LẬP TRÌNH JAVA SPRING BOOT

BÀI 4: MyBatis Và Một số Design Pattern

MỤC TIÊU BÀI HỌC

- Kết thúc bài học này bạn có khả năng
  - Hiểu MyBatis (ORM Framework)
  - Ứng dụng MyBatis với Spring Boot
  - Hiểu Design Pattern
  - Một số Design Pattern trong Spring Boot
  - Ứng dụng Design Pattern trong dự án

- MyBatis là một persistence framework mã nguồn mở, đơn giản, gọn nhẹ và dễ sử dụng
- Nó tự động **ánh xạ** giữa **các trường** của **bảng** trong cơ sở dữ liệu SQL và các **trường** trong Java **POJOs** (Plain Old Java Objects) theo tên trường

Các lợi ích của MyBatis

- MyBatis đóng gói SQL dưới dạng các **stored procedure** sao cho logic nghiệp vụ có thể được **lưu giữ** trong cơ sở dữ liệu, và ứng dụng này có tính khả chuyển, dễ triển khai và thử nghiệm hơn
- Hỗ trợ **inline SQL**
- Hỗ trợ SQL động - MyBatis cung cấp các tính năng cho các truy vấn SQL **động** dựa trên các **tham số**
- Hỗ trợ ORM - MyBatis hỗ trợ nhiều tính năng tương tự như một **ORM tool**. Chẳng hạn như **lazy loading**, **join fetching**, **caching**, sinh ra **runtime code**, và **kế thừa**

- Thiết lập bằng maven
- Bạn chỉ cần thêm các dependency sau vào file pom.xml

```
<!--  Mybatis  -->
<dependency>
  <groupId>org.mybatis.spring.boot</groupId>
  <artifactId>mybatis-spring-boot-starter</artifactId>
  <version>2.2.2</version>
</dependency>
<!--  H2 -->
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
```

- Các câu lệnh SQL **ánh xạ** được định nghĩa trong file mapper **XML** hoặc **Annotations**
- Từ **MyBatis 3**, MyBatis cũng cung cấp các annotation để định nghĩa các câu lệnh SQL
- Ứng dụng các annotation để dùng MyBatis trong Spring Boot

- Để sử dụng MyBatis ở tầng tương tác với database, bạn cần sử dụng annotation @Mapper trên tên của interface repository

```
@Mapper
public interface PersonRepository {
}
```

- Bạn cần tạo các raw sql cho các method và sử dụng annotation để đánh dấu
- MyBatis cung cấp các annotation để định nghĩa các câu lênh như sau
  - **@Insert**: Định nghĩa một câu lệnh INSERT
  - **@Update** : Định nghĩa một câu lệnh UPDATE
  - **@Delete** : Định nghĩa một câu lệnh DELETE
  - **@Select** : Định nghĩa một câu lệnh truy vấn
  - **@Results** : dùng để mapper các dữ liệu database với thuộc tính của đối tượng JAVA
  - **@Options** : Định nghĩa các tuỳ chọn cho câu lệnh

Lưu ý: MyBatis **không có** các method trừu tượng sẵn có, các method CRUD như JPA.

Tạo Class Person

- Tạo class Person trong package entity

```
import java.util.Date;
import lombok.Data;

@Data
public class Person {

   private Long id;
   private String fullName;
   private Date dateOfBirth;
}
```

- **Lưu ý**:
  - Trong bài này có sử dụng kiến thức **Lombok**
  - H2 sẽ **không tự động** tạo table **Person**. Nên cần tạo table PERSON tại **H2 console**

- Tạo interface **PersonRepository** trong package repository

```
@Mapper
public interface PersonRepository {

   @Select("select count(1) from PERSON")
   int count();

   @Select("SELECT * FROM PERSON WHERE id = #{id}")
   @Results(value = { @Result(property = "id", column = "id"),
         @Result(property = "fullName", column = "fullName"),
         @Result(property = "dateOfBirth", column = "dateOfBirth") }
   )
   Person findById(long id);

   @Delete("DELETE FROM PERSON WHERE id = #{id}")
   int deleteById(long id);

   @Insert(" INSERT INTO PERSON (fullName, dateOfBirth) " +
         " VALUES (#{fullName}, #{dateOfBirth}) ")
   @Options(useGeneratedKeys = true, keyProperty = "id")
   int insert(Person person);

   @Update("Update PERSON set fullName=#{fullName} where id=#{id}")
   int update(Person person);
}
```

Tham số truyền vào

Cú pháp đặt tham số tương tứng

Sử Dụng PersonRepository Tại Tầng Service

- Tạo interface **PersonRepository** trong package repository
- Interface **PersonService**

```
public interface PersonService {
   int countPerson();
}
```

- Class implements PersonService

```
import com.likelion.threetier.repository.PersonRepository;
import com.likelion.threetier.service.PersonService;
import org.springframework.stereotype.Service;

import javax.annotation.Resource;

@Service
public class PersonServiceImpl implements PersonService {

   @Resource
   PersonRepository personRepository;

   @Override
   public int countPerson() {
      return personRepository.count();
   }
}
```

```
@Mapper
public interface PersonRepository {
    ...
}
```

Lưu ý với những repository chú thích là **@Mapper** thì Khi tiêm (inject) ta dùng **@Resource** để thay thế

- Design Patterns trong Spring Boot là một phần **thiết yếu** của phát triển phần mềm
- Design Pattern được sử dụng **thường xuyên** trong các ngôn ngữ **OOP**
- Nó cung cấp cho bạn các **giải pháp** để giải quyết các vấn đề chung, thường gặp một cách **tối ưu nhất**
- Cần nắm được **ý nghĩa** của mỗi Design Pattern để ứng dụng nó cho từng vấn đề phù hợp

Tại sao phải sử dụng Design Patterns?

- Design Pattern giúp bạn **tái** sử dụng mã lệnh và dễ dàng **mở rộng**
- Khi bạn gặp **khó khăn** trong lập trình, design patterns là kim chỉ nam giúp bạn **giải quyết vấn đề**
- Dùng design pattern **tránh** được các vấn đề **tiềm ẩn** gây ra những lỗi lớn, dễ dàng **nâng cấp**, **bảo trì**
- Có thể dễ dàng **trao đổi** với các thành viên trong **team** để cùng nhau **xây dựng** dự án

Khi nào nên sử dụng Design patterns?

- Khi muốn chương trình thực sự **đơn giản**
- Sử dụng các design pattern sẽ giúp **giảm** được **thời gian** và **công sức** suy nghĩ ra các cách giải quyết cho những **vấn đề** đã có lời giải

Phân loại Design Patterns

- Có khoãng **23** mẫu được định nghĩa trong lập trình hiện nay
- Được chia làm **3** nhóm
  - **Creational Pattern** gồm: Singleton, Factory Method, Abstract Factory... Nhóm này cung cấp giải pháp để tạo các object và che giấu được logic của việc tạo ra nó
  - **Structural Pattern** gồm: Proxy, Adapter, Bridge... Nhóm này dùng để thiết lập, định nghĩa quan hệ giữa các đối tượng
  - **Behavioral Pattern** gồm: Template Method, Observer, Strategy... Nhóm này dùng thực hiện các hành vi của đối tượng, sự giao tiếp giữa các object với nhau

Để học Design Patterns cần có gì?

- Nắm vững được **bốn** đặc tính của **OOP**: Kế thừa, Đa hình, Trừu tượng, Bao đóng
- Hai khái niệm **interface** và **abstract**

Học Design Pattern như thế nào?

- Cần tập trung chú ý vào 3 phần sau để học tốt một Design Pattern
  - Khi nào thì sử dụng design pattern đó, giải quyết được vấn đề gì? Và tại sao phải dùng design pattern này mà không phải cái khác
  - Học qua code ví dụ, ứng dụng vào thực tiễn
  - Học qua bằng sơ đồ UML

Các mẫu phổ biến nhất

- **Singleton Pattern**
- **Factory Method Pattern**
- **Proxy Pattern**
- **Template Pattern**

Một số lưu ý khi dùng Design Patterns

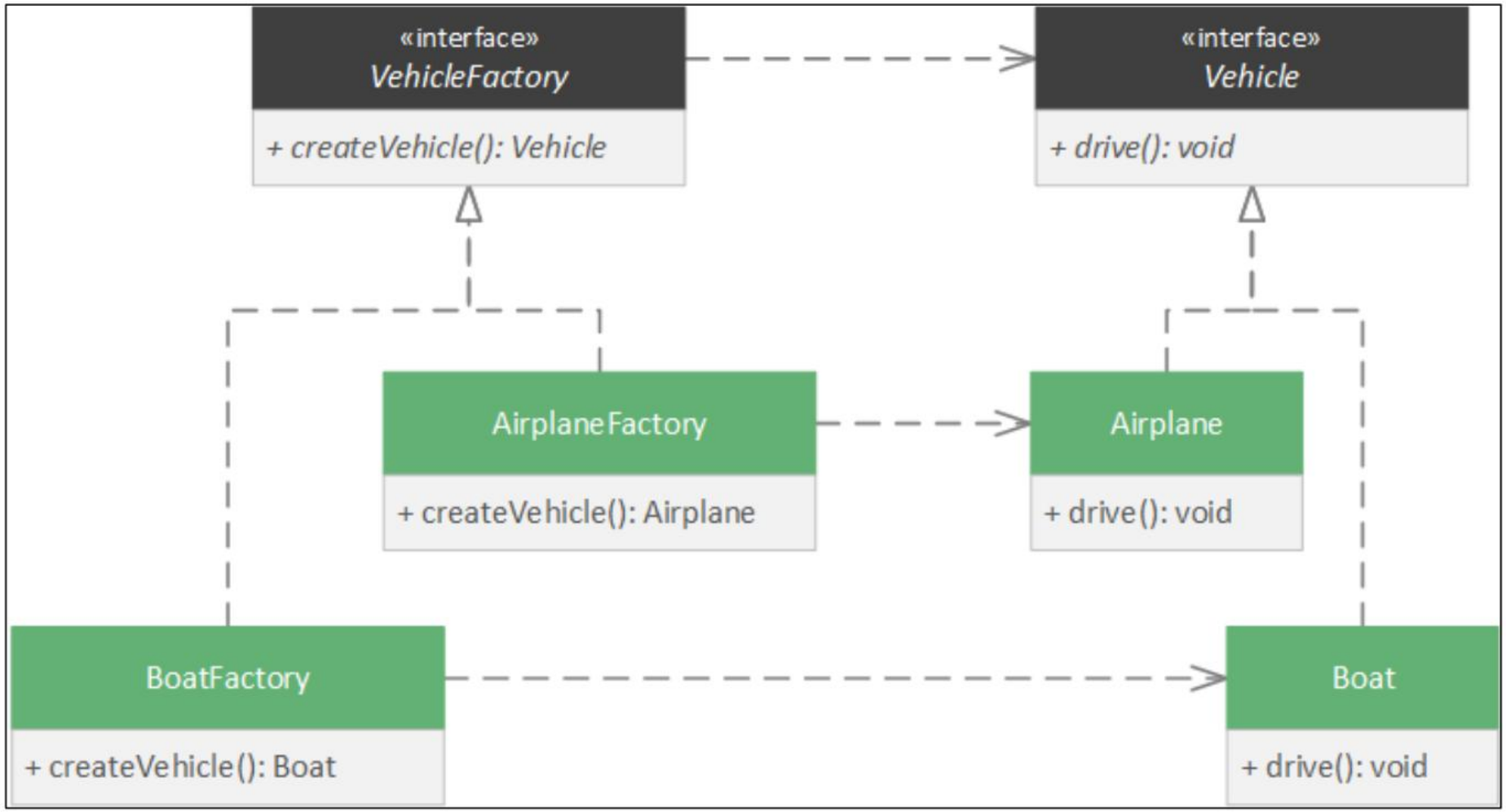
- Là một khuôn mẫu để giải quyết 1 vấn đề nào đó. Nó không phải là thiết kế cuối cùng
- Được tạo ra để giải quyết vấn đề, không phải để phức tạp hóa nó
- Nó phải giúp code được tối ưu hóa, dễ tái sử dụng, dễ hiểu, dễ nâng cấp sửa chữa

- **Singleton Pattern** có thể hiểu rằng một và chỉ một thể hiện (instance) của một lớp cụ thể sẽ được tồn tại trong một ứng dụng cụ thể trong **Spring container** (Spring IoC)
- Ví dụ
  - Tạo hai **controller** trong một **application context** và tiêm (inject) một bean cùng type vào mỗi controller
  - Thực hiện gọi hai controller để in ra địa chỉ của bean đó
    - *org.springframework.data.jpa.repository.support.SimpleJpaRepository@6639506f*
    - *org.springframework.data.jpa.repository.support.SimpleJpaRepository@6639506f*
- Các **ID** Repository trong hai controller là **giống nhau**
- **Spring** đã đưa **cùng** một bean vào cả hai controller

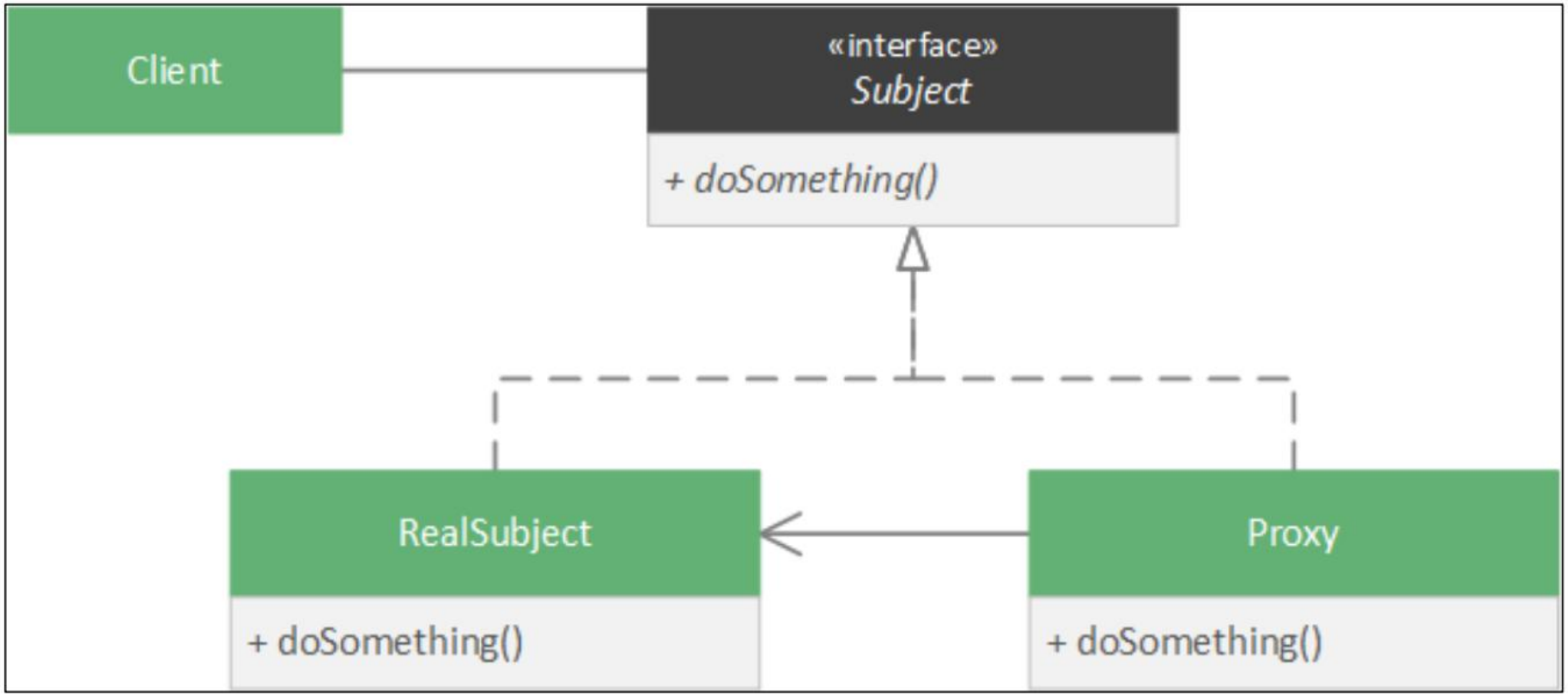
- Khi không áp dụng **Singleton** đối với bean
  - *com.likelion.rest.entity.Person@34aa6bdd*
  - *com.likelion.rest.entity.Person@24332033*
- Khi gọi đến hàm sẽ cho ra những kết quả ID **khác nhau**
- Vì mỗi lần gọi đối tượng sẽ được tạo lại vào lưu vào **ID** vùng nhớ mới
- Mặc định trong Spring Boot các bean là Singleton
- Để có thể thực hiện tạo những phiên bản **riêng biệt** cho mỗi lần gọi bean thì có thể sử thay đổi **phạm vi** của bean bằng cách sử dụng annotation
  - @Scope(value = "prototype" ,proxyMode = ScopedProxyMode.TARGET_CLASS)

- **Factory Method Pattern** yêu cầu một class Factory với một phương thức trừu tượng để tạo đối tượng mong muốn
- Ví dụ
  - Ứng dụng yêu cầu đối tượng phương tiện
  - Trong môi trường hàng hải, cần tạo ra tàu thuyền
  - Trong môi trường hàng không, cần tạo ra máy bay
- Để thực hiện điều này, cần tạo một **factory** implementation cho từng đối tượng mong muốn và trả về đối tượng mong muốn từ phương thức concrete factory

- Minh họa ví dụ trên bằng hình ảnh sau



- **Proxy Pattern** là một kỹ thuật cho phép một đối tượng – proxy – kiểm soát quyền truy cập vào đối tượng khác
- Hình ảnh ví dụ



- **Proxy Pattern** được Spring Boot ứng dụng như annotation **@Transactional**

```
@Service
public class BookManager {
@Autowired
private BookRepository repository;

@Transactional
public Book create(String author) {
    System.out.println(repository.getClass().getName()); return repository.create(author);
}
}
```

- Chú thích phương thức với annotation **@Transactional**
- Với annotation này, Spring tạo ra một **proxy** object **bao bọc BookRepository** object và cung cấp các đoạn mã cần thiết để bắt đầu một **transaction**
- Nếu **không** có proxy, Spring sẽ **không thể** kiểm soát quyền truy cập vào bean BookRepository và đảm bảo **tính nhất quán** giao dịch của nó

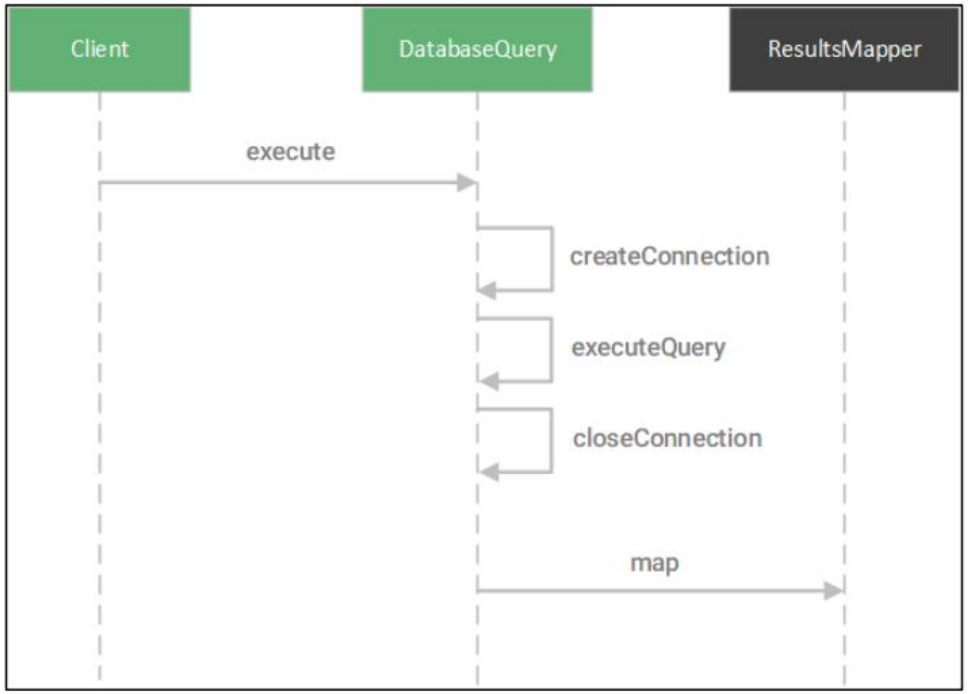
- **Template Method Pattern** là một kỹ thuật xác định **các bước cần thiết** cho một hành động, triển khai các bước **soạn sẵn** và để các bước có thể tùy chỉnh dưới dạng **trừu tượng**
- Các lớp con sau đó có thể **triển khai** lớp trừu tượng này và cung cấp một triển khai **cụ thể** cho các bước **còn thiếu**
- Ví dụ khi thực thi một câu truy vấn trên database, phải hoàn thành một loạt các bước sau

1. Establish a connection
2. Execute query
3. Perform cleanup
4. Close the connection

- Các bước này phù hợp cho một kịch bản lý tưởng cho **Template Method Pattern**

- Chúng ta có thể cung cấp bước còn thiếu bằng cách cung cấp callback method
- **Callback method** là một method cho phép chủ thể báo hiệu cho khách hàng rằng một số hành động mong muốn đã hoàn thành

- Trong một số trường hợp, đối tượng có thể sử dụng lệnh **callback** này để thực hiện các **hành động** – chẳng hạn như kết quả ánh xạ
- Thay vì có một phương thức **executeQuery**, chúng ta có thể **cung cấp** cho **phương thức** thực thi một chuỗi truy vấn và một phương thức gọi lại để xử lý kết quả



Tổng Kết Nội Dung

- MyBatis là một **persistence framework mã nguồn mở**, đơn giản, gọn nhẹ và dễ sử dụng
- Nó **tự động ánh xạ** giữa các trường của **bảng** trong cơ sở dữ liệu SQL và các trường trong **Java POJOs** (Plain Old Java Objects) theo tên trường
- Hỗ trợ SQL động - MyBatis **cung cấp** các tính năng cho các truy vấn SQL động dựa trên các **tham số**
- Hỗ trợ ORM - MyBatis hỗ trợ nhiều tính năng tương tự như một **ORM tool**. Chẳng hạn như **lazy loading**, **join fetching**, **caching**, sinh ra **runtime code**, và **kế thừa**
- Design Patterns trong Spring Boot là một phần **thiết yếu** của phát triển phần mềm
- Design Pattern được sử dụng **thường xuyên** trong các ngôn ngữ **OOP**
- Nó cung cấp cho bạn các **giải pháp** để giải quyết các vấn đề chung, thường gặp một cách **tối ưu nhất**
- Cần nắm được **ý nghĩa** của mỗi Design Pattern để ứng dụng nó cho từng vấn đề phù hợp
- Có **3** nhóm: **Creational Pattern**, **Structural Pattern**, **Behavioral Pattern**
- Cần nắm vững được bốn đặc tính của **OOP**: Kế thừa, Đa hình, Trừu tượng, Bao đóng và hai khái niệm **interface** và **abstract** để ứng dụng **Design Pattern**

Cảm Ơn Bạn Đã Chăm Chỉ!