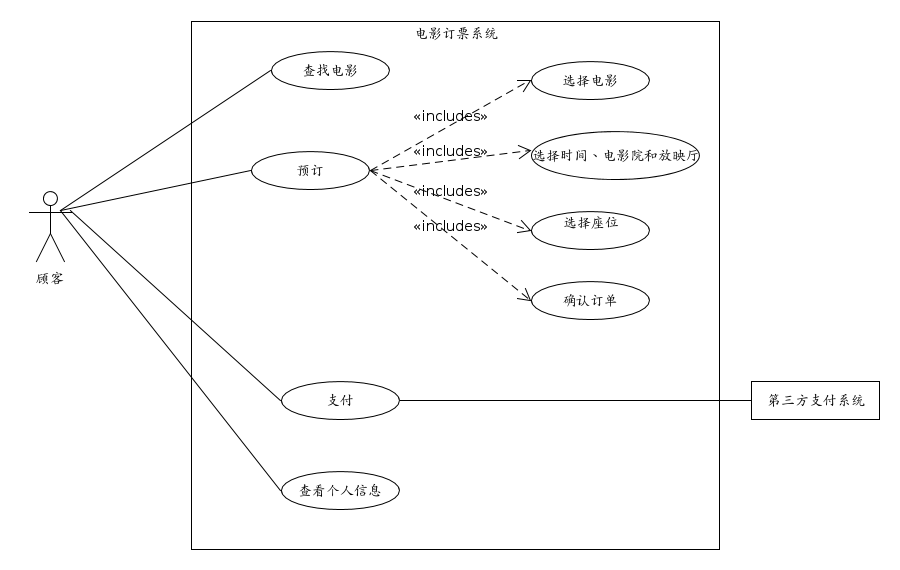
**软件设计文档**

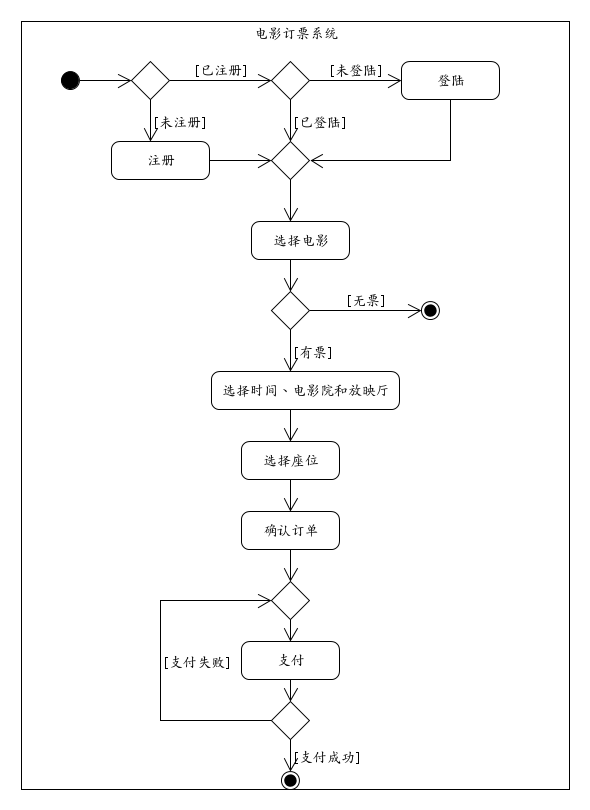
Inception & Elaboration

一、模型图及其说明

1. Usecase

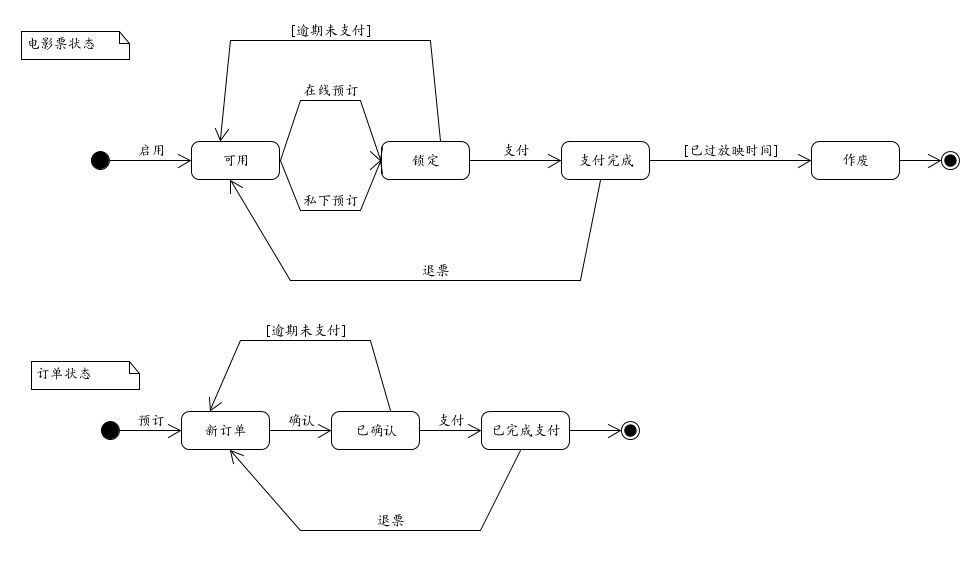
 其中共有四个主用例（支付部分未实现），最重要的用例为“预定”，其中包括了订电影票的主要步骤，即选择电影，选择相应的时间电影院和放映厅，选择座位以及最终确认订单。这也是我们代码所主要实现的部分。

2. Activity

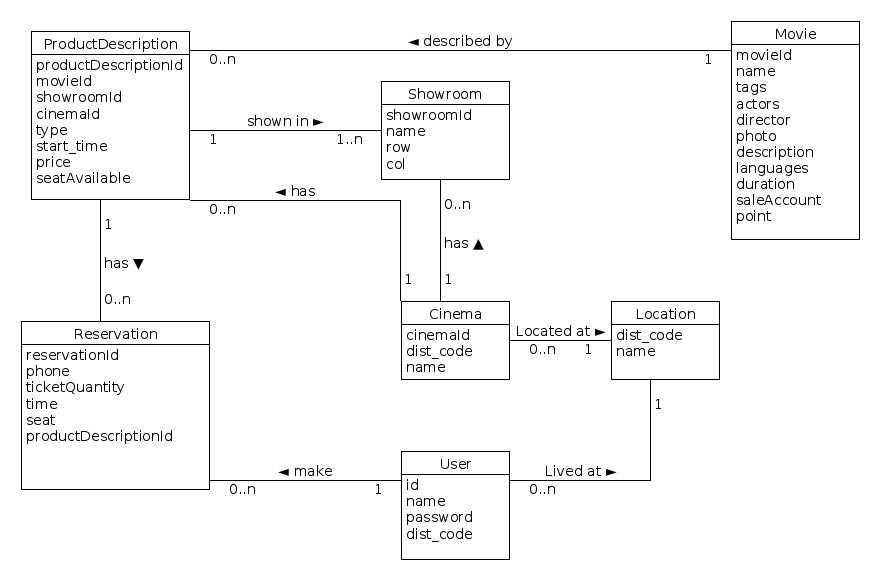


活动图根据“预定”主用例，描述了电影订票的顺序（流程），包括了其中的分支逻辑等。绘制出活动图可以更好地帮助我们理解整个订票的流程。

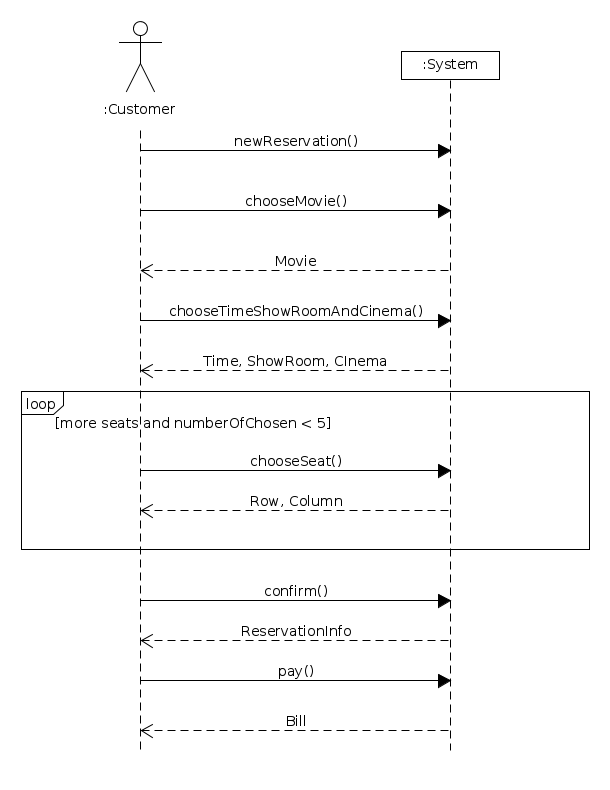
3. State

 状态图主要说明了在主用例“预定”中，电影票和订单状态的变化（其中与支付相关的功能尚未实现，例如逾期未支付等等）。绘制出状态图可以更好地帮助我们理解整个订票流程中电影票和订单所处的状态以及它们之间的变迁。

4. Domain



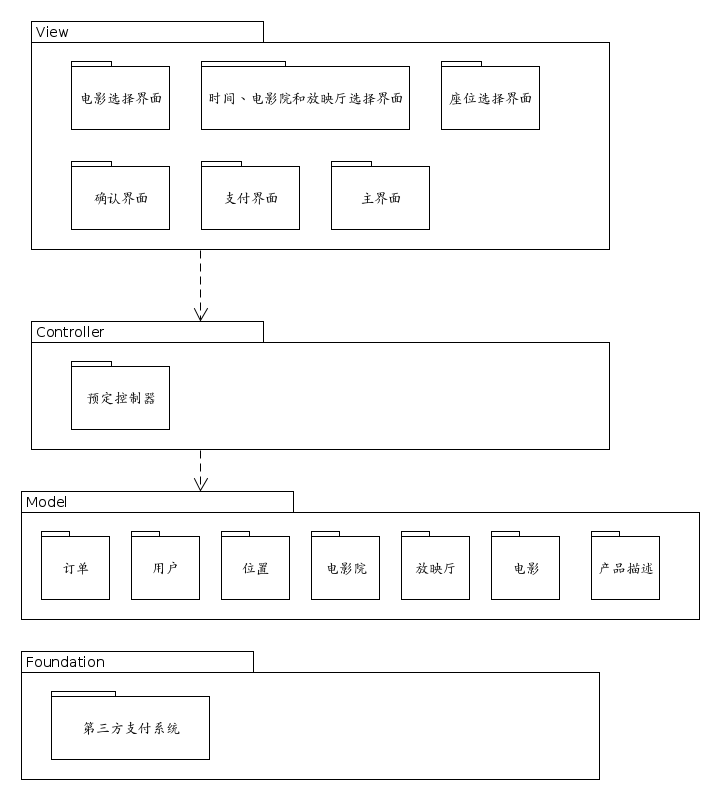
领域模型描述的是各个类初步的属性值和相应的联系。显然这是不准确的，但是通过领域模型可以帮助我们理清各个类的成员变量，也可以加速我们的开发过程。例如，通过绘制领域模型，我们得出了ProductDescription类，如果直接进行开发，很可能会忽略这个类，而将其与其他类（Reservation或Movie类）杂糅起来，这样不仅会造成在数据库中的数据重复，同时会导致业务逻辑变得相当复杂，达不到高内聚、低耦合的软件设计要求。因此，通过领域模型可以很好地帮助我们少走弯路，快速进行后续的DB设计和代码开发。



5. SSD

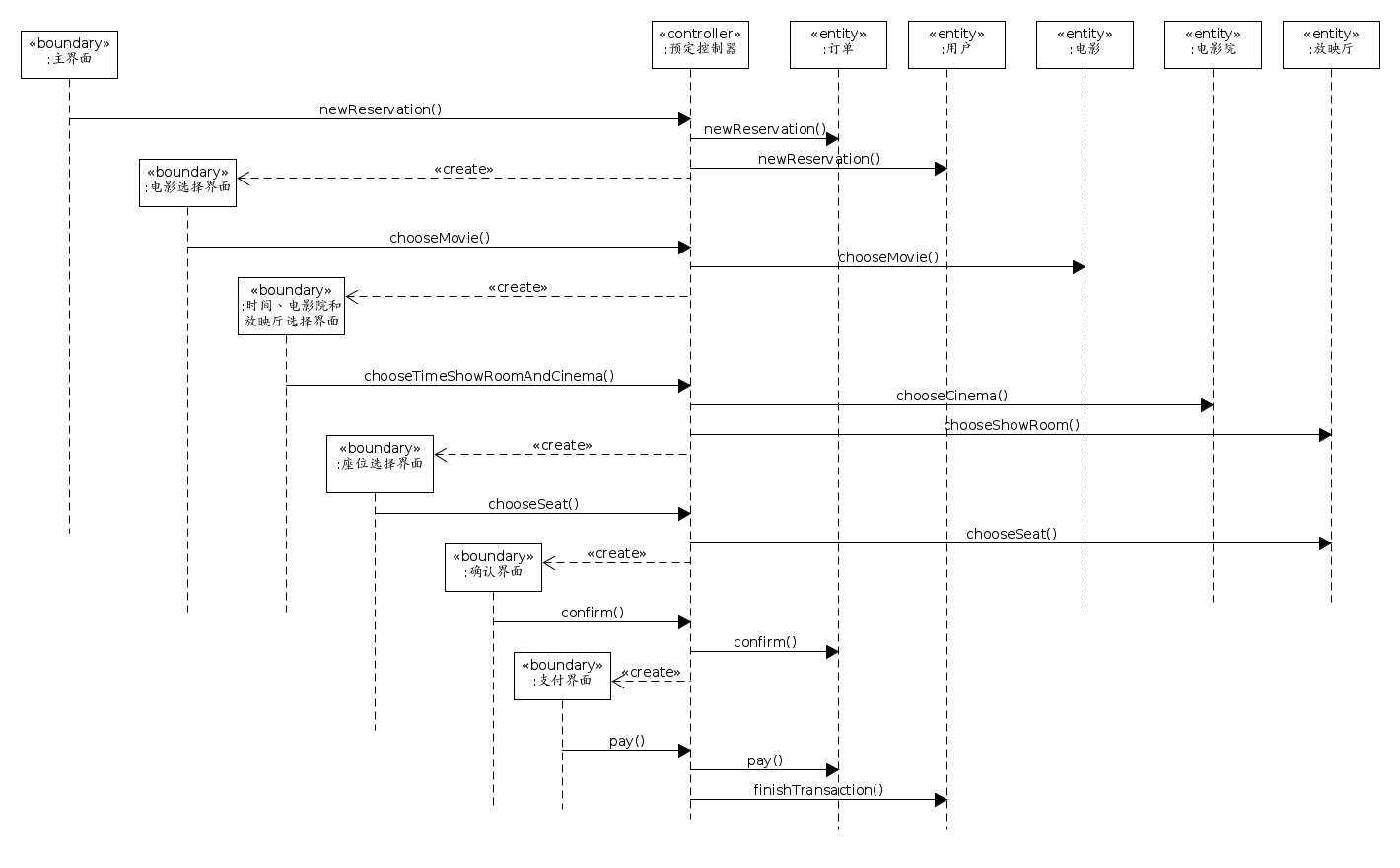
系统顺序图将整个系统当作一个黑盒来看待，描述了用户和系统之间大致的交互。例如，要订电影票，首先需要新创建一个订单，之后不断选择所要看的电影，时间，电影院，放映厅等信息。最后确认订单并完成支付。通过SSD，我们可以大致上确定系统的操作，这样，我们可以进一步细化任务，将任务分派给不同的人，提高开发效率。

6. Package



包图按照MVC的架构，将每一个类分开成界面，控制器和模型。这样可以将UI与业务逻辑操作相分离，提高程序的可维护性。通过包图，我们可以清晰地知道自己所要开发的模块的作用，不至于多编码一些不需要的部分，可以节约时间，同时便于将大的任务各个击破分配给不同的人进行编码。

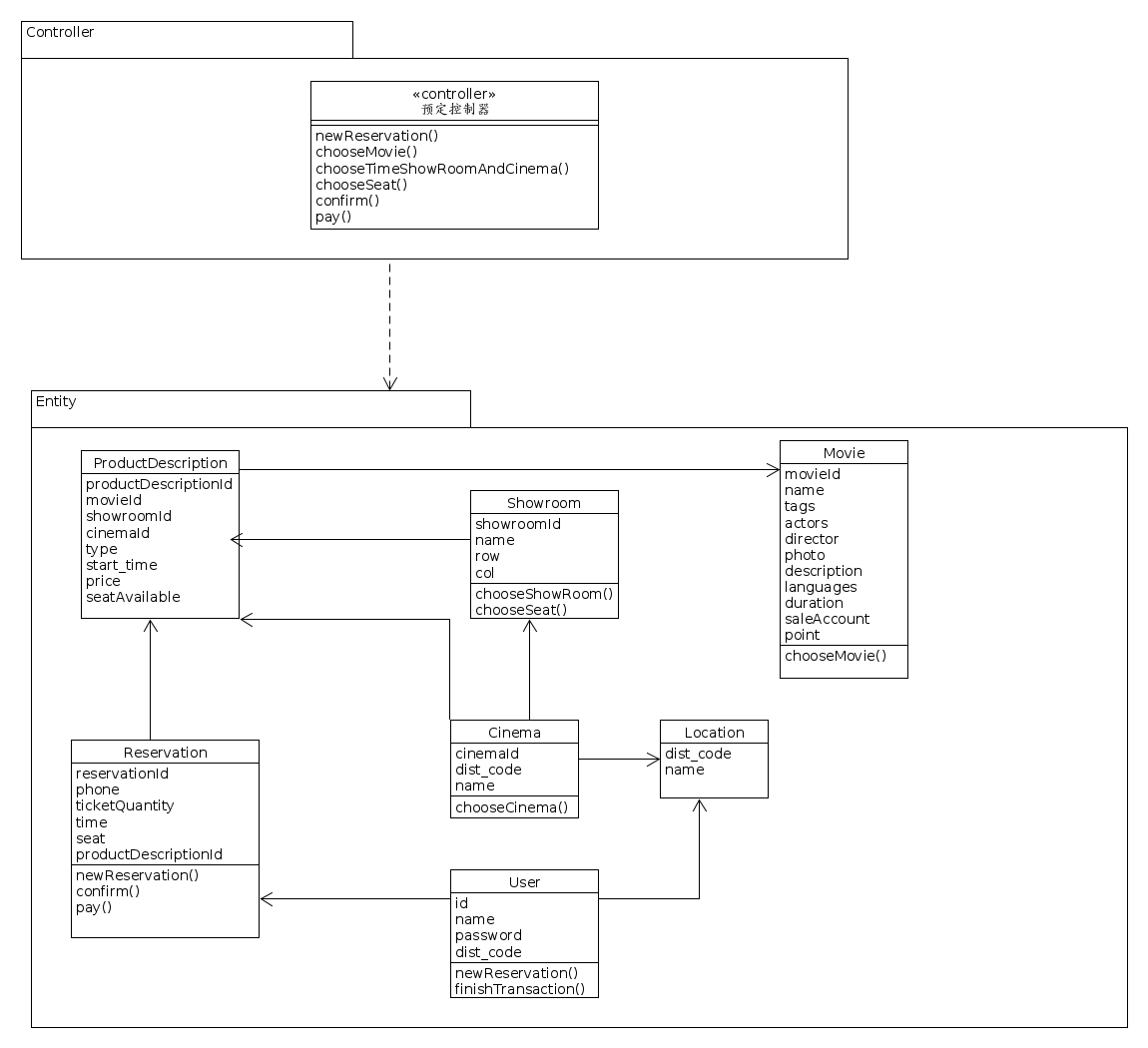
7. Sequence



顺序图属于比较复杂的一个图，他通过界面与控制器与类的交互具体描述了每个类的方法，为我们选取类的方法提供了便利。其中，在控制器左边的都是<<Boundary>>，也就是我们熟知的UI界面，例如主界面，电影选择界面等等。一开始我们只有主界面，而通过调用newReservation()方法，控制器会创建<<create>>出一个新的界面，即电影选择界面。当我们完成电影的选择后，即调用chooseMovie()方法，又会创建出 时间、电影院和放映厅选择界面。不断往复，会有座位选择界面，确认界面和支付界面显示出来，同时不断调用相应的函数。而在调用相对应的控制器函数时，控制器会调用相应类（Entity）的函数，对类的值进行修改或者对数据库中的值进行修改（例如，对订单类的newReservation()方法，confrim()方法和pay()方法）。不过这些都属于实现中的细节，在这里我们不需要关注，只需要用一个函数代替即可。因

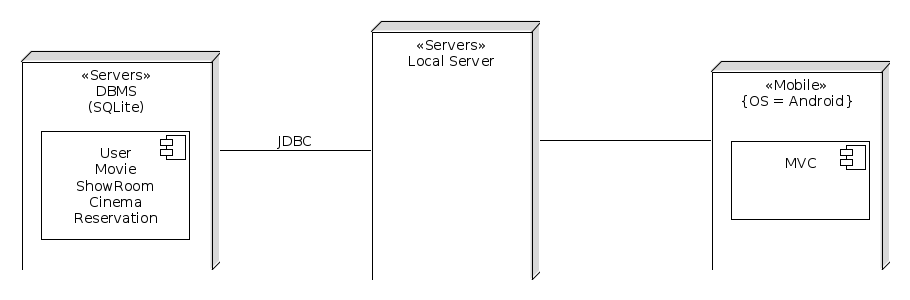
此，通过顺序图，我们了解地不是具体的实现细节，而是各个类之间交互的方法，便于我们对接口的编码。当我们通过顺序图帮助我们找到相应的接口时，我们就可以委派剩下的成员完成相应的实现，而不必浪费时间等待所有的架构确定后再进行编码，因此我们的编码效率也有所提高。不过还有一点需要注意的是，这些方法只是大致上的方法。因此更加详细的接口信息还是需要在细化阶段不断实践，找出最适合的接口。

8. Class



这是BCE类图，不过其中省略了Boundary部分。其实这张图就是对之前几个图（领域模型、顺序图等）的综合。根据领域模型得出Entity的成员变量，根据顺序图得出各个类的方法。这个图的意义在于，指导我们设计基本的类，这也大概是我们类的模型基础。

9. Deployment



部署图说明了我们程序的部署方案。我们使用的是安卓本身提供的SQLite数据库，数据库中保存了相应的信息。同时，我们使用的是Local Server，而不是Web Server（因为使用Web Server需要备案，手续相对繁杂）。同时最右的方框中说明了我们的APP是运行在Android系统上，使用了MVC架构等信息。

10. DB Tables

Cinema

|  |  |  |
| --- | --- | --- |
| CinemaId(PK) | DistCode | Name |

User

|  |  |  |  |
| --- | --- | --- | --- |
| UserId(PK) | Name | Password | DistCode |

Location

|  |  |
| --- | --- |
| DistCode(PK) | AddressName |

Movie

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| MovieId(PK) | | Name | Tag | | Actors | Director | Photo | Description(memo) | |
| Languages | Duration(Integer) | | | Sale Account(Long) | | | Point(Single) | |

Reservation

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ReservationId(PK) | ReserveTime | Phone | Seat | TotalPrice(Single) | TicketQuantity(Integer) |

ProductDescription

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ProductDescriptionId(PK) | Type | StartTime | Price(Double) | SeatAvailiable |

ScreeningRoom

|  |  |  |  |
| --- | --- | --- | --- |
| ScreeningRoomId(PK) | Name | Row(Integer) | Col(Integer) |

注：未标注的均为text类型，以上Table都符合3NF，没有冗余信息。

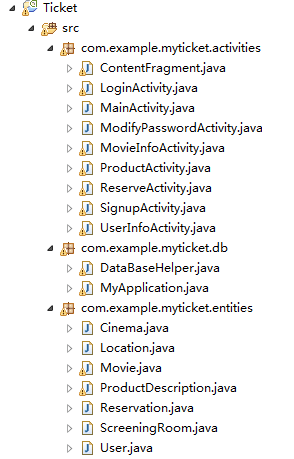
Development

1. 开发总体架构考量

上一个学期学习过Android课程，在期末的大作业中我们基本上把对数据的操作和对界面的更改杂糅在一起。一方面，这为我们的分工造成了很大的麻烦，另一方面，对于debug的难度也很大，因为所有代码都混在一个Activity中，一旦出现异常也很难确认到底是哪一部分出现了问题。

这一次软件设计实验，我们将各种操作分离，因此任务的分派也显得相对简单。我们小组一共有三位同学负责编码，分别负责Activity、数据库和类的编写。这样的话，我们就实现了**UI与业务逻辑的分离**，并且有专门的数据库Helper类来完成对数据库的CRUD操作。在开发时，我们只要事先约定设计好相应的函数接口，就能够各司其职，实现自己部分的代码，之后再整合在一起，我们的开发效率得到了提高。同时每个人在编写自己部分的时候也需要负责相应的测试任务，保证代码的正确性，这样可以降低BUG的出现率。

**程序结构图如下：**

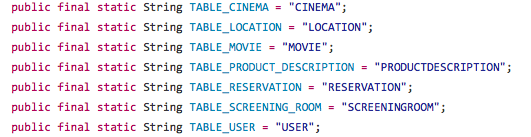
****

更加好的实现方法应该是在界面和数据库操作之间再加入一层抽象，而不是直接在Activity中调用DBHelper中的方法。但是考虑到这个项目并不是很大，测试相对比较容易，因此采用了两层的架构模型，提升开发效率。

2. DataBaseHelper

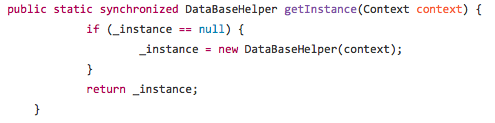
1) 首先定义了许多final static string作为常量，这样就避免了在开发过程中对数据库表名，键值名修改时需要进行多次修改的麻烦。提高了程序的开发效率、可维护性，同时降低了出错的概率。

如图：



2) 单例模式：

如图：



3) 增删改的操作都是调用Android提供的API而不是使用原生的SQL命令，这里有一层封装，当出现问题时会抛出异常，因此便于DEBUG。

如图:

