**编程规范**

1. **Java编程规范**

1. 命名规范：

* 类名应使用驼峰式命名法，首字母大写，例如 MyClass。
* 方法名和变量名应使用驼峰式命名法，首字母小写，例如 myMethod。
* 常量名应使用大写字母和下划线，例如 MAX\_VALUE。
* 包名应全部小写，可以使用逆域名表示，例如 com.example.mypackage。

1. 缩进和空格：

* 使用四个空格进行缩进，而不是制表符。
* 在二元运算符之间、逗号后以及方法参数列表中的逗号前后添加空格。

1. 注释：

* 使用注释来解释代码的意图、功能和注意事项。
* 使用 Javadoc 风格的注释来描述类、方法和字段的作用，以及参数和返回值的含义。
* 在需要时添加注释，帮助他人理解代码的目的和实现。

1. 类和方法设计：

* 类应具有单一责任，尽量遵循单一职责原则。
* 方法应具有明确的目的，并遵循方法名的动词形式。
* 避免使用过长的方法和类。

1. 异常处理：

* 在适当的地方捕获和处理异常。
* 避免捕获所有异常，只捕获需要处理的特定异常。
* 使用 try-catch-finally 块来处理异常，确保资源的正确释放。

1. 格式化和排版：

* 使用一致的代码格式化风格。
* 使用空行和缩进来提高代码的可读性。
* 使用大括号 {} 来限定代码块。

1. 遵循标准库和框架的约定和惯例。

2. **项目架构规范**

1. 分层架构：将项目按照不同的层次进行分离，例如数据访问层、业务逻辑层和表示层。在我们的实际开发中，分为以下层：

* Controller层：
* 控制器层是应用程序的入口点，负责接收用户请求并协调处理过程。
* 它处理请求的路由、参数验证和数据转换，并调用相应的服务方法。
* 控制器层负责将服务的结果转化为适当的响应，通常是返回视图或数据给用户界面。
* Service层：
* 服务层是应用程序的核心业务逻辑层，封装了具体的业务功能。
* 它处理复杂的业务逻辑，协调不同的数据访问操作，并提供高级别的方法供其他层调用。
* 服务层可以调用DAO层来执行数据访问操作，并对数据进行处理和转换。
* DAO层
* 用于封装对数据存储（如数据库）的访问和操作
* 它提供了一组接口或方法，用于执行CRUD操作（创建、读取、更新、删除）和其他与数据存储相关的操作。
* 将业务逻辑与数据访问细节相分离，可以调用Repository层，使得应用程序的其他部分可以独立于底层数据存储进行开发和测试
* Repository层：
* Repository层是用于访问和操作数据存储的层级。
* 它封装了与数据存储的交互
* Repository层负责处理数据的持久化和检索，并与底层的数据访问技术（如ORM框架）进行交互。
* Entity层：
* 实体层是用于表示应用程序的领域模型和业务实体的层级。
* 它定义了与业务相关的数据结构和实体对象，通常映射到数据库表。
* 实体层描述了实体的属性和行为，并与数据访问层和服务层之间进行数据传递。

这些层之间的协作和分离有助于实现代码的可扩展性、可维护性和可测试性。Controller层接收和处理用户请求，调用Service层进行业务处理，Service层通过调用Repository层实现对数据的访问和操作，同时Entity层作为数据的抽象模型存在。

遵循这种分层架构和相应的编程规范可以提高代码的组织性、可读性和可维护性，使团队能够更好地合作开发和维护复杂的应用程序。

1. 模块化设计：将项目划分为模块或组件，每个模块具有明确定义的功能和责任。模块之间应该是松散耦合的，通过良好定义的接口进行通信。
2. 设计模式：使用适当的设计模式来解决常见的软件设计问题，例如单例模式、工厂模式、观察者模式等。设计模式提供了经过验证的解决方案，并促进代码的可重用性和可扩展性。
3. SOLID 原则：遵循 SOLID 原则，包括单一职责原则、开闭原则、里氏替换原则、接口隔离原则和依赖倒置原则。这些原则有助于设计松耦合、可维护和可扩展的代码。
4. 依赖注入：使用依赖注入来管理组件之间的依赖关系，而不是硬编码依赖。这可以提高代码的可测试性和灵活性，并减少模块之间的耦合。
5. 清晰的命名和约定：使用有意义的名称来命名类、方法、变量和文件。遵循一致的命名约定，以增加代码的可读性。
6. 文档和注释：为项目提供清晰、详尽的文档和注释。描述项目的整体结构、模块之间的关系、重要的设计决策和代码的使用方法。这有助于团队成员理解项目，并提供指导和参考。
7. 测试驱动开发：采用测试驱动开发（TDD）的方法，先编写测试，然后再编写代码以满足测试。这有助于确保代码的质量和正确性，并促进良好的项目结构和可测试性。