实验内容：

1. 参考教材6.2，结合项目的进程和开发历程，从设计原则的几个方面，组员对负责设计的模块进行评估，思考存在的问题和解决方案。

1. 模块化

评估：

代码结构： 代码中不同功能的逻辑分散在多个函数中，如数据库操作、界面操作、用户登录等。

问题：

功能逻辑没有很好地模块化，导致代码难以维护和测试。

解决方案：

将不同功能的逻辑封装到不同的模块中。

2. 接口

评估：

数据库操作： 数据库操作的逻辑直接依赖于 sqlite3 模块。

界面操作： 界面操作的逻辑直接依赖于 tkinter 模块。

问题：

直接依赖于具体的实现，导致代码难以测试和替换。

解决方案：

定义数据库操作的接口，如 DatabaseInterface，并在 DatabaseModule 中实现该接口。

定义界面操作的接口，如 UserInterfaceInterface，并在 UserInterfaceModule 中实现该接口。

3. 信息隐藏

评估：

数据库操作： 数据库操作的逻辑暴露在多个函数中，如 doSql、check\_lock 等。

问题：

数据库操作的逻辑没有很好地隐藏，导致代码难以维护和测试。

解决方案：

将数据库操作的逻辑封装到 DatabaseModule 中，并提供对外的接口，隐藏具体的实现细节。

4. 增量式开发

评估：

代码结构： 代码中不同功能的逻辑分散在多个函数中，没有很好地分阶段开发。

问题：

功能逻辑没有很好地分阶段开发，导致代码难以维护和测试。

解决方案：

将不同功能的逻辑分阶段开发，如先开发数据库操作模块，再开发界面操作模块，最后开发用户登录模块。

5. 抽象

评估：

用户类型处理： 代码中对不同用户类型的处理逻辑分散在 login 函数中，没有使用抽象。

问题：

没有使用抽象，导致代码难以扩展和维护。

解决方案：

定义一个基类 User，并为每种用户类型定义一个子类，如 Student、Teacher、Admin。在 login 函数中使用基类类型来处理用户。

6. 通用性

评估：

数据库操作： 数据库操作的逻辑直接依赖于 sqlite3 模块。

问题：

直接依赖于具体的实现，导致代码难以替换和扩展。

解决方案：

定义一个数据库操作的接口，在其他模块中依赖于该接口，而不是具体的实现。

2. 阅读下面DI资料（或查阅其它相关资料），学习依赖注入技术。

依赖注入（Dependency Injection）是一种设计模式，它通过将对象的依赖关系交给外部容器来管理，从而实现对象之间的松耦合。依赖注入的主要目的是将复杂的依赖关系管理和创建对象的任务交给容器来实现，从而简化代码的编写和维护。

在依赖注入中，有三个角色：依赖关系的提供者（Provider）、依赖关系注入的目标（Consumer）和容器（Container）。依赖关系提供者是负责创建和提供依赖关系的对象，依赖关系注入的目标是需要依赖的对象，容器则是负责管理依赖关系的中介。

依赖注入主要有三种方式：构造函数注入、属性注入和方法注入。构造函数注入是通过构造函数将依赖关系传递给对象，属性注入是通过属性的setter方法注入依赖关系，方法注入是通过调用方法将依赖关系传递给对象。

依赖注入可以改善代码的可测试性、可维护性和可重用性，它可以使得代码变得更加灵活和可扩展。