xử lý yêu cầu dữ liệu từ Controller.

- Lấy dữ liệu từ Database (Step 2).

- Trả dữ liệu về Controller (Step 3).

- Đại diện cho lõi logic kinh doanh và dữ liệu.

5. Database (Cơ sở dữ liệu):

- Xử lý yêu cầu dữ liệu từ Model.

- Cung cấp dữ liệu cần thiết cho Model.

6. View:

- Nhận kết quả từ Controller (Step 4).

- Chuyển đổi kết quả thành định dạng HTML/JSON (Step 5).

- Trả về phản hồi cho khách hàng.

Tóm lại, mô hình này thể hiện cách các thành phần (Model, View, Controller) phối hợp để xử lý yêu cầu, lấy dữ liệu từ cơ sở dữ liệu và hiển thị kết quả cho khách hàng theo quy trình tuần tự.

    MÔ HÌNH PHÂN LỚP 3 LỚP (THREE-TIER ARCHITECTURE)

Mô hình 3 lớp là kiến trúc phần mềm phổ biến giúp tổ chức mã nguồn rõ ràng, dễ bảo trì và mở rộng. Gồm 3 tầng chính:

1. Presentation Layer (Tầng trình bày / Giao diện người dùng)

Hay còn gọi là: UI (User Interface) / Front-End / View

✔️ Chức năng chính:

Hiển thị dữ liệu cho người dùng

Thu thập dữ liệu từ người dùng (qua form, input,...)

Gửi yêu cầu (request) đến tầng xử lý (Business Layer)

Nhận và xử lý sự kiện giao diện (event: click, submit,...)

2. Business Logic Layer (Tầng xử lý nghiệp vụ / Business Layer - BL)

Còn gọi là: Middle Tier / Domain Layer / Logic Layer

✔️ Chức năng chính:

Xử lý logic nghiệp vụ của ứng dụng

Xử lý tính toán, kiểm tra điều kiện, ràng buộc,...

Nhận dữ liệu từ UI và gửi xuống tầng Data Access

Là nơi tách biệt phần xử lý logic khỏi giao diện

3. Data Access Layer (Tầng truy xuất dữ liệu / DAL)

Còn gọi là: Data Layer / Persistence Layer

✔️ Chức năng chính:

Giao tiếp với hệ quản trị cơ sở dữ liệu (MySQL, SQL Server,...)

Thực hiện truy vấn (SQL), lưu trữ, đọc, cập nhật và xóa dữ liệu

Tách biệt việc truy xuất dữ liệu khỏi logic xử lý

💡 Ví dụ đơn giản (thêm khách hàng)

Presentation Layer:

btnSave.addActionListener(e -> customerBL.AddCustomer(customer))*;*

Business Layer:

public class CustomerBL {

    public void AddCustomer(Customer c) {

        customerDAL.insertCustomer(c)*;*

    }

}

Data Access Layer:

public class CustomerDAL {

    public void insertCustomer(Customer c) {

        String sql = "INSERT INTO Customer VALUES (?, ?)"*;*

        // sử dụng JdbcTemplate hoặc Connection để thực thi

    }

}

Dưới đây là toàn bộ lý thuyết mô hình MVC (Model – View – Controller) một cách rõ ràng, dễ hiểu, có in đậm những phần trọng tâm dễ ra trắc nghiệm, kèm ví dụ minh họa:

---

*# 🧠 MÔ HÌNH MVC LÀ GÌ?*

MVC (Model – View – Controller) là một mô hình thiết kế phần mềm chia chương trình thành 3 phần riêng biệt để quản lý code tốt hơn, dễ bảo trì và mở rộng.

---

*# ✅ CÁC THÀNH PHẦN TRONG MVC*

*## 1. Model – Mô hình dữ liệu*

*✔️ Nhiệm vụ:*

\* Quản lý dữ liệu và logic nghiệp vụ

\* Kết nối với cơ sở dữ liệu (CSDL)

\* Thực hiện các thao tác CRUD (Create, Read, Update, Delete)

*🧠 Trắc nghiệm dễ ra:*

\* “Thành phần nào chịu trách nhiệm xử lý dữ liệu?” → Model

--

*## 2. View – Giao diện người dùng*

*✔️ Nhiệm vụ:*

\* Hiển thị dữ liệu cho người dùng

\* Thu thập dữ liệu nhập vào (qua form, button, input,...)

\* Nhận sự kiện từ người dùng (click, nhập liệu,...)

*🧠 Trắc nghiệm dễ ra:*

\* “Lớp View có chức năng gì?” → Hiển thị dữ liệu và nhận sự kiện

\* “Tầng nào nhận sự kiện trong mô hình MVC?” → View

---

*## 3. Controller – Bộ điều khiển*

*✔️ Nhiệm vụ:*

\* Nhận yêu cầu từ View

\* Gọi các phương thức trong Model để xử lý

\* Trả dữ liệu về cho View

*# 🔁 QUY TRÌNH HOẠT ĐỘNG MVC*

1. Người dùng thao tác với View (ví dụ: click “Đăng nhập”)

2. View gửi yêu cầu đến Controller

3. Controller xử lý yêu cầu và gọi Model để thao tác dữ liệu

4. Model trả kết quả về Controller

5. Controller gửi dữ liệu tới View để hiển thị

*# 📌 So sánh MVC và mô hình 3 lớp*

| Đặc điểm             | MVC                       | 3-Layer Architecture      |

| -------------------- | ------------------------- | ------------------------- |

| Mục tiêu chính       | Tổ chức giao diện & logic | Tổ chức toàn bộ kiến trúc |

| Tầng giao diện       | View                      | Presentation Layer        |

| Tầng xử lý nghiệp vụ | Controller                | Business Logic Layer      |

| Tầng dữ liệu         | Model                     | Data Access Layer         |

Dưới đây là toàn bộ kiến thức tổng hợp về ORM (Object-Relational Mapping) – rất hữu ích cho kiểm tra, thực hành và làm đồ án. Mình sẽ in đậm các ý dễ ra trắc nghiệm và cung cấp ví dụ bằng Java (Hibernate) để bạn hiểu rõ hơn.

---

*# 🧠 ORM là gì?*

ORM (Object-Relational Mapping) là một kỹ thuật lập trình giúp chuyển đổi dữ liệu giữa hệ quản trị cơ sở dữ liệu quan hệ (RDBMS) (như MySQL, PostgreSQL...) và các đối tượng trong ngôn ngữ lập trình hướng đối tượng (như Java, C*#...).*

👉 Mục tiêu chính:

Giúp lập trình viên làm việc với database thông qua đối tượng, không cần viết câu lệnh SQL thuần.

---

*# ✅ Ưu điểm của ORM*

\* Giảm code SQL thủ công → dễ viết code hơn

\* Dễ bảo trì, dễ mở rộng

\* Tự động ánh xạ (mapping) giữa bảng và class

\* Hỗ trợ các mối quan hệ: One-to-One, One-to-Many, Many-to-Many

\* Hỗ trợ transaction, lazy loading, caching,...

---

*# ⚠️ Nhược điểm của ORM*

\* Không linh hoạt bằng viết SQL thuần

\* Xử lý kém với truy vấn phức tạp, đặc biệt là JOIN nhiều bảng

\* Hiệu suất đôi khi kém hơn dùng JDBC trực tiếp

---

*# 🔰 Các thư viện ORM phổ biến*

| Ngôn ngữ | ORM                                      |

| -------- | ---------------------------------------- |

| Java     | Hibernate, JPA (Jakarta Persistence) |

| C*#       | Entity Framework                         |*

| Python   | SQLAlchemy, Django ORM                   |

| PHP      | Doctrine ORM, Eloquent (Laravel)         |

---

*# 📦 Cách hoạt động của ORM*

*Ví dụ:*

Giả sử có bảng `users` trong MySQL:

```sql

CREATE TABLE users (

  id INT PRIMARY KEY AUTO\_INCREMENT,

  name VARCHAR(50),

  email VARCHAR(100)

)*;*

```

ORM sẽ ánh xạ thành class Java như sau:

```java

@Entity

@Table(name = "users")

public class User {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private int id*;*

    private String name*;*

    private String email*;*

    // getters, setters

}

---

*# 📌 Các annotation quan trọng trong ORM (Hibernate / JPA)*

| Annotation                                             | Ý nghĩa                                      |

| ------------------------------------------------------ | -------------------------------------------- |

| `@Entity`                                              | Đánh dấu class là 1 thực thể ánh xạ với bảng |

| `@Table(name = "...")`                                 | Chỉ định tên bảng tương ứng                  |

| `@Id`                                                  | Chỉ định khóa chính                          |

| `@GeneratedValue`                                      | Tự động tăng giá trị id                      |

| `@Column(name = "...")`                                | Ánh xạ tên cột nếu khác tên biến             |

| `@OneToMany`, `@ManyToOne`, `@ManyToMany`, `@OneToOne` | Ánh xạ các mối quan hệ giữa các entity       |

---

*# 🔄 Các loại quan hệ trong ORM*

*## 1. One-to-One*

Ví dụ: Mỗi `User` có một `Profile`.

```java

@OneToOne

@JoinColumn(name = "profile\_id")

private Profile profile*;*

```

*## 2. One-to-Many*

Ví dụ: Một `User` có nhiều `Post`.

```java

@OneToMany(mappedBy = "user")

private List<Post> posts*;*

```

*## 3. Many-to-One*

Ví dụ: Nhiều `Post` thuộc về 1 `User`.

```java

@ManyToOne

@JoinColumn(name = "user\_id")

private User user*;*

```

*## 4. Many-to-Many*

Ví dụ: Nhiều `Student` học nhiều `Course`.

```java

@ManyToMany

@JoinTable(

  name = "student\_course",

  joinColumns = @JoinColumn(name = "student\_id"),

  inverseJoinColumns = @JoinColumn(name = "course\_id")

)

private Set<Course> courses*;*

```

---

*# 🔧 ORM vs JDBC (truy vấn thuần)*

| Tiêu chí       | ORM                              | JDBC truyền thống         |

| -------------- | -------------------------------- | ------------------------- |

| Tính dễ dùng   | Cao (code ít, dễ hiểu)       | Phải viết SQL thủ công    |

| Mapping        | Tự động ánh xạ với class         | Phải ánh xạ thủ công      |

| Tính mở rộng   | Cao                              | Trung bình                |

| Hiệu năng      | Có thể kém nếu truy vấn phức tạp | Nhanh nếu viết SQL tối ưu |

| Tính kiểm soát | Kém hơn JDBC                     | Kiểm soát toàn bộ         |

---

*# 📝 Câu hỏi trắc nghiệm mẫu*

1. ORM là gì?

   → Kỹ thuật ánh xạ giữa đối tượng và bảng dữ liệu

2. Annotation nào dùng để đánh dấu khóa chính?

   → `@Id`

3. Mối quan hệ giữa lớp `Student` và `Course` (nhiều - nhiều) dùng annotation nào?

   → @ManyToMany

4. ORM hỗ trợ điều gì sau đây?

   \* a. Lazy Loading

   \* b. Mapping bảng sang class

   \* c. Transaction

   \* d. Tất cả các đáp án trên

     → Đáp án đúng: d

5. ORM phù hợp nhất khi nào?

   → Dự án vừa hoặc lớn, muốn dễ bảo trì và ít viết SQL

Dưới đây là một ví dụ đầy đủ CRUD bằng Hibernate với Spring Boot, sử dụng `User` entity (bao gồm các thao tác Create, Read, Update, Delete).

```

└── src/main/java/com/example/demo/

    ├── entity/User.java

    ├── repository/UserRepository.java

    ├── service/UserService.java

    ├── controller/UserController.java

    └── DemoApplication.java

```

---

*## 1. `User` Entity (Dùng JPA - Hibernate)*

package com.example.demo.entity*;*

import jakarta.persistence.\**;*

@Entity

@Table(name = "users")

public class User {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id*;*

    private String name*;*

    private String email*;*

    // Constructors

    public User() {}

    public User(String name, String email) {

        this.name = name*;*

        this.email = email*;*

    }

    // Getters & Setters

    public Long getId() { return id*; }*

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

    public String getName() { return name*; }*

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

    public String getEmail() { return email*; }*

    public void setEmail(String email) { this.email = email*; }*

}

*## 2. `UserRepository` (Dùng `JpaRepository`)*

package com.example.demo.repository*;*

import com.example.demo.entity.User*;*

import org.springframework.data.jpa.repository.JpaRepository*;*

public interface UserRepository extends JpaRepository<User, Long> {

}

*## 3. `UserService` (Chứa logic nghiệp vụ)*

@Service

public class UserService {

    @Autowired

    private UserRepository userRepository*;*

    public User createUser(User user) {

        return userRepository.save(user)*;*

    }

    public List<User> getAllUsers() {

        return userRepository.findAll()*;*

    }

    public Optional<User> getUserById(Long id) {

        return userRepository.findById(id)*;*

    }

    public User updateUser(Long id, User userDetails) {

        User user = userRepository.findById(id).orElseThrow()*;*

        user.setName(userDetails.getName())*;*

        user.setEmail(userDetails.getEmail())*;*

        return userRepository.save(user)*;*

    }

    public void deleteUser(Long id) {

        userRepository.deleteById(id)*;*

    }

}

@RestController

@RequestMapping("/api/users")

public class UserController {

    @Autowired

    private UserService userService*;*

    @PostMapping

    public User createUser(@RequestBody User user) {

        return userService.createUser(user)*;*

    }

    @GetMapping

    public List<User> getAllUsers() {

        return userService.getAllUsers()*;*

    }

    @GetMapping("/{id}")

    public User getUserById(@PathVariable Long id) {

        return userService.getUserById(id).orElse(null)*;*

    }

    @PutMapping("/{id}")

    public User updateUser(@PathVariable Long id, @RequestBody User userDetails) {

        return userService.updateUser(id, userDetails)*;*

    }

    @DeleteMapping("/{id}")

    public void deleteUser(@PathVariable Long id) {

        userService.deleteUser(id)*;*

    }

}

Dưới đây là toàn bộ kiến thức cơ bản cần nhớ về Hibernate – một phần rất quan trọng trong lập trình Java và ORM, thường được hỏi trong kiểm tra và thực hành.

*## 🔰 1. Hibernate là*

Hibernate là một framework ORM mã nguồn mở cho Java, dùng để ánh xạ các đối tượng Java (class) với bảng trong cơ sở dữ liệu quan hệ (RDBMS).

👉 Nó giúp quản lý cơ sở dữ liệu bằng lập trình hướng đối tượng, thay vì phải viết SQL thủ công.

---

*## ✅ 2. Ưu điểm của Hibernate*

\* Tự động ánh xạ (mapping) giữa class và table

\* Không cần viết nhiều SQL → dễ bảo trì, gọn gàng

\* Hỗ trợ transaction, lazy loading, caching,…

\* Độc lập CSDL (MySQL, PostgreSQL, Oracle, H2,…)

\* Quản lý mối quan hệ One-to-One, One-to-Many, Many-to-Many

---

*## ⚙️ 3. Cách hoạt động của Hibernate*

1. Hibernate Configuration: Đọc file `hibernate.cfg.xml` hoặc cấu hình Spring Boot để kết nối database.

2. SessionFactory: Tạo ra các `Session`.

3. Session: Dùng để thao tác với DB (thêm, sửa, xóa, truy vấn).

4. Transaction: Đảm bảo an toàn dữ liệu (ACID).

---

*## 📌 4. Các thành phần chính của Hibernate*

| Thành phần         | Chức năng                                               |

| ------------------ | ------------------------------------------------------- |

| Configuration  | Cấu hình Hibernate (database, dialect, driver, entity…) |

| SessionFactory | Factory tạo `Session`                                   |

| Session        | Làm việc với database                                   |

| Transaction    | Quản lý giao dịch                                       |

| Query / HQL    | Truy vấn dữ liệu                                        |

---

*## 🧾 5. Annotation quan trọng trong Hibernate*

| Annotation          | Ý nghĩa                                |

| ------------------- | -------------------------------------- |

| `@Entity`           | Đánh dấu class là thực thể ánh xạ bảng |

| `@Table(name = "")` | Tên bảng                               |

| `@Id`               | Khóa chính                             |

| `@GeneratedValue`   | Tự động tăng                           |

| `@Column`           | Ánh xạ cột                             |

| `@OneToMany`        | Một-nhiều                              |

| `@ManyToOne`        | Nhiều-một                              |

| `@ManyToMany`       | Nhiều-nhiều                            |

| `@JoinColumn`       | Cột khóa ngoại                         |

---

*## 🧪 6. Hibernate Query Language (HQL)*

Giống SQL nhưng dùng tên class và field thay vì tên bảng.

Ví dụ:

Session session = sessionFactory.openSession()*;*

Query query = session.createQuery("FROM User WHERE name = :name")*;*

query.setParameter("name", "Nam")*;*

List<User> users = query.list()*;*

```

--

*## 🔄 7. Các hàm CRUD phổ biến*

| Phương thức Hibernate | Mô tả       |

| --------------------- | ----------- |

| `save()`              | Thêm mới    |

| `get()` / `find()`    | Tìm theo ID |

| `update()`            | Cập nhật    |

| `delete()`            | Xóa bản ghi |

---

*## 📝 8. File cấu hình `hibernate.cfg.xml` (nếu không dùng Spring Boot)*

```xml

<hibernate-configuration>

  <session-factory>

    <property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property>

    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/your\_db</property>

    <property name="hibernate.connection.username">root</property>

    <property name="hibernate.connection.password">password</property>

    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

    <property name="show\_sql">true</property>

    <property name="hbm2ddl.auto">update</property>

    <mapping class="com.example.User"/>

  </session-factory>

</hibernate-configuration>

```

*## 💡 9. Một số cấu hình quan trọng trong Hibernate*

| Thuộc tính               | Ý nghĩa                                                          |

| ------------------------ | ---------------------------------------------------------------- |

| `hibernate.dialect`      | Biết cách viết SQL phù hợp với loại CSDL                         |

| `hibernate.hbm2ddl.auto` | `update`, `create`, `validate`, `none` – cách Hibernate tạo bảng |

| `show\_sql`               | In SQL ra console                                                |

Dưới đây là toàn bộ kiến thức và ví dụ chi tiết về One-to-Many và Many-to-One trong Hibernate (dùng với Spring Boot). Đây là phần cực kỳ quan trọng, thường xuất hiện trong thi lý thuyết lẫn khi làm đồ án thực tế.

---

*## 🔁 1. One-to-Many và Many-to-One là gì?*

*✅ One-to-Many*

\* Một đối tượng cha có thể liên kết với nhiều đối tượng con.

\* Ví dụ: Một User có nhiều Post.

*✅ Many-to-One*

\* Nhiều đối tượng con liên kết đến một đối tượng cha.

\* Ví dụ: Nhiều Post thuộc về một User.

⚠️ Hai mối quan hệ này luôn đi kèm: Một bên là `@OneToMany`, bên kia là `@ManyToOne`.

---

*## 📦 2. Cấu trúc bảng trong ví dụ*

Giả sử bạn có 2 bảng:

\* `users` (User)

\* `posts` (Post, có khóa ngoại `user\_id` trỏ đến User)

---

*## 🧱 3. Entity: User và Post*

*🔹 `User.java` (One-to-Many side)*

```java

@Entity

@Table(name = "users")

public class User {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id*;*

    private String name*;*

    // Một User có nhiều Post

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL, orphanRemoval = true)

    private List<Post> posts = new ArrayList<>()*;*

    // getters, setters

}

```

*🔹 `Post.java` (Many-to-One side)*

```java

@Entity

@Table(name = "posts")

public class Post {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id*;*

    private String content*;*

    // Nhiều Post thuộc về một User

    @ManyToOne(fetch = FetchType.LAZY)

    @JoinColumn(name = "user\_id")

    private User user*;*

    // getters, setters

}

```

---

*## 🔁 4. Cách sử dụng (thêm dữ liệu)*

```java

User user = new User()*;*

user.setName("Nguyễn Văn A")*;*

Post post1 = new Post()*;*

post1.setContent("Bài viết 1")*;*

post1.setUser(user)*; // thiết lập chủ nhân bài viết*

Post post2 = new Post()*;*

post2.setContent("Bài viết 2")*;*

post2.setUser(user)*;*

user.getPosts().add(post1)*;*

user.getPosts().add(post2)*;*

userRepository.save(user)*; // Tự động lưu cả 2 bài viết nhờ cascade*

```

---

*## 💡 5. Chú thích các Annotation chính*

| Annotation                      | Ý nghĩa                                                  |

| ------------------------------- | -------------------------------------------------------- |

| `@OneToMany(mappedBy = "user")` | Quan hệ một-nhiều. `mappedBy` phải là tên field bên Post |

| `cascade = CascadeType.ALL`     | Tự động lưu cả quan hệ con                               |

| `orphanRemoval = true`          | Xóa con khi bị remove khỏi danh sách                     |

| `@ManyToOne`                    | Quan hệ nhiều-một                                        |

| `@JoinColumn(name = "user\_id")` | Khóa ngoại                                               |

*## 🧠 6. Câu hỏi trắc nghiệm thường gặp*

1. Annotation nào dùng để biểu diễn mối quan hệ One-to-Many trong Hibernate?

   → `@OneToMany`

2. Trong `@OneToMany`, thuộc tính `mappedBy` dùng để làm gì?

   → Xác định tên biến bên phía chủ sở hữu của quan hệ (phía Many).

3. CascadeType.ALL dùng để làm gì?

   → Tự động truyền thao tác save/update/delete đến các entity liên quan.

4. Trong quan hệ nhiều-một, cột khóa ngoại được ánh xạ bằng annotation nào?

   → `@JoinColumn`

5. Nếu muốn tự động xóa các phần tử con khi bị remove khỏi danh sách, dùng?

   → `orphanRemoval = true`

*## 🧪 7. Lưu ý khi triển khai*

\* Đặt `@OneToMany(mappedBy = "user")` ở phía không giữ khóa ngoại.

\* Phía giữ khóa ngoại (ở đây là `Post`) sẽ có `@ManyToOne` và `@JoinColumn`.

\* Nên dùng `CascadeType.ALL` nếu muốn thao tác đồng thời.

\* Tránh lặp vô hạn khi `toString()` do gọi qua lại giữa hai class → dùng `@JsonIgnore` ở phía `@ManyToOne`.

🔗 1. One-to-One (Quan hệ độc lập)

✅ Khái niệm:

One-to-One là quan hệ giữa 2 bảng mà mỗi bản ghi của bảng A chỉ liên kết với 1 bản ghi của bảng B và ngược lại.

Ví dụ: Một User chỉ có một Profile riêng biệt và mỗi Profile chỉ thuộc về một User.

📘 Ví dụ: User – Profile

🔹 User.java

@Entity

@Table(name = "users")

public class User {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id*;*

    private String username*;*

    @OneToOne(cascade = CascadeType.ALL)

    @JoinColumn(name = "profile\_id", referencedColumnName = "id")

    private Profile profile*;*

    // getters, setters

}

🔹 Profile.java

@Entity

@Table(name = "profiles")

public class Profile {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id*;*

    private String fullName*;*

    private String address*;*

    @OneToOne(mappedBy = "profile")

    private User user*;*

    // getters, setters

}

📌 Giải thích:

@OneToOne: xác định quan hệ một-một

@JoinColumn: bên giữ khóa ngoại (ở đây là User)

mappedBy = "profile": chỉ định quan hệ ở phía ngược lại

💡 Các chiến lược quan hệ trong One-to-One

Chiến lược  Mô tả

@JoinColumn Bảng A giữ khóa ngoại trỏ đến bảng B

@PrimaryKeyJoinColumn   Hai bảng dùng chung cùng một khóa chính (hiếm dùng)

mappedBy    Dùng ở phía không sở hữu quan hệ

📊 2. FetchType.LAZY vs EAGER

✅ Khái niệm:

Hibernate hỗ trợ 2 chế độ load dữ liệu liên quan (fetching):

FetchType   Mô tả

LAZY    Chỉ truy vấn dữ liệu khi cần dùng (trì hoãn)

EAGER   Tự động load ngay lập tức cùng đối tượng cha

🔍 Ví dụ:

@OneToMany(mappedBy = "user", fetch = FetchType.LAZY)

private List<Post> posts*;*

→ Khi gọi userRepository.findById(id), dữ liệu trong posts chưa được load ngay, chỉ khi gọi user.getPosts() thì Hibernate mới truy vấn.

Ngược lại:

@OneToMany(mappedBy = "user", fetch = FetchType.EAGER)

→ Dữ liệu posts sẽ load ngay lập tức khi User được truy vấn.

⚠️ Lưu ý khi dùng

Trường hợp  Nên dùng

Dữ liệu liên quan ít và cần dùng ngay   EAGER

Dữ liệu lớn, quan hệ phức tạp (OneToMany)   LAZY (tốt hơn cho hiệu suất)

💥 Lỗi phổ biến:

Khi dùng LAZY, nếu bạn đóng Session quá sớm, gọi getXXX() sẽ gây lỗi:

org.hibernate.LazyInitializationException

→ Cần xử lý qua:

@Transactional

Dùng fetch join trong JPQL/HQL

Dưới đây là toàn bộ kiến thức cơ bản và quan trọng nhất về Spring Boot – phần thường ra thi lý thuyết, trắc nghiệm hoặc cần hiểu kỹ khi làm dự án thực tế:

---

*## 🚀 1. Spring Boot là gì?*

> Spring Boot là một framework dựa trên Spring, giúp lập trình viên xây dựng ứng dụng web và microservices một cách nhanh chóng, dễ dàng và ít cấu hình.

---

*## 🎯 2. Mục tiêu của Spring Boot*

\* ✅ Tự động cấu hình (Auto-configuration)

\* ✅ Giảm cấu hình XML rườm rà

\* ✅ Tích hợp dễ dàng với cơ sở dữ liệu, security, RESTful API, v.v.

\* ✅ Chạy ứng dụng dễ dàng chỉ với `main()` (ứng dụng độc lập - standalone)

---

*## 🧩 3. Cấu trúc 3 lớp cơ bản trong Spring Boot*

| Tầng           | Vai trò                              |

| -------------- | ------------------------------------ |

| Controller | Giao tiếp với client (giao diện/API) |

| Service    | Xử lý logic nghiệp vụ                |

| Repository | Truy cập dữ liệu (kết nối DB)        |

---

*## 📦 4. Các annotation thường gặp*

| Annotation                          | Vai trò                          |

| ----------------------------------- | -------------------------------- |

| `@SpringBootApplication`            | Đánh dấu lớp chính chạy ứng dụng |

| `@RestController`                   | Tạo controller RESTful           |

| `@Service`                          | Đánh dấu tầng nghiệp vụ          |

| `@Repository`                       | Kết nối cơ sở dữ liệu            |

| `@Autowired`                        | Tự động tiêm phụ thuộc           |

| `@Entity`                           | Ánh xạ lớp Java thành bảng CSDL  |

| `@Id`                               | Định danh khóa chính             |

| `@GeneratedValue`                   | Tự động tăng ID                  |

| `@GetMapping`, `@PostMapping`, etc. | Mapping các HTTP method          |

---

*## 🧪 5. Cấu trúc thư mục cơ bản*

```

src/

 └── main/

      └── java/

           └── com.example.demo/

                 ├── controller/

                 ├── service/

                 ├── repository/

                 ├── model/

                 └── DemoApplication.java

```

---

*## 🔁 6. CRUD cơ bản với Spring Boot*

| Phương thức | Annotation       | Chức năng   |

| ----------- | ---------------- | ----------- |

| `GET`       | `@GetMapping`    | Lấy dữ liệu |

| `POST`      | `@PostMapping`   | Tạo mới     |

| `PUT`       | `@PutMapping`    | Cập nhật    |

| `DELETE`    | `@DeleteMapping` | Xóa dữ liệu |

---

*## 🧠 7. Spring Boot DevTools*

\* Tự động reload lại ứng dụng khi có thay đổi code.

\* Giúp lập trình nhanh hơn mà không cần stop/start mỗi lần.

---

*## 🐘 8. Spring Boot với Cơ sở dữ liệu*

\* Kết nối thông qua `application.properties` hoặc `application.yml`

```properties

spring.datasource.url=jdbc:mysql://localhost:3306/ten\_db

spring.datasource.username=root

spring.datasource.password=123456

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

```

*## 📘 9. Spring Boot Starter Dependencies*

Spring Boot cung cấp nhiều starter giúp thêm chức năng dễ dàng:

| Starter                        | Chức năng              |

| ------------------------------ | ---------------------- |

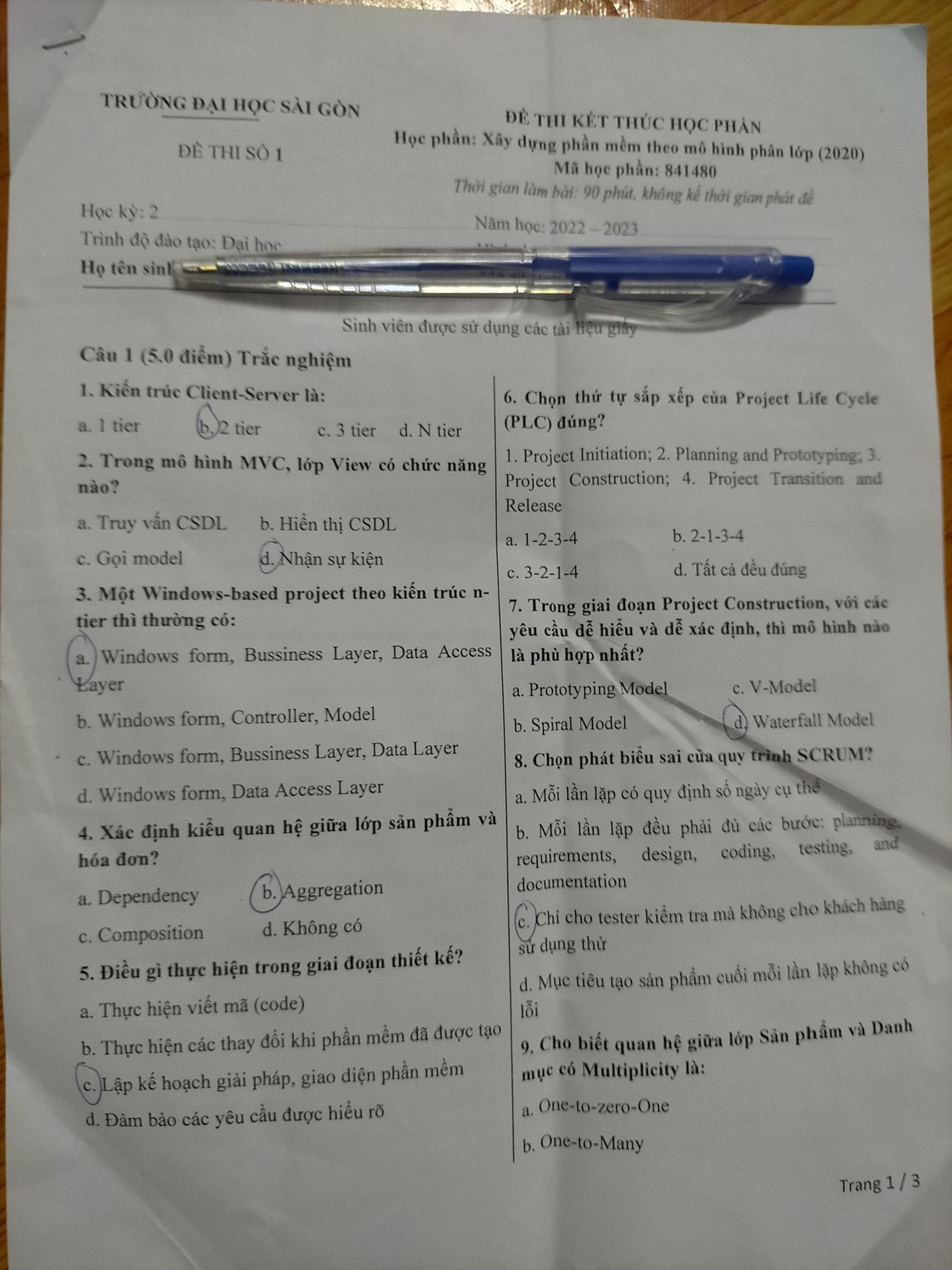
| `spring-boot-starter-web`      | Web, REST API          |

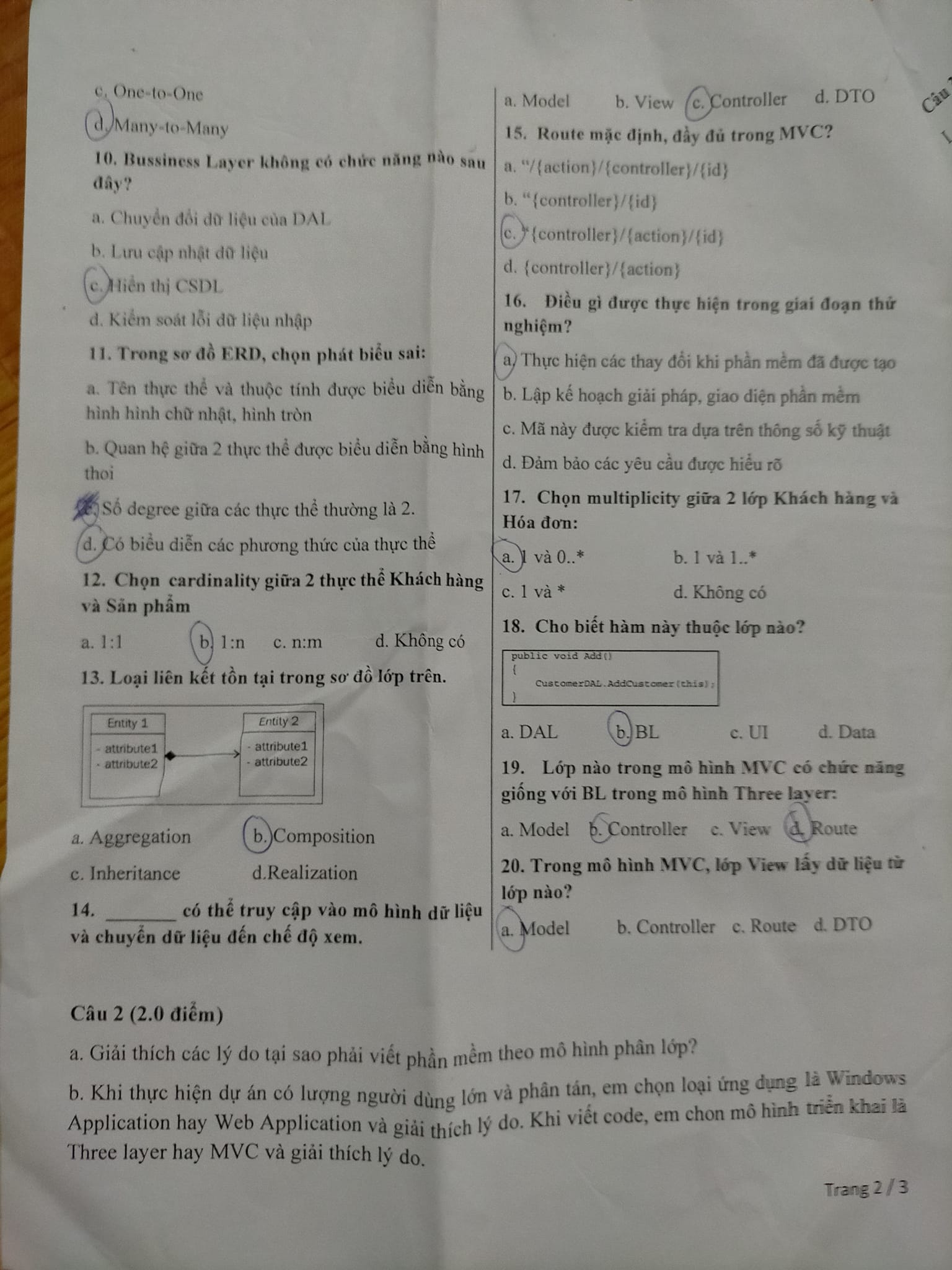
| `spring-boot-starter-data-jpa` | Kết nối và thao tác DB |

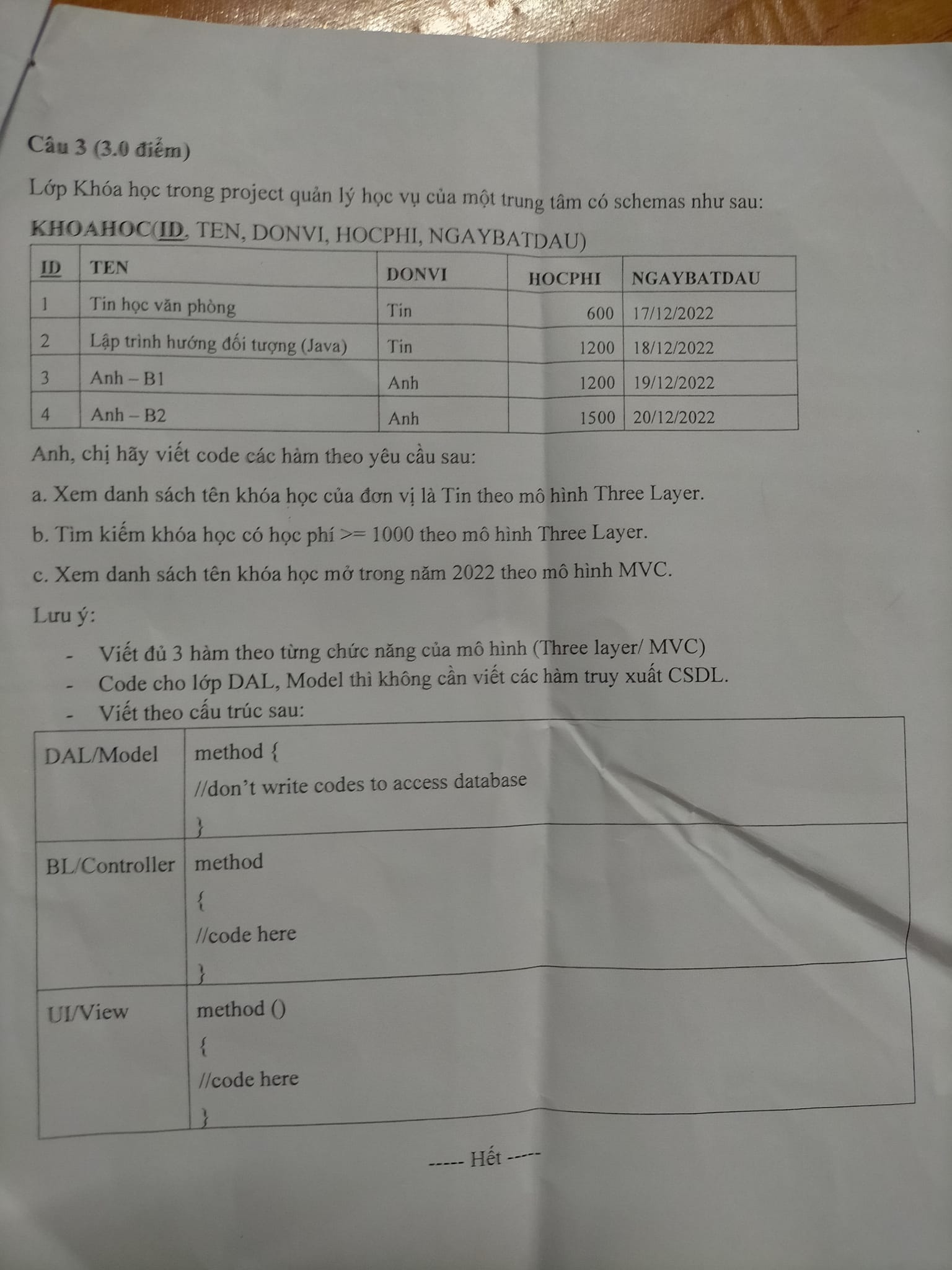
| `spring-boot-starter-security` | Bảo mật                |

| `spring-boot-starter-test`     | Unit test              |

Đề 22-23







Câu hỏi tào lao

**Kiến trúc Client-Server là:**

- Đáp án của bạn chọn b (2 tier) là đúng. Phân tích như sau:

Phân tích đáp án:

- a. 1 tier: Kiến trúc 1 tầng (1-tier) là mô hình tất cả các thành phần (giao diện, logic kinh doanh, và dữ liệu) được tích hợp trên một máy duy nhất, thường thấy trong các ứng dụng đơn giản hoặc cục bộ (ví dụ: phần mềm chạy trên một máy tính cá nhân mà không cần kết nối mạng). Đây không phải là kiến trúc Client-Server.

- b. 2 tier ✓: Kiến trúc Client-Server cơ bản gồm 2 tầng:

- Client: Chịu trách nhiệm giao diện người dùng và gửi yêu cầu.

- Server: Xử lý logic kinh doanh và quản lý dữ liệu. Đây là mô hình phổ biến cho các hệ thống sớm như ứng dụng desktop kết nối trực tiếp đến cơ sở dữ liệu.

-c. 3 tier: Kiến trúc 3 tầng tách biệt thành 3 lớp: Presentation (giao diện), Application (logic kinh doanh), và Data (dữ liệu). Đây là một biến thể nâng cao của Client-Server, thường dùng trong các hệ thống web hiện đại.

- N tier: Kiến trúc N tầng là sự mở rộng của 3 tier, với nhiều lớp trung gian (ví dụ: lớp bảo mật, caching), thường thấy trong các hệ thống phức tạp như hệ thống doanh nghiệp lớn.

Khi nào nên chọn các đáp án khác?

- Chọn a (1 tier) nếu câu hỏi đề cập đến một ứng dụng độc lập, không có sự phân chia giữa client và server (ví dụ: phần mềm chạy offline trên một máy).

- Chọn c (3 tier) nếu câu hỏi yêu cầu mô tả một hệ thống phân tầng rõ ràng với lớp logic kinh doanh tách biệt (ví dụ: ứng dụng web với server trung gian).

- Chọn d (N tier) nếu câu hỏi liên quan đến các hệ thống phức tạp với nhiều lớp phân tán (ví dụ: hệ thống cloud hoặc microservices).

Một số câu hỏi tương tự và đáp án:

1. Kiến trúc nào thường được sử dụng cho các ứng dụng web hiện đại?

- a. 1 tier

- b. 2 tier

- c. 3 tier ✓

- d. N tier

- Đáp án: c (3 tier) - Vì ứng dụng web thường tách biệt giao diện (client), logic (server ứng dụng), và dữ liệu (cơ sở dữ liệu).

2. Hệ thống nào dưới đây là ví dụ điển hình của kiến trúc 1 tier?

- a. Ứng dụng Microsoft Word ✓

- b. Hệ thống ngân hàng trực tuyến

- c. Website thương mại điện tử

- d. Hệ thống email server

- Đáp án: a (Ứng dụng Microsoft Word) - Vì nó chạy cục bộ trên một máy mà không cần server.

3. Kiến trúc N tier phù hợp nhất với loại hệ thống nào?

- a. Ứng dụng đơn giản trên điện thoại

- b. Hệ thống quản lý doanh nghiệp lớn ✓

- c. Phần mềm văn phòng

- d. Trò chơi offline

- Đáp án: b (Hệ thống quản lý doanh nghiệp lớn) - Vì N tier hỗ trợ nhiều lớp phân tán, phù hợp với các hệ thống phức tạp.

**Đáp án bạn chọn a (Windows form, Business Layer, Data Access Layer) là đúng. Dưới đây là phân tích chi tiết:**

Phân tích đáp án:

- a. Windows form, Business Layer, Data Access Layer ✓:

- Đây là cấu trúc chuẩn của một ứng dụng Windows-based theo kiến trúc N-tier:

- Windows Form: Lớp giao diện (Presentation Layer), nơi người dùng tương tác (UI).

- Business Layer: Lớp xử lý logic kinh doanh, chịu trách nhiệm cho các quy tắc và xử lý nghiệp vụ.

- Data Access Layer (DAL): Lớp truy cập dữ liệu, giao tiếp với cơ sở dữ liệu, đảm bảo tách biệt giữa logic kinh doanh và truy cập dữ liệu.

- Đây là mô hình phân tầng phổ biến trong các ứng dụng Windows, đảm bảo tính mô-đun và dễ bảo trì.

- b. Windows form, Controller, Model:

- Cấu trúc này phù hợp hơn với mô hình MVC (Model-View-Controller), thường thấy trong các ứng dụng web hoặc một số ứng dụng không phải Windows-based. Controller không phải là thành phần điển hình trong kiến trúc N-tier của ứng dụng Windows.

- c. Windows form, Business Layer, Data Layer:

- Mặc dù có vẻ giống đáp án đúng, nhưng thuật ngữ "Data Layer" không chính xác trong ngữ cảnh N-tier. Thuật ngữ chuẩn là Data Access Layer (DAL), vì lớp này không chỉ lưu trữ dữ liệu mà còn xử lý việc truy cập và tương tác với cơ sở dữ liệu. "Data Layer" thường ám chỉ chính cơ sở dữ liệu (database), không phải lớp trung gian.

- d. Windows form, Data Access Layer:

- Cấu trúc này thiếu Business Layer, không đáp ứng được yêu cầu của kiến trúc N-tier (ít nhất 3 tầng: Presentation, Business, Data Access). Thiếu Business Layer sẽ làm logic kinh doanh bị trộn lẫn vào giao diện hoặc DAL, gây khó khăn trong bảo trì.

Khi nào nên chọn các đáp án khác?

- Chọn b nếu câu hỏi liên quan đến kiến trúc MVC thay vì N-tier, ví dụ: ứng dụng web hoặc mobile.

- Chọn c nếu ngữ cảnh chấp nhận "Data Layer" thay cho "Data Access Layer", nhưng điều này hiếm xảy ra vì thuật ngữ không chuẩn.

- Chọn d nếu câu hỏi đề cập đến kiến trúc 2-tier đơn giản, không yêu cầu Business Layer.

Kết luận:

Đáp án a là chính xác và phù hợp nhất với kiến trúc N-tier của một dự án Windows-based, vì nó thể hiện đầy đủ các tầng cần thiết: giao diện (Windows Form), logic kinh doanh (Business Layer), và truy cập dữ liệu (Data Access Layer).

**4: Xác định tầng quan hệ giữa lớp sản phẩm và hóa đơn?**

Đáp án bạn chọn b (Aggregation) là đúng. Dưới đây là phân tích chi tiết:

Phân tích đáp án:

- a. Dependency (Phụ thuộc):

- Mối quan hệ Dependency xảy ra khi một lớp sử dụng một lớp khác nhưng không giữ tham chiếu lâu dài (thường chỉ trong một phương thức). Ví dụ, nếu lớp hóa đơn chỉ tạm thời sử dụng lớp sản phẩm để tính giá, nhưng không lưu trữ sản phẩm, thì đó là Dependency. Trong trường hợp này, hóa đơn thường lưu danh sách sản phẩm, nên Dependency không phù hợp.

- b. Aggregation ✓ (Tập hợp):

- Aggregation là một dạng của mối quan hệ "has-a" (có một), nhưng lỏng lẻo hơn Composition. Trong trường hợp này:

- Một hóa đơn (Invoice) chứa nhiều sản phẩm (Product), tức là hóa đơn "có" các sản phẩm.

- Tuy nhiên, sản phẩm có thể tồn tại độc lập, không phụ thuộc vào hóa đơn (nếu hóa đơn bị xóa, sản phẩm vẫn tồn tại trong hệ thống).

- Đây là đặc điểm của Aggregation, vì mối quan hệ giữa hóa đơn và sản phẩm là tập hợp nhưng không ràng buộc về vòng đời.

- c. Composition (Hợp thành):

- Composition cũng là một dạng của "has-a", nhưng chặt chẽ hơn Aggregation. Trong Composition, nếu đối tượng cha (Invoice) bị hủy, thì các đối tượng con (Product) cũng phải bị hủy.

- Ví dụ: Nếu một hóa đơn bị xóa, sản phẩm cũng bị xóa theo, thì đó là Composition. Nhưng trong thực tế, sản phẩm thường không bị xóa khi hóa đơn bị xóa, vì sản phẩm có thể được sử dụng trong nhiều hóa đơn khác. Do đó, Composition không phù hợp.

- d. Không có (No relation):

- Rõ ràng giữa hóa đơn và sản phẩm có mối quan hệ, vì hóa đơn cần tham chiếu đến sản phẩm để liệt kê chi tiết. Do đó, đáp án này sai.

Khi nào nên chọn các đáp án khác?

- Chọn a (Dependency) nếu hóa đơn chỉ tạm thời sử dụng thông tin sản phẩm mà không lưu trữ (ví dụ: chỉ lấy giá sản phẩm để tính tổng mà không giữ danh sách sản phẩm).

- Chọn c (Composition) nếu sản phẩm được tạo ra chỉ để phục vụ cho một hóa đơn cụ thể và không thể tồn tại độc lập (ví dụ: các mục trong hóa đơn được tạo động và bị xóa khi hóa đơn xóa).

- Chọn d (Không có) nếu không có bất kỳ mối quan hệ nào giữa hai lớp, ví dụ: hóa đơn và một lớp không liên quan như "Khách hàng VIP".

Kết luận:

Đáp án b (Aggregation) là chính xác vì mối quan hệ giữa hóa đơn và sản phẩm là "has-a" lỏng lẻo: hóa đơn chứa danh sách sản phẩm, nhưng sản phẩm có thể tồn tại độc lập ngoài hóa đơn. Đây là đặc điểm điển hình của Aggregation trong thiết kế hướng đối tượng.

**5: Điều gì thực hiện trong giai đoạn thiết kế?**

Đáp án bạn chọn c (Lập kế hoạch giải pháp, giao diện phần mềm) là đúng. Dưới đây là phân tích chi tiết:

Phân tích đáp án:

- a. Thực hiện viết mã (code):

- Viết mã là hoạt động thuộc giai đoạn phát triển (development), không phải giai đoạn thiết kế. Giai đoạn thiết kế diễn ra trước khi viết mã, nên đáp án này sai.

- b. Thực hiện các thay đổi khi phần mềm đã được tạo:

- Việc thực hiện thay đổi là hoạt động của giai đoạn bảo trì (maintenance) hoặc kiểm thử (testing), không phải giai đoạn thiết kế. Giai đoạn thiết kế tập trung vào lập kế hoạch ban đầu, không phải sửa chữa.

- c. Lập kế hoạch giải pháp, giao diện phần mềm ✓:

- Giai đoạn thiết kế trong vòng đời phát triển phần mềm (SDLC) tập trung vào việc:

- Lập kế hoạch giải pháp kỹ thuật để đáp ứng yêu cầu.

- Thiết kế giao diện người dùng (UI) và kiến trúc hệ thống (architecture).

- Đây là giai đoạn xác định cách phần mềm sẽ được xây dựng, bao gồm sơ đồ, biểu đồ luồng dữ liệu, và bố cục giao diện, nên đáp án này đúng.

- d. Đảm bảo các yêu cầu được hiểu rõ:

- Việc đảm bảo yêu cầu được hiểu rõ thuộc về giai đoạn phân tích yêu cầu (requirements analysis), diễn ra trước giai đoạn thiết kế. Giai đoạn thiết kế sử dụng các yêu cầu đã được xác định để lập kế hoạch, không phải để hiểu chúng, nên đáp án này sai.

Khi nào nên chọn các đáp án khác?

- Chọn a nếu câu hỏi đề cập đến giai đoạn phát triển hoặc lập trình.

- Chọn b nếu câu hỏi liên quan đến giai đoạn bảo trì hoặc sửa lỗi sau khi phần mềm đã được triển khai.

- Chọn d nếu câu hỏi hỏi về giai đoạn phân tích yêu cầu.

Kết luận:

Đáp án c là chính xác vì giai đoạn thiết kế tập trung vào việc lập kế hoạch giải pháp và thiết kế giao diện phần mềm, dựa trên các yêu cầu đã được xác định trước đó.

**6: Chọn thứ tự sắp xếp của Project Life Cycle (PLC) đúng?**

Đáp án chọn: a (1-2-3-4)

Thứ tự đúng: 1. Project Initiation; 2. Planning and Prototyping; 3. Project Construction; 4. Project Transition and Release

Đáp án bạn chọn a (1-2-3-4) là đúng. Dưới đây là phân tích chi tiết:

Phân tích đáp án:

Thứ tự của Project Life Cycle (PLC) mà bạn đưa ra là:

1. Project Initiation (Khởi tạo dự án)

2. Planning and Prototyping (Lập kế hoạch và tạo mẫu thử)

3. Project Construction (Xây dựng dự án)

4. Project Transition and Release (Chuyển giao và phát hành dự án)

- a. 1-2-3-4 ✓:

- Đây là thứ tự đúng của vòng đời dự án (PLC):

- Project Initiation: Bắt đầu dự án, xác định mục tiêu, phạm vi, và các bên liên quan.

- Planning and Prototyping: Lập kế hoạch chi tiết (nguồn lực, thời gian, ngân sách) và tạo mẫu thử để kiểm tra ý tưởng.

- Project Construction: Thực hiện xây dựng sản phẩm/dịch vụ, bao gồm phát triển và kiểm thử.

- Project Transition and Release: Chuyển giao sản phẩm cho khách hàng hoặc người dùng và phát hành chính thức.

- Thứ tự này phản ánh quy trình logic từ khởi tạo đến hoàn thành dự án, nên đáp án này đúng.

**7: Trong giai đoạn Project Construction, với các yêu cầu để hiển thị và đê cập nhật, thì mô hình nào là phù hợp nhất?**

Đáp án bạn chọn d (Waterfall Model) là hợp lý trong ngữ cảnh câu hỏi, nhưng cần phân tích kỹ để xác nhận. Dưới đây là phân tích chi tiết:

Phân tích đáp án:

- a. Prototyping Model:

- Mô hình này tập trung vào việc tạo các nguyên mẫu (prototype) để kiểm tra và lấy phản hồi từ người dùng, thường phù hợp khi yêu cầu không rõ ràng hoặc cần thử nghiệm giao diện sớm. Tuy nhiên, nó không tối ưu cho việc hiển thị và cập nhật khi yêu cầu đã được xác định rõ ràng, vì nó thiên về thử nghiệm hơn là xây dựng theo quy trình tuyến tính.

- b. Spiral Model:

- Mô hình này kết hợp các vòng lặp (iteration) với quản lý rủi ro, phù hợp cho các dự án phức tạp hoặc có nhiều thay đổi yêu cầu trong suốt quá trình. Nếu yêu cầu hiển thị và cập nhật có thể thay đổi thường xuyên, Spiral Model có thể được xem xét. Tuy nhiên, câu hỏi không đề cập đến rủi ro hoặc thay đổi liên tục, nên nó không phải lựa chọn tối ưu nhất.

- c. V-Model:

- Mô hình V-Model là một biến thể của Waterfall, tập trung vào kiểm thử (testing) song song với từng giai đoạn phát triển (verification và validation). Nó phù hợp khi yêu cầu rõ ràng và cần đảm bảo chất lượng cao. Tuy nhiên, nó không đặc biệt nhấn mạnh vào hiển thị hoặc cập nhật mà tập trung vào kiểm thử, nên không phải lựa chọn hàng đầu.

- d. Waterfall Model ✓:

- Mô hình Waterfall là một quy trình tuyến tính, với các giai đoạn rõ ràng: yêu cầu, thiết kế, xây dựng, kiểm thử, triển khai. Trong giai đoạn Project Construction, khi yêu cầu hiển thị (display) và cập nhật (update) đã được xác định rõ ràng trong các giai đoạn trước (như Initiation và Planning), Waterfall Model phù hợp vì nó cho phép xây dựng và triển khai theo thứ tự cố định. Điều này đảm bảo rằng giao diện hiển thị và chức năng cập nhật được phát triển một cách có hệ thống dựa trên yêu cầu đã phê duyệt.

Khi nào nên chọn các đáp án khác?

- Chọn a (Prototyping Model) nếu yêu cầu hiển thị và cập nhật chưa rõ ràng, cần tạo mẫu thử để lấy phản hồi từ người dùng trước khi xây dựng chính thức.

- Chọn b (Spiral Model) nếu dự án có rủi ro cao hoặc yêu cầu thay đổi thường xuyên trong giai đoạn xây dựng.

- Chọn c (V-Model) nếu trọng tâm là đảm bảo chất lượng kiểm thử song song với phát triển, đặc biệt khi yêu cầu hiển thị và cập nhật cần được xác nhận kỹ lưỡng.

Kết luận:

Đáp án d (Waterfall Model) là phù hợp nhất trong ngữ cảnh câu hỏi, vì giai đoạn Project Construction trong PLC thường diễn ra sau khi yêu cầu đã được xác định rõ ràng (như hiển thị và cập nhật), và Waterfall Model cung cấp một quy trình tuyến tính để thực hiện xây dựng hiệu quả.

**8: Chọn phát biểu sai của quy trình SCRUM?**

Đáp án bạn chọn c (Chỉ cho tester kiểm tra mà không cho khách hàng sử dụng thử) là đúng, vì đây là phát biểu sai về quy trình SCRUM. Dưới đây là phân tích chi tiết:

Phân tích đáp án:

- a. Mỗi lần lặp có quy định thời gian:

- Đúng. Trong SCRUM, mỗi lần lặp (Sprint) có thời gian cố định, thường kéo dài từ 1 đến 4 tuần, được xác định trước khi bắt đầu Sprint. Phát biểu này đúng với quy trình SCRUM.

- b. Mỗi lần lặp đều phải đủ các bước: planning, requirements, design, coding, testing, and documentation:

- Đúng. Một Sprint trong SCRUM bao gồm tất cả các bước cần thiết để tạo ra một sản phẩm hoàn chỉnh (potentially shippable increment), bao gồm lập kế hoạch (planning), xác định yêu cầu (requirements), thiết kế (design), viết mã (coding), kiểm thử (testing), và tài liệu (documentation). Mặc dù SCRUM không bắt buộc chi tiết từng bước, nhưng các hoạt động này thường được thực hiện để đạt được mục tiêu Sprint, nên phát biểu này được xem là đúng.

- c. Chỉ cho tester kiểm tra mà không cho khách hàng sử dụng thử ✓:

- Sai. Trong SCRUM, một trong những nguyên tắc cốt lõi là sự minh bạch và hợp tác với các bên liên quan, bao gồm khách hàng. Cuối mỗi Sprint, nhóm SCRUM tổ chức Sprint Review, nơi sản phẩm được trình bày và khách hàng (hoặc Product Owner) có thể sử dụng thử, đưa ra phản hồi để cải thiện. Việc chỉ cho tester kiểm tra mà không cho khách hàng sử dụng thử đi ngược lại tinh thần của SCRUM, nên phát biểu này sai.

- d. Mục tiêu tạo sản phẩm cuối mỗi lần lặp không có lỗi:

- Đúng. Mục tiêu của mỗi Sprint là tạo ra một sản phẩm hoàn chỉnh, có thể sử dụng được (potentially shippable), và không có lỗi nghiêm trọng. Điều này phù hợp với khái niệm "Done" trong SCRUM, nơi sản phẩm phải đạt chất lượng cao và sẵn sàng để phát hành nếu cần. Phát biểu này đúng.

Khi nào nên chọn các đáp án khác?

- Nếu phát biểu sai liên quan đến thời gian Sprint không cố định, thì a sẽ là đáp án sai.

- Nếu phát biểu sai liên quan đến việc không cần tất cả các bước trong một Sprint (ví dụ: bỏ qua testing), thì b sẽ là đáp án sai.

- Nếu phát biểu sai liên quan đến việc không yêu cầu sản phẩm không có lỗi (thực tế SCRUM yêu cầu sản phẩm đạt chất lượng), thì d sẽ là đáp án sai.

Kết luận:

Phát biểu c (Chỉ cho tester kiểm tra mà không cho khách hàng sử dụng thử) là sai, vì SCRUM khuyến khích khách hàng tham gia Sprint Review để sử dụng thử và đưa phản hồi, không chỉ giới hạn ở tester. Đáp án bạn chọn là chính xác.

**9: Cho biết quan hệ giữa lớp Sản phẩm và Danh mục có Multiplicity là:**

Đáp án bạn chọn b (One-to-Many) là đúng. Dưới đây là phân tích chi tiết:

Phân tích đáp án:

- Quan hệ giữa Sản phẩm (Product) và Danh mục (Category):

- Một Danh mục (Category) có thể chứa nhiều Sản phẩm (Product), ví dụ: Danh mục "Điện thoại" có thể chứa nhiều sản phẩm như iPhone, Samsung, Xiaomi, v.v.

- Ngược lại, một Sản phẩm thường chỉ thuộc về một Danh mục (trong trường hợp thiết kế đơn giản, không xét đến sản phẩm thuộc nhiều danh mục cùng lúc).

- Đây là mối quan hệ One-to-Many (Một-đến-Nhiều): Một Danh mục có thể liên kết với nhiều Sản phẩm, nhưng mỗi Sản phẩm chỉ liên kết với một Danh mục.

- a. One-to-zero-One:

- Mối quan hệ này có nghĩa là một Danh mục có thể không liên kết với Sản phẩm nào (zero) hoặc chỉ liên kết với tối đa một Sản phẩm (one). Điều này không phù hợp, vì một Danh mục thường chứa nhiều Sản phẩm, không giới hạn ở một.

- b. One-to-Many ✓:

- Như đã giải thích, một Danh mục có thể chứa nhiều Sản phẩm, và mỗi Sản phẩm thuộc về một Danh mục. Đây là mối quan hệ phổ biến trong thiết kế cơ sở dữ liệu hoặc mô hình hóa UML, nên đáp án này đúng.

- c. One-to-One:

- Mối quan hệ này có nghĩa là một Danh mục chỉ liên kết với đúng một Sản phẩm, và một Sản phẩm chỉ liên kết với một Danh mục. Điều này không thực tế, vì một Danh mục thường chứa nhiều Sản phẩm, không chỉ một.

- d. Many-to-Many:

- Mối quan hệ này có nghĩa là một Danh mục có thể chứa nhiều Sản phẩm, và một Sản phẩm có thể thuộc về nhiều Danh mục. Trong thực tế, điều này có thể xảy ra (ví dụ: một sản phẩm thuộc cả danh mục "Điện thoại" và "Khuyến mãi"). Tuy nhiên, trong thiết kế cơ bản (như ngữ cảnh câu hỏi), một Sản phẩm thường chỉ thuộc một Danh mục chính, nên Many-to-Many không phải lựa chọn phổ biến ở đây.

Khi nào nên chọn các đáp án khác?

- Chọn a (One-to-zero-One) nếu một Danh mục chỉ có thể không có hoặc tối đa một Sản phẩm (rất hiếm trong thực tế).

- Chọn c (One-to-One) nếu mỗi Danh mục chỉ có đúng một Sản phẩm và mỗi Sản phẩm chỉ thuộc một Danh mục (không hợp lý trong trường hợp này).

- Chọn d (Many-to-Many) nếu Sản phẩm có thể thuộc nhiều Danh mục cùng lúc (ví dụ: một sản phẩm thuộc cả "Điện thoại" và "Phụ kiện"), nhưng điều này thường cần bảng trung gian trong cơ sở dữ liệu và không phải thiết kế cơ bản.

Kết luận:

Đáp án b (One-to-Many) là chính xác, vì trong thiết kế thông thường, một Danh mục có thể chứa nhiều Sản phẩm, và mỗi Sản phẩm thuộc về một Danh mục duy nhất. Đây là mối quan hệ phổ biến và hợp lý nhất trong ngữ cảnh câu hỏi.

**Phân tích thêm**

Tôi hiểu bạn đang hỏi về việc khi nào nên sử dụng các loại mối quan hệ khác (One-to-zero-One, One-to-One, Many-to-Many) thay vì One-to-Many trong ngữ cảnh mối quan hệ giữa lớp Sản phẩm (Product) và Danh mục (Category) hoặc các tình huống tương tự. Dưới đây là phân tích chi tiết về từng loại mối quan hệ và trường hợp áp dụng:

1. One-to-zero-One (Một-đến-Không-hoặc-Một)

- Ý nghĩa: Một đối tượng ở lớp này có thể liên kết với không hoặc tối đa một đối tượng ở lớp kia.

- Khi nào sử dụng?

- Khi một thực thể có thể tồn tại mà không cần liên kết với thực thể khác, và nếu có liên kết thì chỉ với một đối tượng duy nhất.

- Ví dụ:

- Quan hệ giữa Khách hàng (Customer) và Số điện thoại (Phone Number): Một khách hàng có thể không có số điện thoại (zero) hoặc chỉ có một số điện thoại (one) chính thức trong hệ thống đơn giản.

- Quan hệ giữa Nhân viên (Employee) và Phòng làm việc (Office Room): Một nhân viên có thể không được cấp phòng riêng (zero) hoặc chỉ được cấp một phòng (one).

- Lưu ý: Đây là trường hợp hiếm trong thiết kế, thường chỉ áp dụng khi giới hạn mối quan hệ rất chặt chẽ.

2. One-to-One (Một-đến-Một)

- Ý nghĩa: Một đối tượng ở lớp này chỉ liên kết với đúng một đối tượng ở lớp kia, và ngược lại.

- Khi nào sử dụng?

- Khi mỗi thực thể trong hai lớp chỉ có thể liên kết với một thực thể duy nhất trong lớp kia, thường do ràng buộc kinh doanh hoặc thiết kế đặc thù.

- Ví dụ:

- Quan hệ giữa Nhân viên (Employee) và Hồ sơ cá nhân (Personal Profile): Mỗi nhân viên chỉ có một hồ sơ cá nhân, và mỗi hồ sơ chỉ thuộc về một nhân viên.

- Quan hệ giữa Xe hơi (Car) và Số khung (Chassis Number): Mỗi xe có một số khung duy nhất, và mỗi số khung chỉ thuộc về một xe.

- Lưu ý: One-to-One thường được dùng để tách dữ liệu nhạy cảm (như thông tin cá nhân) hoặc tối ưu hóa cơ sở dữ liệu.

3. Many-to-Many (Nhiều-đến-Nhiều)

- Ý nghĩa: Một đối tượng ở lớp này có thể liên kết với nhiều đối tượng ở lớp kia, và ngược lại.

- Khi nào sử dụng?

- Khi một thực thể có thể liên kết với nhiều thực thể khác, và thực thể kia cũng có thể liên kết với nhiều thực thể trong lớp đầu tiên, thường yêu cầu bảng trung gian trong cơ sở dữ liệu.

- Ví dụ:

- Quan hệ giữa Sinh viên (Student) và Khóa học (Course): Một sinh viên có thể đăng ký nhiều khóa học, và một khóa học có nhiều sinh viên.

- Quan hệ giữa Sản phẩm (Product) và Danh mục (Category) (trong trường hợp mở rộng): Một sản phẩm có thể thuộc nhiều danh mục (ví dụ: "Điện thoại" và "Khuyến mãi"), và một danh mục có thể chứa nhiều sản phẩm.

- Lưu ý: Trong thực tế, Many-to-Many cần được chuyển thành hai mối quan hệ One-to-Many thông qua bảng trung gian (ví dụ: bảng "Student\_Course" để lưu quan hệ giữa sinh viên và khóa học).

So sánh với One-to-Many (trong ngữ cảnh Sản phẩm và Danh mục):

- One-to-Many (như bạn đã chọn) phù hợp khi một Danh mục chứa nhiều Sản phẩm, nhưng mỗi Sản phẩm chỉ thuộc một Danh mục. Đây là thiết kế cơ bản và phổ biến.

- Many-to-Many sẽ phù hợp hơn nếu hệ thống cho phép một Sản phẩm thuộc nhiều Danh mục (ví dụ: sản phẩm "iPhone 14" thuộc cả "Điện thoại" và "Sản phẩm mới"), điều này phổ biến trong các hệ thống thương mại điện tử phức tạp.

- One-to-One hoặc One-to-zero-One không phù hợp vì Danh mục thường chứa nhiều Sản phẩm, không giới hạn ở một hoặc không.

Khi nào nên chuyển sang dùng các mối quan hệ khác?

- Chuyển sang One-to-zero-One: Nếu hệ thống yêu cầu Danh mục chỉ có thể chứa không hoặc một Sản phẩm (ví dụ: danh mục đặc biệt chỉ cho một sản phẩm nổi bật).

- Chuyển sang One-to-One: Nếu mỗi Danh mục chỉ chứa một Sản phẩm duy nhất và mỗi Sản phẩm chỉ thuộc một Danh mục (hiếm gặp, có thể áp dụng trong thiết kế đặc thù).

- Chuyển sang Many-to-Many: Nếu Sản phẩm có thể thuộc nhiều Danh mục và Danh mục có thể chứa nhiều Sản phẩm (cần bảng trung gian để quản lý quan hệ).

Kết luận:

Bạn đã chọn đúng One-to-Many cho mối quan hệ giữa Sản phẩm và Danh mục trong ngữ cảnh cơ bản. Các mối quan hệ khác (One-to-zero-One, One-to-One, Many-to-Many) chỉ nên sử dụng khi yêu cầu kinh doanh hoặc thiết kế hệ thống có sự khác biệt rõ rệt, như được mô tả ở trên. Nếu bạn có ngữ cảnh cụ thể hơn, mình có thể phân tích thêm!

**11: Trong sơ đồ ERD, chọn phát biểu sai:**

Đáp án bạn chọn d (Có biểu diễn các phương thức của thực thể) là đúng, vì đây là phát biểu sai về sơ đồ ERD. Dưới đây là phân tích chi tiết và bổ sung kiến thức về ERD:

Phân tích đáp án:

- a. Thực thể và thuộc tính được biểu diễn bằng hình chữ nhật, hình tròn:

- Đúng. Trong sơ đồ ERD (Entity-Relationship Diagram):

- Thực thể (Entity) được biểu diễn bằng hình chữ nhật. Ví dụ: "Khách hàng", "Sản phẩm".

- Thuộc tính (Attribute) của thực thể được biểu diễn bằng hình tròn (hoặc hình elip), kết nối với thực thể bằng đường thẳng. Ví dụ: Thuộc tính "Tên" hoặc "ID" của thực thể "Khách hàng".

- Phát biểu này đúng với chuẩn ERD.

- b. Quan hệ giữa 2 thực thể được biểu diễn bằng hình thoi:

- Đúng. Quan hệ (Relationship) giữa các thực thể trong ERD được biểu diễn bằng hình thoi. Ví dụ: Quan hệ "Mua" giữa thực thể "Khách hàng" và "Sản phẩm".

- Phát biểu này đúng.

- c. Số degree giữa các thực thể thường là 2:

- Đúng. Trong ERD, degree của một quan hệ là số lượng thực thể tham gia vào quan hệ đó:

- Quan hệ binary (degree = 2) là phổ biến nhất, ví dụ: Quan hệ "Mua" giữa "Khách hàng" và "Sản phẩm".

- Ngoài ra, có thể có quan hệ unary (degree = 1, ví dụ: "Quản lý" giữa "Nhân viên" với chính nó) hoặc ternary (degree = 3, ví dụ: "Đăng ký" giữa "Sinh viên", "Khóa học", và "Giảng viên").

- Tuy nhiên, quan hệ degree = 2 là trường hợp thông dụng nhất, nên phát biểu này đúng.

- d. Có biểu diễn các phương thức của thực thể ✓:

- Sai. Sơ đồ ERD không biểu diễn phương thức (method) của thực thể. ERD chỉ tập trung vào:

- Thực thể (Entity): Các đối tượng chính (như bảng trong cơ sở dữ liệu).

- Thuộc tính (Attribute): Các đặc điểm của thực thể (như cột trong bảng).

- Quan hệ (Relationship): Mối liên kết giữa các thực thể.

- Phương thức (hành vi, như các hàm hoặc chức năng) là khái niệm thuộc về lập trình hướng đối tượng (OOP) và thường được biểu diễn trong sơ đồ lớp (Class Diagram) của UML, không phải ERD.

Khi nào nên chọn các đáp án khác?

- Nếu phát biểu sai liên quan đến hình dạng biểu diễn (ví dụ: nói thực thể là hình tròn), thì a sẽ là đáp án sai.

- Nếu phát biểu sai liên quan đến hình thoi (ví dụ: nói quan hệ là hình chữ nhật), thì b sẽ là đáp án sai.

- Nếu phát biểu sai về degree (ví dụ: nói degree luôn phải là 3), thì c sẽ là đáp án sai.

---

**Kiến thức bổ sung về ERD (Entity-Relationship Diagram):**

# 1. Mục đích của ERD:

- ERD là một công cụ mô hình hóa dữ liệu, dùng để thiết kế và biểu diễn cấu trúc của cơ sở dữ liệu.

- Nó giúp:

- Hiểu rõ các thực thể và mối quan hệ giữa chúng.

- Xây dựng lược đồ cơ sở dữ liệu trước khi triển khai (ví dụ: tạo bảng trong SQL).

- Đảm bảo tính nhất quán và tránh dư thừa dữ liệu.

# 2. Các thành phần chính trong ERD:

- Thực thể (Entity):

- Là các đối tượng chính trong hệ thống, ví dụ: "Khách hàng", "Sản phẩm", "Đơn hàng".

- Gồm thực thể mạnh (Strong Entity) và thực thể yếu (Weak Entity):

- Thực thể mạnh: Có khóa chính độc lập (ví dụ: "Khách hàng" với khóa chính là "ID\_KhachHang").

- Thực thể yếu: Phụ thuộc vào thực thể mạnh, không có khóa chính độc lập (ví dụ: "Chi tiết đơn hàng" phụ thuộc "Đơn hàng").

- Biểu diễn: Hình chữ nhật (thực thể yếu thường dùng hình chữ nhật kép).

- Thuộc tính (Attribute):

- Là các đặc điểm của thực thể, ví dụ: "Tên", "Giá", "Ngày sinh".

- Loại thuộc tính:

- Thuộc tính đơn (Simple): Không thể chia nhỏ (ví dụ: "Tên").

- Thuộc tính phức hợp (Composite): Có thể chia nhỏ (ví dụ: "Địa chỉ" gồm "Số nhà", "Đường", "Thành phố").

- Thuộc tính khóa (Key): Dùng để xác định duy nhất (ví dụ: "ID").

- Thuộc tính đa giá trị (Multivalued): Có nhiều giá trị (ví dụ: "Số điện thoại" của khách hàng, biểu diễn bằng hình elip kép).

- Biểu diễn: Hình elip (hoặc hình tròn), kết nối với thực thể.

- Quan hệ (Relationship):

- Là mối liên kết giữa các thực thể, ví dụ: "Khách hàng" đặt "Đơn hàng".

- Degree của quan hệ:

- Unary (degree = 1): Quan hệ trong cùng một thực thể (ví dụ: "Nhân viên" quản lý "Nhân viên").

- Binary (degree = 2): Quan hệ giữa 2 thực thể (phổ biến nhất).

- Ternary (degree = 3): Quan hệ giữa 3 thực thể.

- Multiplicity (Tính đa dạng):

- One-to-One (1:1): Ví dụ: "Người" và "CMND".

- One-to-Many (1:N): Ví dụ: "Danh mục" và "Sản phẩm".

- Many-to-Many (N:N): Ví dụ: "Sinh viên" và "Khóa học".

- Biểu diễn: Hình thoi.

# 3. Ví dụ minh họa:

- Giả sử bạn thiết kế ERD cho một hệ thống thương mại điện tử:

- Thực thể: "Khách hàng", "Sản phẩm", "Đơn hàng".

- Thuộc tính:

- "Khách hàng": ID\_KhachHang (khóa chính), Tên, Địa chỉ.

- "Sản phẩm": ID\_SanPham (khóa chính), Tên, Giá.

- "Đơn hàng": ID\_DonHang (khóa chính), Ngày đặt.

- Quan hệ:

- "Khách hàng" đặt "Đơn hàng" (One-to-Many).

- "Đơn hàng" chứa "Sản phẩm" (Many-to-Many, cần bảng trung gian "Chi tiết đơn hàng").

# 4. Sự khác biệt giữa ERD và Class Diagram:

- Như đã đề cập, ERD không biểu diễn phương thức (method). Phương thức là đặc điểm của Class Diagram trong UML, vì:

- ERD tập trung vào cơ sở dữ liệu (dữ liệu tĩnh: thực thể, thuộc tính, quan hệ).

- Class Diagram tập trung vào lập trình hướng đối tượng (bao gồm dữ liệu và hành vi: thuộc tính và phương thức).

# 5. Ứng dụng thực tế:

- ERD thường được sử dụng trong:

- Thiết kế cơ sở dữ liệu trước khi triển khai (ví dụ: tạo bảng trong MySQL, PostgreSQL).

- Giao tiếp giữa các nhóm (developer, designer, analyst) để thống nhất cấu trúc dữ liệu.

- Phát hiện lỗi thiết kế sớm (ví dụ: dư thừa dữ liệu, quan hệ không hợp lý).

Đáp án bạn chọn c (n:m) là đúng. Dưới đây là phân tích chi tiết và bổ sung thêm kiến thức về cardinality trong mối quan hệ giữa các thực thể.

---

Phân tích đáp án:

- Quan hệ giữa Khách hàng (Customer) và Sản phẩm (Product):

- Một Khách hàng có thể mua nhiều Sản phẩm. Ví dụ: Khách hàng A có thể mua iPhone, tai nghe, và ốp lưng.

- Một Sản phẩm có thể được nhiều Khách hàng mua. Ví dụ: Sản phẩm iPhone có thể được mua bởi Khách hàng A, B, và C.

- Đây là mối quan hệ Many-to-Many (n:m): Nhiều Khách hàng có thể liên kết với nhiều Sản phẩm và ngược lại.

- a. 1:1 (One-to-One):

- Mối quan hệ này có nghĩa là một Khách hàng chỉ mua đúng một Sản phẩm, và một Sản phẩm chỉ được mua bởi đúng một Khách hàng. Điều này không thực tế trong ngữ cảnh mua sắm, vì một Khách hàng thường mua nhiều Sản phẩm và một Sản phẩm có thể được nhiều Khách hàng mua.

- b. 1:n (One-to-Many):

- Mối quan hệ này có nghĩa là một Khách hàng có thể mua nhiều Sản phẩm, nhưng một Sản phẩm chỉ được mua bởi một Khách hàng duy nhất. Điều này không đúng, vì một Sản phẩm (như iPhone) có thể được nhiều Khách hàng mua, không chỉ một.

- c. n:m (Many-to-Many) ✓:

- Như đã giải thích, một Khách hàng có thể mua nhiều Sản phẩm, và một Sản phẩm có thể được nhiều Khách hàng mua. Đây là mối quan hệ phổ biến trong hệ thống thương mại điện tử, nên đáp án này đúng.

- Trong thực tế, mối quan hệ Many-to-Many thường cần một thực thể trung gian (junction entity) để quản lý, ví dụ: thực thể "Đơn hàng" (Order) hoặc "Chi tiết đơn hàng" (Order Detail) để lưu thông tin về việc Khách hàng nào mua Sản phẩm nào.

- d. Không có (No relationship):

- Rõ ràng giữa Khách hàng và Sản phẩm có mối quan hệ (qua hành vi mua sắm), nên đáp án này sai.

---

Khi nào nên chọn các đáp án khác?

- Chọn a (1:1) nếu mỗi Khách hàng chỉ mua đúng một Sản phẩm và mỗi Sản phẩm chỉ được mua bởi một Khách hàng (rất hiếm, ví dụ: trong hệ thống bán sản phẩm độc quyền như vé số trúng thưởng duy nhất).

- Chọn b (1:n) nếu một Khách hàng có thể mua nhiều Sản phẩm, nhưng mỗi Sản phẩm chỉ được mua bởi một Khách hàng duy nhất (cũng hiếm, ví dụ: trong hệ thống đấu giá mà mỗi sản phẩm chỉ được bán cho một người).

- Chọn d (Không có) nếu không có bất kỳ mối quan hệ nào giữa hai thực thể, ví dụ: giữa Khách hàng và một thực thể không liên quan như "Nhà cung cấp".

---

Kiến thức bổ sung về Cardinality trong ERD:

# 1. Cardinality là gì?

- Cardinality (tính đa dạng) trong sơ đồ ERD (Entity-Relationship Diagram) mô tả số lượng thực thể ở một phía của quan hệ có thể liên kết với số lượng thực thể ở phía bên kia.

- Nó được biểu diễn dưới dạng tỷ lệ (ví dụ: 1:1, 1:n, n:m) và thường đi kèm với multiplicity (giới hạn cụ thể, như 0..1, 1..\*, \*..\*).

# 2. Các loại Cardinality:

- 1:1 (One-to-One):

- Một thực thể ở lớp này chỉ liên kết với đúng một thực thể ở lớp kia.

- Ví dụ: "Người" và "CMND" (mỗi người chỉ có một CMND, mỗi CMND chỉ thuộc về một người).

- 1:n (One-to-Many):

- Một thực thể ở lớp này có thể liên kết với nhiều thực thể ở lớp kia, nhưng mỗi thực thể ở lớp kia chỉ liên kết với một thực thể ở lớp này.

- Ví dụ: "Danh mục" và "Sản phẩm" (một Danh mục chứa nhiều Sản phẩm, nhưng mỗi Sản phẩm chỉ thuộc một Danh mục).

- n:m (Many-to-Many):

- Một thực thể ở lớp này có thể liên kết với nhiều thực thể ở lớp kia, và ngược lại.

- Ví dụ: "Sinh viên" và "Khóa học" (một Sinh viên có thể học nhiều Khóa học, một Khóa học có nhiều Sinh viên).

- Trong cơ sở dữ liệu, mối quan hệ n:m thường được triển khai bằng cách tạo một bảng trung gian (junction table). Ví dụ: Bảng "Chi tiết đơn hàng" để lưu thông tin về Khách hàng nào mua Sản phẩm nào.

# 3. Multiplicity (Giới hạn cụ thể của Cardinality):

- Cardinality thường đi kèm với multiplicity để chỉ rõ số lượng cụ thể:

- 0..1: Không hoặc tối đa một (zero or one).

- 1..1: Đúng một (exactly one).

- 0..\*: Không hoặc nhiều (zero or many).

- 1..\*: Một hoặc nhiều (one or many).

- Ví dụ:

- Quan hệ giữa "Khách hàng" và "Đơn hàng" có thể là 1..\* (một Khách hàng có thể có một hoặc nhiều Đơn hàng) và 0..1 (một Đơn hàng chỉ thuộc về một Khách hàng hoặc không có Khách hàng nào, nếu là đơn hàng thử nghiệm).

# 4. Cách triển khai n:m trong cơ sở dữ liệu:

- Mối quan hệ Many-to-Many (n:m) không thể triển khai trực tiếp trong cơ sở dữ liệu quan hệ (như MySQL) mà cần một bảng trung gian.

- Trong trường hợp Khách hàng và Sản phẩm:

- Bảng Khách hàng: Lưu thông tin khách hàng (ID\_KhachHang, Tên, Địa chỉ).

- Bảng Sản phẩm: Lưu thông tin sản phẩm (ID\_SanPham, Tên, Giá).

- Bảng trung gian (ví dụ: Chi tiết đơn hàng): Lưu thông tin về việc Khách hàng nào mua Sản phẩm nào (ID\_KhachHang, ID\_SanPham, Số lượng, Ngày mua).

- Bảng trung gian chuyển mối quan hệ n:m thành hai mối quan hệ 1:n:

- "Khách hàng" (1) - (n) "Chi tiết đơn hàng".

- "Sản phẩm" (1) - (n) "Chi tiết đơn hàng".

# 5. Ví dụ thực tế:

- Quan hệ 1:1: "Người" và "Hộ chiếu" (mỗi người chỉ có một hộ chiếu, mỗi hộ chiếu chỉ thuộc về một người).

- Quan hệ 1:n: "Phòng ban" và "Nhân viên" (một Phòng ban có nhiều Nhân viên, nhưng mỗi Nhân viên chỉ thuộc một Phòng ban).

- Quan hệ n:m: "Tác giả" và "Sách" (một Tác giả có thể viết nhiều Sách, một Sách có thể do nhiều Tác giả viết).

# 6. Ứng dụng của Cardinality:

- Giúp thiết kế cơ sở dữ liệu chính xác, tránh dư thừa hoặc lỗi logic.

- Xác định các ràng buộc (constraints) trong cơ sở dữ liệu, như khóa ngoại (foreign key).

- Hỗ trợ lập kế hoạch cho các truy vấn (query) hiệu quả, ví dụ: biết được một Sản phẩm có thể liên quan đến nhiều Khách hàng sẽ ảnh hưởng đến cách viết câu lệnh JOIN.

---

Kết luận:

Đáp án c (n:m) là chính xác vì mối quan hệ giữa Khách hàng và Sản phẩm là Many-to-Many: một Khách hàng có thể mua nhiều Sản phẩm, và một Sản phẩm có thể được nhiều Khách hàng mua. Phần kiến thức bổ sung về cardinality và cách triển khai mối quan hệ n:m trong cơ sở dữ liệu hy vọng sẽ giúp bạn hiểu rõ hơn về thiết kế ERD và ứng dụng thực tế! Nếu cần thêm ví dụ hoặc giải thích, hãy cho mình biết nhé.

**16: Điều gì được thực hiện trong giai đoạn thử nghiệm?**

Đáp án bạn chọn c (Mã nay được kiểm tra dựa trên thông số kỹ thuật) là đúng. Dưới đây là phân tích chi tiết:

Phân tích đáp án:

- a. Thực hiện các thay đổi khi phần mềm đã được tạo:

- Việc thực hiện thay đổi thuộc về giai đoạn bảo trì (maintenance) hoặc kiểm thử (testing) ở mức sửa lỗi, nhưng không phải trọng tâm chính của giai đoạn thử nghiệm. Giai đoạn thử nghiệm chủ yếu kiểm tra chứ không tập trung vào thay đổi, nên đáp án này sai.

- b. Lập kế hoạch giải pháp, giao diện phần mềm:

- Lập kế hoạch và thiết kế giao diện là hoạt động của giai đoạn thiết kế (design), diễn ra trước khi mã hóa và thử nghiệm. Do đó, đáp án này không liên quan đến giai đoạn thử nghiệm.

- c. Mã nay được kiểm tra dựa trên thông số kỹ thuật ✓:

- Đúng. Giai đoạn thử nghiệm (testing) trong vòng đời phát triển phần mềm (SDLC) tập trung vào việc kiểm tra mã nguồn dựa trên các thông số kỹ thuật (specifications) đã được xác định trước đó. Điều này bao gồm:

- Kiểm tra tính đúng đắn (verification).

- Đảm bảo phần mềm đáp ứng yêu cầu (validation).

- Phát hiện lỗi (bugs) và đảm bảo chất lượng trước khi phát hành.

- Đây là mục tiêu chính của giai đoạn thử nghiệm, nên đáp án này đúng.

- d. Đảm bảo các yêu cầu được hiểu rõ:

- Việc đảm bảo yêu cầu được hiểu rõ thuộc về giai đoạn phân tích yêu cầu (requirements analysis), diễn ra trước cả thiết kế và thử nghiệm. Giai đoạn thử nghiệm sử dụng các yêu cầu đã được xác định để kiểm tra, không phải để hiểu chúng, nên đáp án này sai.

Khi nào nên chọn các đáp án khác?

- Chọn a nếu câu hỏi liên quan đến giai đoạn bảo trì hoặc sửa lỗi sau khi phát hành phần mềm.

- Chọn b nếu câu hỏi hỏi về giai đoạn thiết kế.

- Chọn d nếu câu hỏi liên quan đến giai đoạn phân tích yêu cầu.

Kết luận:

Đáp án c (Mã nay được kiểm tra dựa trên thông số kỹ thuật) là chính xác, vì giai đoạn thử nghiệm tập trung vào việc kiểm tra mã nguồn để đảm bảo nó hoạt động đúng theo các thông số kỹ thuật đã được định nghĩa trước đó.

**Dựa trên bối cảnh các câu hỏi trước về vòng đời phát triển phần mềm (SDLC) và Project Life Cycle (PLC), tôi sẽ cung cấp danh sách đầy đủ các giai đoạn trong một quy trình phát triển phần mềm tiêu chuẩn, bao gồm cả các giai đoạn đã đề cập trước đó. Tuy nhiên, vì câu hỏi không chỉ định rõ mô hình cụ thể (như Waterfall, Agile, hay Scrum), tôi sẽ trình bày một mô hình đầy đủ dựa trên Waterfall Model (mô hình thác nước) làm nền tảng, sau đó bổ sung các biến thể nếu cần. Đây là danh sách đầy đủ và phổ biến nhất:**

Các giai đoạn trong vòng đời phát triển phần mềm (SDLC):

1. Project Initiation (Khởi tạo dự án)

- Xác định mục tiêu, phạm vi, và tính khả thi của dự án.

- Đánh giá nhu cầu ban đầu và lập kế hoạch sơ bộ.

- Định nghĩa các bên liên quan (stakeholders).

2. Requirements Analysis (Phân tích yêu cầu)

- Thu thập, phân tích, và ghi nhận chi tiết các yêu cầu của người dùng và hệ thống.

- Đảm bảo các yêu cầu được hiểu rõ và được phê duyệt.

3. System Design (Thiết kế hệ thống)

- Lập kế hoạch giải pháp kỹ thuật (architecture design).

- Thiết kế giao diện người dùng (UI/UX) và cơ sở dữ liệu.

- Tạo các sơ đồ như ERD, sơ đồ luồng dữ liệu (DFD).

4. Project Construction / Implementation (Xây dựng / Triển khai)

- Viết mã nguồn (coding) dựa trên thiết kế.

- Phát triển các thành phần phần mềm.

5. Testing (Thử nghiệm)

- Kiểm tra mã dựa trên thông số kỹ thuật (verification và validation).

- Phát hiện và sửa lỗi (bugs).

- Đảm bảo phần mềm đáp ứng yêu cầu và hoạt động ổn định.

6. Deployment / Project Transition (Triển khai / Chuyển giao)

- Phát hành phần mềm cho người dùng hoặc khách hàng.

- Cài đặt và cấu hình hệ thống tại môi trường thực tế.

7. Maintenance (Bảo trì)

- Thực hiện các thay đổi, cập nhật, hoặc sửa lỗi sau khi phần mềm được triển khai.

- Hỗ trợ người dùng và cải tiến hệ thống dựa trên phản hồi.

---

Ghi chú:

- Thứ tự: Các giai đoạn trên thường theo trình tự tuyến tính trong mô hình Waterfall (1-2-3-4-5-6-7). Tuy nhiên, trong các mô hình Agile hoặc Scrum:

- Các giai đoạn như Testing và Construction có thể lặp lại trong từng Sprint.

- Maintenance có thể diễn ra liên tục và tích hợp với các giai đoạn khác.

- Biến thể: Tùy thuộc vào tổ chức hoặc dự án, số lượng giai đoạn có thể thay đổi (ví dụ: gộp Initiation và Requirements, hoặc tách Testing thành Unit Testing, Integration Testing, System Testing).

Liên hệ với các câu hỏi trước:

- Giai đoạn Project Initiation và Planning and Prototyping (câu 6) tương ứng với bước 1 và một phần của bước 3.

- Project Construction (câu 7) là bước 4.

- Testing (câu 16) là bước 5.

- Project Transition and Release (câu 6) là bước 6.

- Các hoạt động như thay đổi phần mềm (câu 5, 16) thuộc bước 7 (Maintenance).

Kiến thức bổ sung:

- Waterfall Model: Quy trình tuyến tính, từng giai đoạn hoàn thành trước khi chuyển sang giai đoạn tiếp theo, phù hợp với dự án có yêu cầu cố định.

- Agile Model: Linh hoạt hơn, chia dự án thành các vòng lặp (iteration/sprint), với các giai đoạn lặp lại như Planning, Construction, Testing.

- Spiral Model: Kết hợp Waterfall và vòng lặp, tập trung vào quản lý rủi ro, phù hợp với dự án phức tạp.

- V-Model: Biến thể của Waterfall, nhấn mạnh kiểm thử song song với từng giai đoạn phát triển.

Hy vọng danh sách đầy đủ các giai đoạn và kiến thức bổ sung này giúp bạn nắm rõ hơn về vòng đời phát triển phần mềm! Nếu bạn muốn đi sâu vào mô hình cụ thể hoặc cần ví dụ, hãy cho mình biết nhé.

Đáp án bạn chọn là \*\*a (\* và 0..\*)\*\*, nhưng dựa trên phân tích, đáp án đúng thực tế là \*\*b (1 và 1..\*)\*\*. Dưới đây là phân tích chi tiết và lý do tại sao lựa chọn của bạn không chính xác:

---

**17: Chọn multiplicity giữa 2 lớp Khách hàng và Hóa đơn:**

Phân tích đáp án:

- \*\*Quan hệ giữa Khách hàng (Customer) và Hóa đơn (Invoice)\*\*:

- Một \*\*Khách hàng\*\* có thể có nhiều \*\*Hóa đơn\*\*. Ví dụ: Khách hàng A có thể có nhiều hóa đơn từ các lần mua sắm khác nhau.

- Một \*\*Hóa đơn\*\* thường chỉ thuộc về một \*\*Khách hàng\*\*. Ví dụ: Một hóa đơn cụ thể chỉ được tạo ra cho một khách hàng duy nhất (trong thiết kế cơ bản, không xét đến các trường hợp phức tạp như hóa đơn chung).

- \*\*Multiplicity\*\*:

- Multiplicity mô tả số lượng đối tượng của một lớp có thể liên kết với đối tượng của lớp kia.

- Ký hiệu:

- \*\*1\*\*: Đúng một.

- \*\*0..1\*\*: Không hoặc tối đa một.

- \*\*0..\*\*\* hoặc \*\*\\*\*\*: Không hoặc nhiều (zero or many).

- \*\*1..\*\*\*: Một hoặc nhiều (one or many).

- \*\*a. \* và 0..\* (tức là 0..\* và 0..\*)\*\*:

- Nghĩa là một Khách hàng có thể có không hoặc nhiều Hóa đơn (0..\*), và một Hóa đơn có thể thuộc về không hoặc nhiều Khách hàng (0..\*).

- Phần đầu (Khách hàng có thể có nhiều Hóa đơn) là đúng.

- Nhưng phần sau (một Hóa đơn thuộc về không hoặc nhiều Khách hàng) không thực tế trong thiết kế cơ bản, vì một Hóa đơn thường chỉ thuộc về một Khách hàng duy nhất (ví dụ: hóa đơn mua hàng chỉ ghi nhận một người mua). Do đó, đáp án này sai.

- \*\*b. 1 và 1..\*\*\*:

- Nghĩa là một Hóa đơn phải thuộc về đúng một Khách hàng (1), và một Khách hàng có thể có một hoặc nhiều Hóa đơn (1..\*).

- Điều này phù hợp với thực tế:

- Một Hóa đơn luôn được gán cho một Khách hàng (1).

- Một Khách hàng có thể có ít nhất một hoặc nhiều Hóa đơn (1..\*), tùy thuộc vào số lần mua sắm.

- Đây là thiết kế phổ biến trong hệ thống thương mại điện tử hoặc bán lẻ, nên đáp án này đúng.

- \*\*c. 1 và \*\*\*:

- Nghĩa là một Hóa đơn thuộc về đúng một Khách hàng (1), và một Khách hàng có thể có không hoặc nhiều Hóa đơn (0..\*).

- Phần đầu (Hóa đơn thuộc về một Khách hàng) là đúng.

- Phần sau (Khách hàng có thể có 0 Hóa đơn) cũng có thể đúng trong một số hệ thống, nhưng trong ngữ cảnh thông thường (hệ thống quản lý hóa đơn), một Khách hàng thường được giả định phải có ít nhất một Hóa đơn (1..\*) nếu họ tồn tại trong hệ thống. Nếu hệ thống cho phép Khách hàng không có hóa đơn (ví dụ: khách hàng mới chưa mua), thì đáp án này cũng có thể đúng. Tuy nhiên, trong thiết kế chuẩn, \*\*1..\*\*\* được ưu tiên hơn.

- \*\*d. Không có\*\*:

- Rõ ràng giữa Khách hàng và Hóa đơn có mối quan hệ (Hóa đơn được tạo bởi Khách hàng), nên đáp án này sai.

---

- Đáp án đúng là \*\*b (1 và 1..\*)\*\*, vì:

- Một Hóa đơn phải thuộc về đúng một Khách hàng (1).

- Một Khách hàng có thể có ít nhất một hoặc nhiều Hóa đơn (1..\*), phản ánh thực tế rằng khách hàng trong hệ thống thường có lịch sử mua hàng.

---

Khi nào nên chọn các đáp án khác?

- Chọn \*\*a (\* và 0..\*)\*\* nếu một Hóa đơn có thể thuộc về nhiều Khách hàng (ví dụ: hóa đơn chung trong trường hợp mua sắm nhóm), nhưng điều này hiếm và không phải thiết kế cơ bản.

- Chọn \*\*c (1 và \*)\*\* nếu hệ thống cho phép Khách hàng không có hóa đơn (ví dụ: khách hàng mới chưa mua hàng).

- Chọn \*\*d (Không có)\*\* nếu không có mối quan hệ giữa hai lớp, nhưng điều này không áp dụng ở đây.

---

Kiến thức bổ sung về Multiplicity trong thiết kế hệ thống:

# 1. \*\*Multiplicity và Cardinality\*\*:

- \*\*Cardinality\*\* (tính đa dạng) mô tả số lượng tổng quát của mối quan hệ (ví dụ: 1:1, 1:n, n:m). Trong câu hỏi trước (câu 12), bạn đã chọn cardinality giữa Khách hàng và Sản phẩm là \*\*n:m\*\*.

- \*\*Multiplicity\*\* là chi tiết hơn của cardinality, chỉ rõ số lượng cụ thể (ví dụ: 0..1, 1..\*, 0..\*). Trong câu này, bạn đang chọn multiplicity giữa Khách hàng và Hóa đơn.

# 2. \*\*Cách biểu diễn Multiplicity trong ERD\*\*:

- Multiplicity thường được ghi ở hai đầu của đường quan hệ trong sơ đồ ERD:

- Ví dụ: Quan hệ giữa "Khách hàng" và "Hóa đơn":

- Phía "Khách hàng": \*\*1\*\* (một Hóa đơn thuộc về đúng một Khách hàng).

- Phía "Hóa đơn": \*\*1..\*\*\* (một Khách hàng có một hoặc nhiều Hóa đơn).

# 3. \*\*Ví dụ minh họa\*\*:

- \*\*Khách hàng và Hóa đơn\*\* (như câu hỏi này):

- Multiplicity: \*\*1\*\* (Hóa đơn) - \*\*1..\*\*\* (Khách hàng).

- Trong cơ sở dữ liệu: Bảng "Hóa đơn" sẽ có khóa ngoại (foreign key) tham chiếu đến "Khách hàng", ví dụ: cột "ID\_KhachHang".

- \*\*Khác biệt với Khách hàng và Sản phẩm (câu 12)\*\*:

- Quan hệ giữa Khách hàng và Sản phẩm là \*\*n:m\*\*, nên cần bảng trung gian (ví dụ: "Chi tiết đơn hàng").

- Multiplicity có thể là \*\*0..\*\*\* (Khách hàng) - \*\*0..\*\*\* (Sản phẩm), vì một Khách hàng có thể không mua sản phẩm nào, và một Sản phẩm có thể không được ai mua.

# 4. \*\*Các trường hợp Multiplicity khác\*\*:

- \*\*0..1\*\*: Không hoặc tối đa một. Ví dụ: Quan hệ giữa "Nhân viên" và "Phòng làm việc" (một Nhân viên có thể không có phòng hoặc có tối đa một phòng).

- \*\*1..1\*\*: Đúng một. Ví dụ: Quan hệ giữa "Người" và "CMND" (mỗi người có đúng một CMND).

- \*\*0..\*\*\*: Không hoặc nhiều. Ví dụ: Quan hệ giữa "Khách hàng" và "Địa chỉ giao hàng" (một Khách hàng có thể có không hoặc nhiều địa chỉ).

# 5. \*\*Ứng dụng thực tế\*\*:

- Multiplicity giúp thiết kế cơ sở dữ liệu chính xác, đảm bảo các ràng buộc (constraints) được áp dụng đúng:

- Ví dụ: Multiplicity \*\*1\*\* ở phía Hóa đơn đảm bảo rằng mỗi Hóa đơn phải có một Khách hàng (không được để trống).

- Multiplicity \*\*1..\*\*\* ở phía Khách hàng có thể yêu cầu kiểm tra logic (ví dụ: đảm bảo Khách hàng phải có ít nhất một Hóa đơn trong hệ thống quản lý hóa đơn).

---

Kết luận:

Đáp án đúng là \*\*b (1 và 1..\*)\*\*, vì:

- Một Hóa đơn thuộc về đúng một Khách hàng (1).

- Một Khách hàng có thể có một hoặc nhiều Hóa đơn (1..\*), phản ánh thiết kế logic và phổ biến trong hệ thống quản lý hóa đơn.

**Câu 2:**

a. Giải thích các lý do tại sao phải viết phần mềm theo mô hình phân lớp?

Việc xây dựng phần mềm theo mô hình phân lớp là một nguyên tắc quan trọng trong lập trình hiện đại, mang lại nhiều lợi ích như sau:

Tính tổ chức và dễ bảo trì:

Mỗi lớp đảm nhận một nhiệm vụ cụ thể, giúp hệ thống rõ ràng, dễ hiểu và dễ quản lý hơn.

Khi cần sửa lỗi hoặc nâng cấp, chỉ cần thay đổi ở lớp liên quan mà không ảnh hưởng đến toàn bộ hệ thống.

Tính tái sử dụng (Reusability):

Các lớp có thể được sử dụng lại ở nhiều nơi khác nhau hoặc trong các dự án khác nhờ vào việc tách biệt chức năng.

Tính mở rộng (Scalability):

Hệ thống dễ dàng mở rộng khi nhu cầu phát triển tăng lên, vì mỗi lớp độc lập nên có thể phát triển riêng biệt.

Dễ kiểm thử (Testability):

Việc kiểm thử từng lớp riêng biệt giúp xác định lỗi chính xác hơn và giảm thời gian kiểm thử hệ thống.

Phân công công việc hiệu quả trong nhóm:

Các thành viên trong nhóm có thể làm việc song song trên các lớp khác nhau mà không xung đột.

Tăng tính bảo mật và kiểm soát dữ liệu:

Lớp logic nghiệp vụ giúp kiểm soát chặt chẽ việc truy cập và xử lý dữ liệu, tránh truy xuất trực tiếp từ giao diện người dùng.

b. Khi thực hiện dự án có lượng người dùng lớn và phân tán, em chọn loại ứng dụng là Windows Application hay Web Application và giải thích lý do. Khi viết code, em chọn mô hình triển khai là Three layer hay MVC và giải thích lý do.

Lựa chọn loại ứng dụng: Web Application

Lý do:

Tính truy cập mọi lúc mọi nơi: Người dùng có thể truy cập từ bất kỳ thiết bị nào có trình duyệt và kết nối internet.

Không cần cài đặt: Không cần cài đặt ứng dụng trên từng máy người dùng, thuận tiện cho số lượng người dùng lớn và phân tán.

Cập nhật dễ dàng: Chỉ cần cập nhật ở phía server là toàn bộ người dùng đều được dùng phiên bản mới.

Khả năng mở rộng cao: Dễ dàng mở rộng cơ sở hạ tầng để phục vụ nhiều người dùng đồng thời.

Tương thích đa nền tảng: Hoạt động tốt trên cả máy tính, điện thoại, tablet,...

Ngược lại, Windows Application thường phù hợp với những ứng dụng dành riêng cho môi trường nội bộ, có ít người dùng, và yêu cầu hiệu năng cao.

Lựa chọn mô hình triển khai: MVC

Lý do chọn MVC (Model - View - Controller):

Tách biệt giữa giao diện, logic và dữ liệu: Giúp dễ bảo trì, dễ mở rộng và hỗ trợ làm việc nhóm hiệu quả.

Thân thiện với Web: MVC là mô hình phổ biến và tối ưu nhất cho các ứng dụng Web (như ASP.NET MVC, Spring MVC, Laravel...).

Hỗ trợ phát triển nhanh và linh hoạt: Có thể phát triển song song giữa Frontend (View), Backend (Controller + Model).

SEO-friendly: So với một số framework SPA khác, ứng dụng MVC truyền thống thường thân thiện với công cụ tìm kiếm hơn.

Tối ưu cho RESTful API: Dễ dàng tích hợp API phục vụ đa nền tảng nếu cần.

Trong khi đó, Three Layer Architecture cũng rất tốt, nhưng nó thường được áp dụng cùng với MVC để tăng thêm mức độ phân tầng trong logic nghiệp vụ và truy xuất dữ liệu.

**Câu 3:**

a. Xem danh sách tên khóa học của đơn vị là Tin theo mô hình Three Layer:

// DAL/Model

method {

//don't write codes to access database

public List<KhoaHoc> getDanhSachKhoaHoc(String donVi) {

// Code truy xuất CSDL ở đây

// Giả sử có hàm select \* from KhoaHoc where donvi = donVi

}

}

// BL/Controller

method {

//code here

public List<String> getTenKhoaHocByDonVi(String donVi) {

List<KhoaHoc> dsKhoaHoc = dal.getDanhSachKhoaHoc(donVi);

List<String> dsTenKhoaHoc = new ArrayList<>();

for(KhoaHoc kh : dsKhoaHoc) {

dsTenKhoaHoc.add(kh.getTen());

}

return dsTenKhoaHoc;

}

}

// UI/View

method () {

//code here

public void xemDanhSachKhoaHocTin() {

List<String> dsTenKhoaHoc = bl.getTenKhoaHocByDonVi("Tin");

System.out.println("Danh sách tên khóa học của đơn vị Tin:");

for(String ten : dsTenKhoaHoc) {

System.out.println("- " + ten);

}

}

}

b. Tìm kiếm khóa học có học phí >= 1000 theo mô hình Three Layer:

// DAL/Model

method {

//don't write codes to access database

public List<KhoaHoc> getAllKhoaHoc() {

// Code truy xuất tất cả khóa học từ CSDL

}

}

// BL/Controller

method {

//code here

public List<KhoaHoc> getKhoaHocByHocPhi(double hocPhiMin) {

List<KhoaHoc> dsKhoaHoc = dal.getAllKhoaHoc();

List<KhoaHoc> ketQua = new ArrayList<>();

for(KhoaHoc kh : dsKhoaHoc) {

if(kh.getHocPhi() >= hocPhiMin) {

ketQua.add(kh);

}

}

return ketQua;

}

}

// UI/View

method () {

//code here

public void timKhoaHocTheoHocPhi() {

List<KhoaHoc> dsKhoaHoc = bl.getKhoaHocByHocPhi(1000);

System.out.println("Danh sách khóa học có học phí >= 1000:");

for(KhoaHoc kh : dsKhoaHoc) {

System.out.println("ID: " + kh.getId() +

", Tên: " + kh.getTen() +

", Đơn vị: " + kh.getDonVi() +

", Học phí: " + kh.getHocPhi());

}

}

}

c. Xem danh sách tên khóa học mở trong năm 2022 theo mô hình MVC:

// Model

method {

//don't write codes to access database

public List<KhoaHoc> getKhoaHocByYear(int year) {

// Code để lấy khóa học từ CSDL theo năm

// SELECT \* FROM KHOAHOC WHERE YEAR(NGAYBATDAU) = year

}

}

// Controller

method {

//code here

public List<KhoaHoc> getKhoaHocTrongNam(int year) {

return model.getKhoaHocByYear(year);

}

public void showKhoaHocTrongNam(int year) {

List<KhoaHoc> dsKhoaHoc = getKhoaHocTrongNam(year);

view.displayKhoaHocList(dsKhoaHoc);

}

}

// View

method () {

//code here

public void displayKhoaHocList(List<KhoaHoc> dsKhoaHoc) {

System.out.println("Danh sách khóa học mở trong năm 2022:");

for(KhoaHoc kh : dsKhoaHoc) {

System.out.println("- " + kh.getTen() + " (Bắt đầu: " + kh.getNgayBatDau() + ")");

}

}

public void requestKhoaHoc2022() {

controller.showKhoaHocTrongNam(2022);

}

}