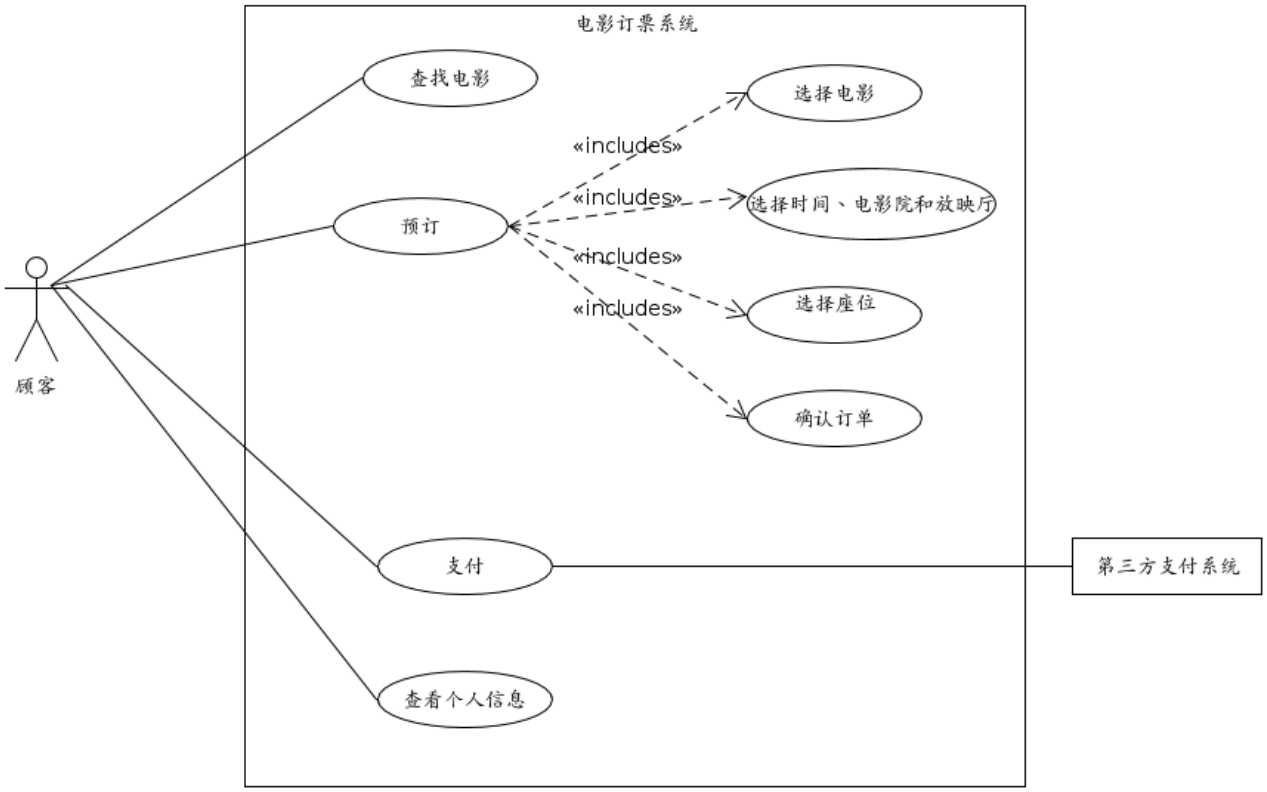
软件设计文档

Inception & Elaboration

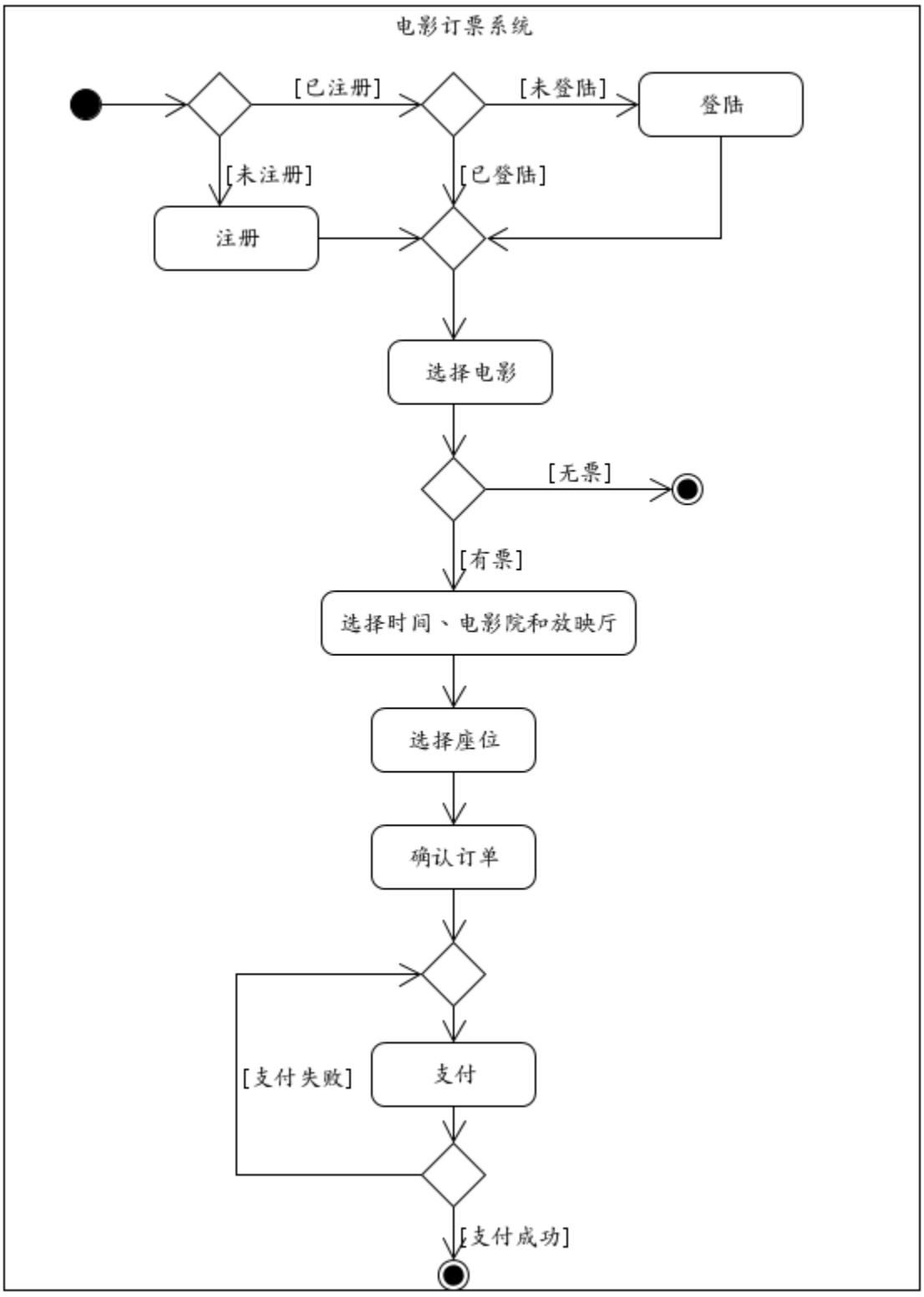
一、模型图及其说明

1. Usecase



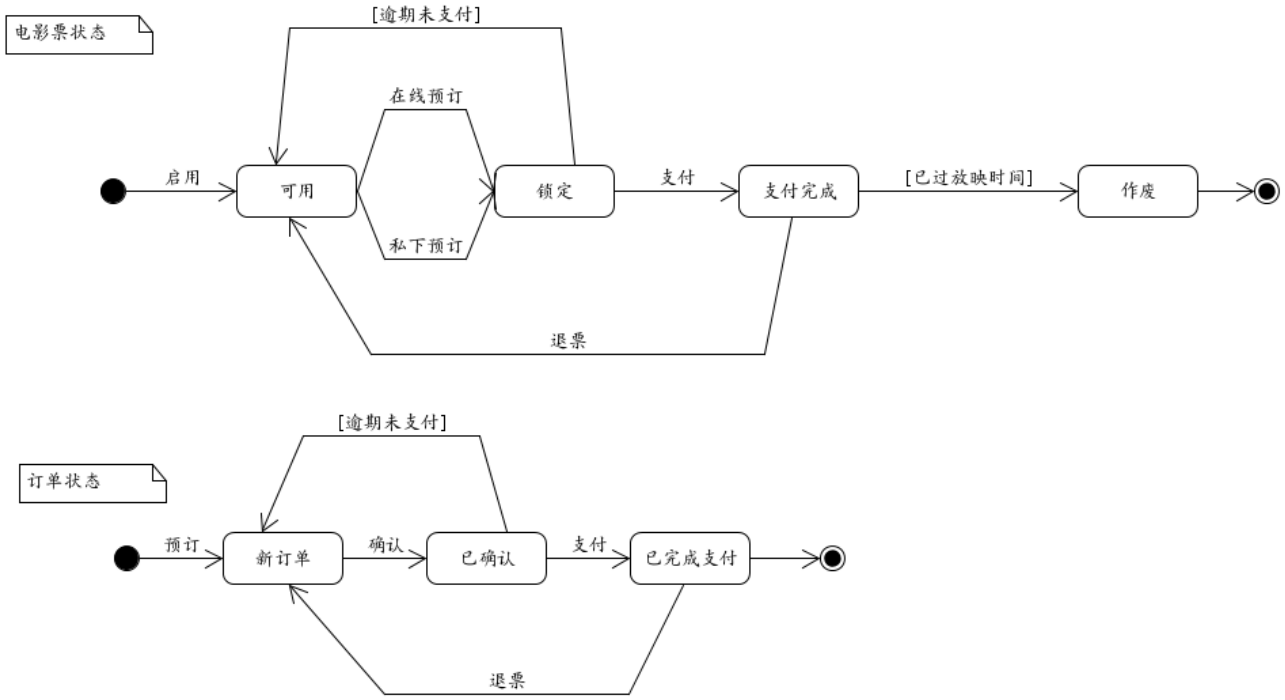
其中共有四个主用例（支付部分未实现），最重要的用例为“预定”，其中包括了订电影票的主要步骤，即选择电影，选择相应的时间电影院和放映厅，选择座位以及最终确认订单。这也是我们代码所主要实现的部分。

2. Activity



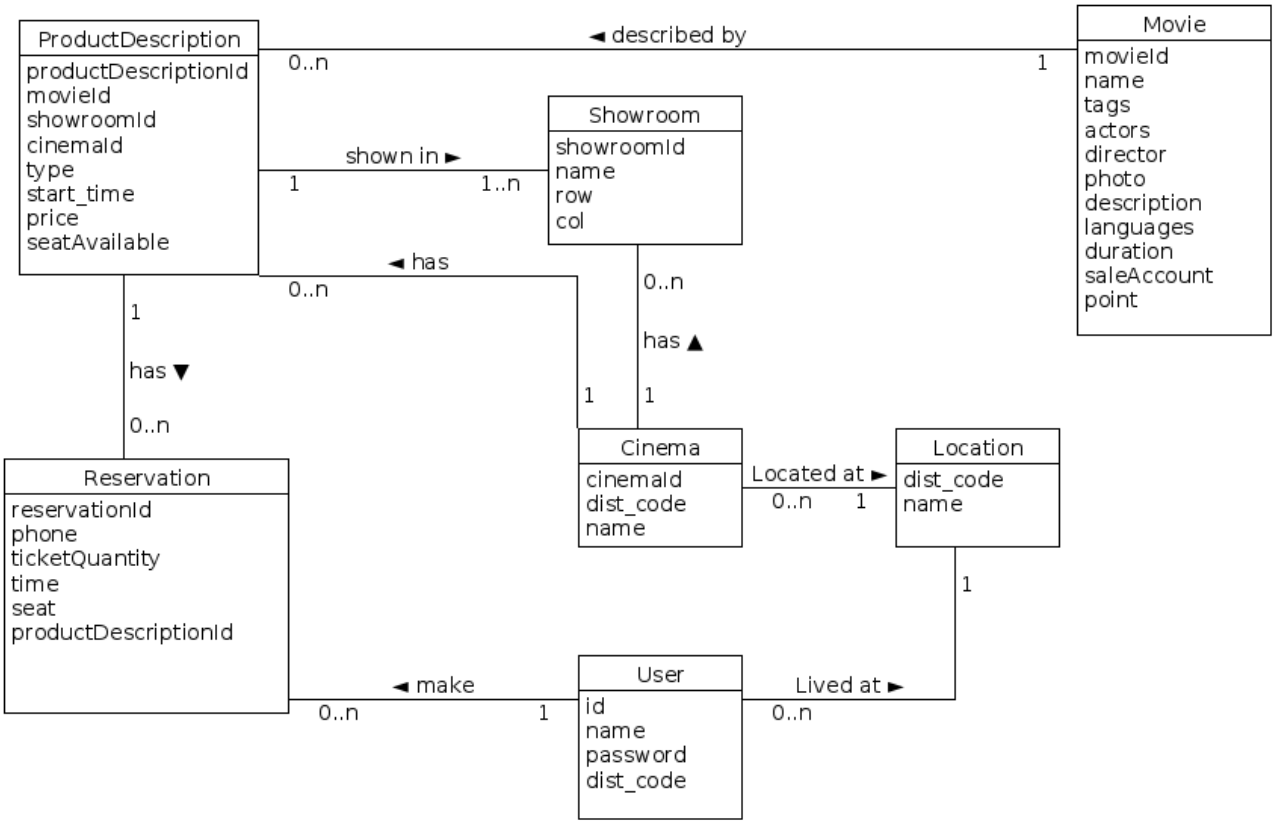
活动图根据“预定”主用例，描述了电影订票的顺序（流程），包括了其中的分支逻辑等。绘制出活动图可以更好地帮助我们理解整个订票的流程。

3. State



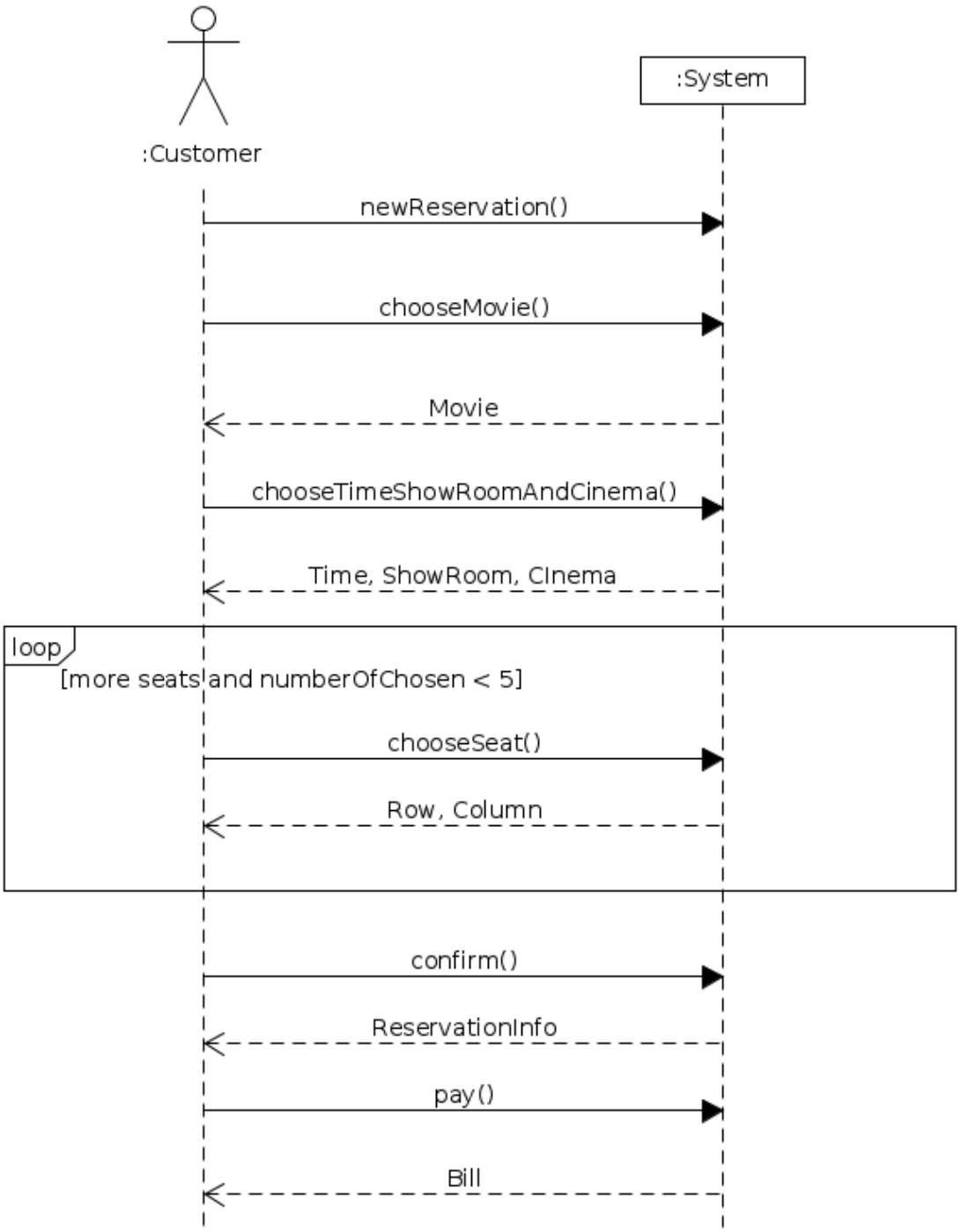
状态图主要说明了在主用例“预定”中，电影票和订单状态的变化（其中与支付相关的功能尚未实现，例如逾期未支付等等）。绘制出状态图可以更好地帮助我们理解整个订票流程中电影票和订单所处的状态以及它们之间的变迁。

4. Domain



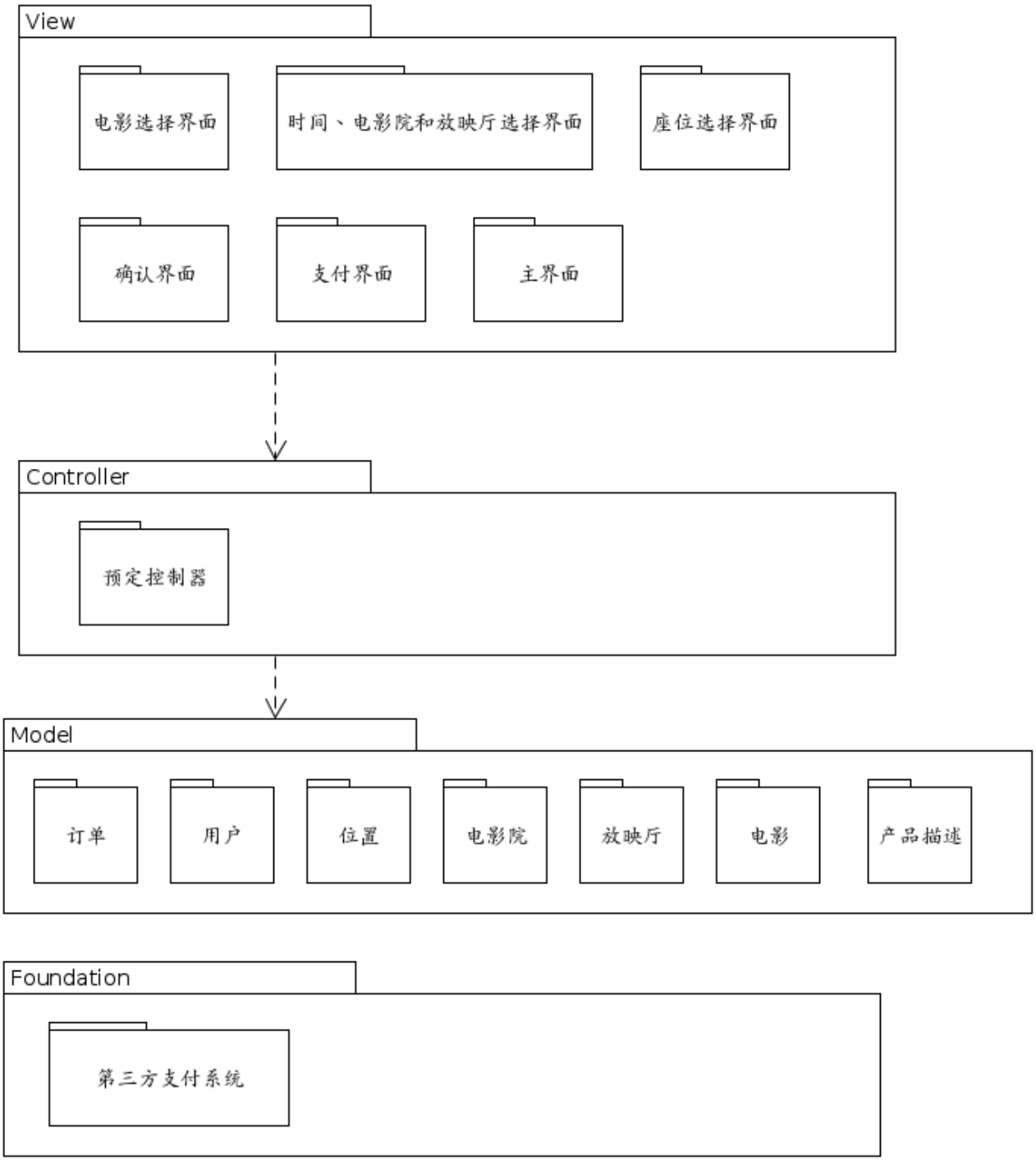
领域模型描述的是各个类初步的属性值和相应的联系。显然这是不准确的，但是通过领域模型可以帮助我们理清各个类的成员变量，也可以加速我们的开发过程。例如，通过绘制领域模型，我们得出了 ProductDescription 类，如果直接进行开发，很可能会忽略这个类，而将其与其他类（Reservation 或 Movie 类）杂糅起来，这样不仅会造成在数据库中的数据重复，同时会导致业务逻辑变得相当复杂，达不到高内聚、低耦合的软件设计要求。因此，通过领域模型可以很好地帮助我们少走弯路，快速进行后续的 DB 设计和代码开发。

5. SSD



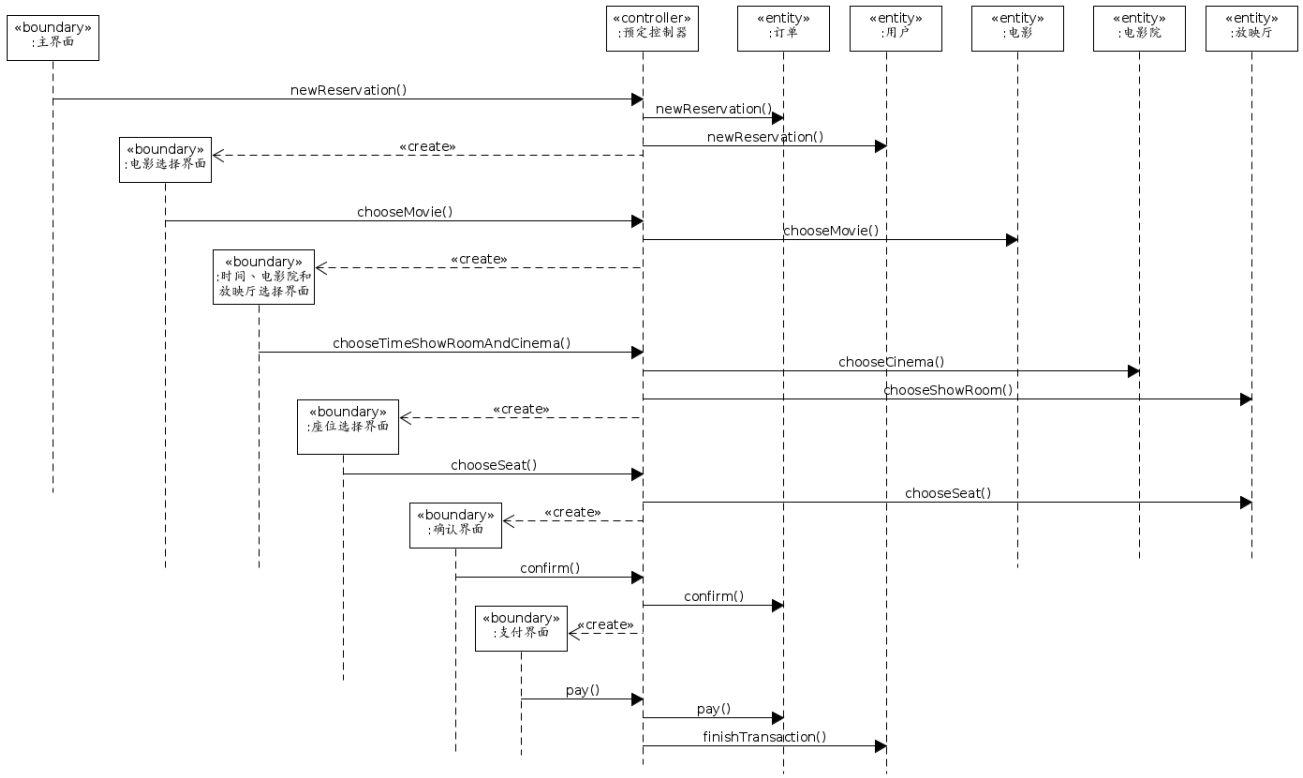
系统顺序图将整个系统当作一个黑盒来看待，描述了用户和系统之间大致的交互。例如，要订电影票，首先需要新创建一个订单，之后不断选择所要看的电影，时间，电影院，放映厅等信息。最后确认订单并完成支付。通过 SSD，我们可以大致上确定系统的操作，这样，我们可以进一步细化任务，将任务分派给不同的人，提高开发效率。

6. Package



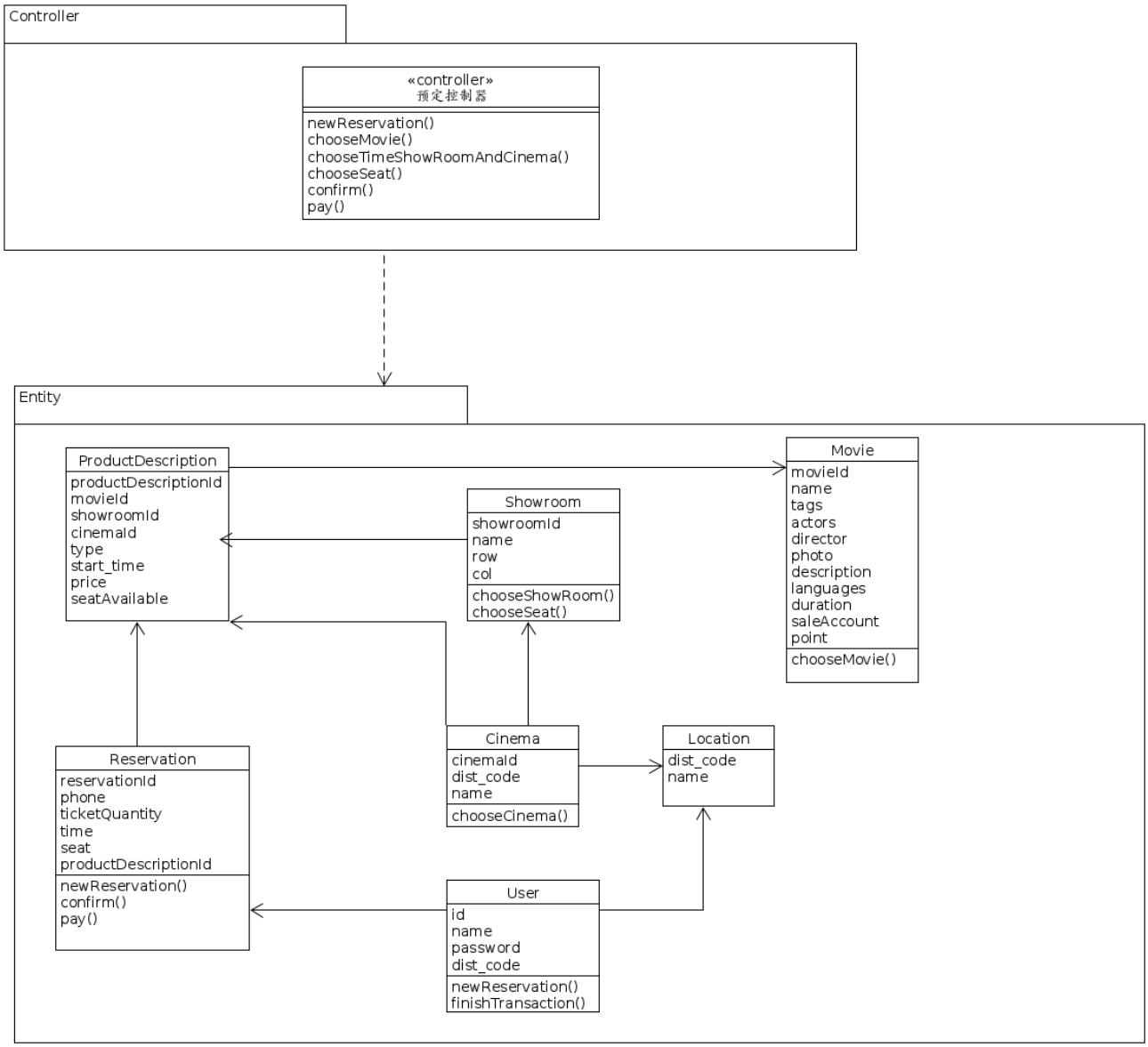
包图按照 MVC 的架构，将每一个类分开成界面，控制器和模型。这样可以将 UI 与业务逻辑操作相分离，提高程序的可维护性。通过包图，我们可以清晰地知道自己所要开发的模块的作用，不至于多编码一些不需要的部分，可以节约时间，同时便于将大的任务各个击破分配给不同的人进行编码。

7. Sequence



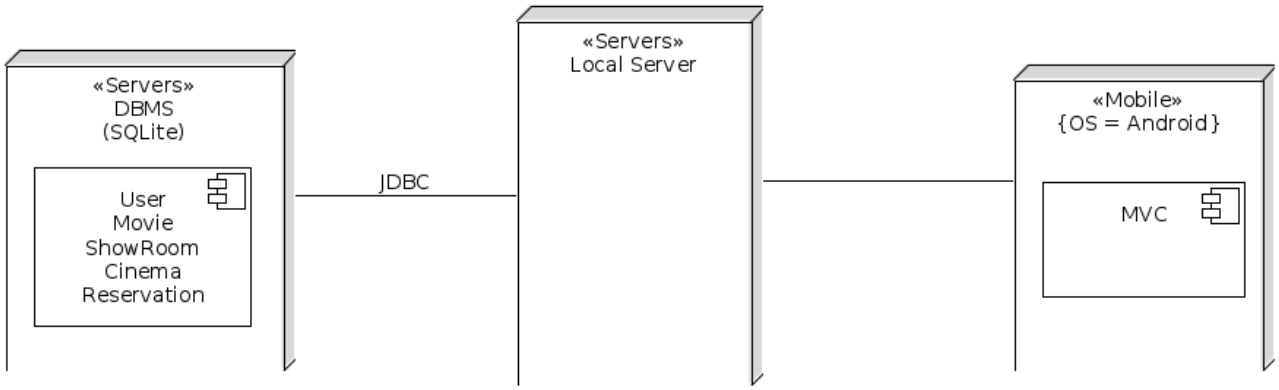
顺序图属于比较复杂的一个图，他通过界面与控制器与类的交互具体描述了每个类的方法，为我们选取类的方法提供了便利。其中，在控制器左边的都是<<Boundary>>，也就是我们熟知的 UI 界面，例如主界面，电影选择界面等等。一开始我们只有主界面，而通过调用 newReservation()方法，控制器会创建<<create>>出一个新的界面，即电影选择界面。当我们完成电影的选择后，即调用 chooseMovie()方法，又会创建出 时间、电影院和放映厅选择界面。不断往复，会有座位选择界面，确认界面和支付界面显示出来，同时不断调用相应的函数。而在调用相对应的控制器函数时，控制器会调用相应类（Entity）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

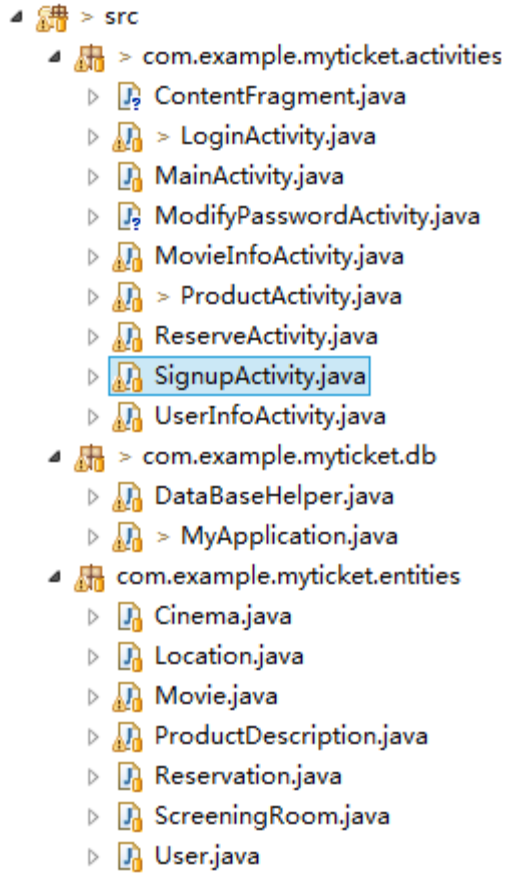
一、技术选型理由

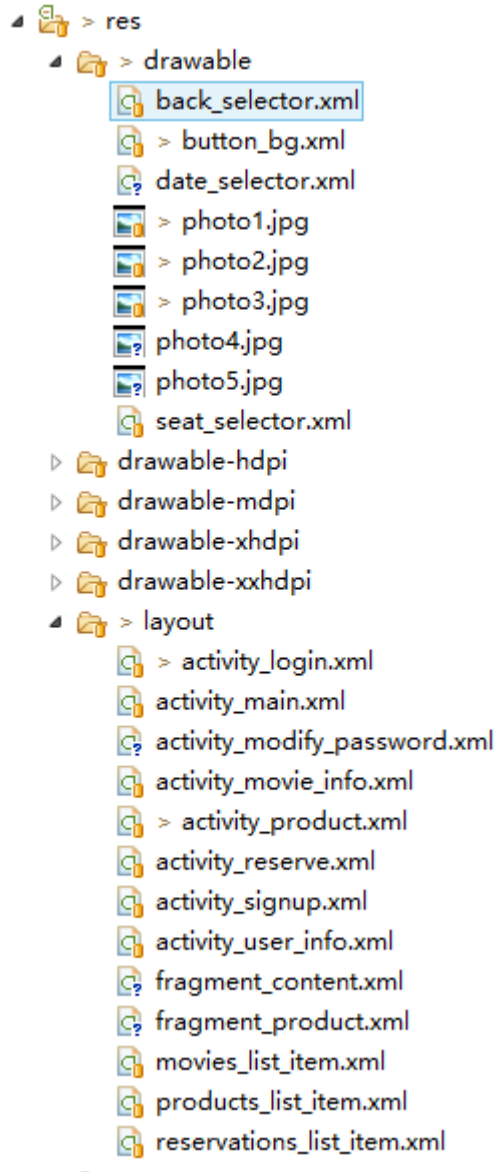
本次作业采用 Android 应用开发技术。Android 系统是市场上两大手机系统之一，其市场份额位居全球第一，受众广泛。

其次就是因为 Android 平台的开发性。开发的平台允许任何移动终端厂商加入到 Android 联盟中来，显著的开放性使其拥有更多的开发者。Android 平台提供给第三方开发商一个十分宽泛、自由的环境，不会受到各种条条框框的阻扰，更易于开发。

二、模块划分

划分了数据库模块，实体类模块，页面模块以及资源模块。





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。

三、设计技术

1. 结构化编程

将软件划分成不同模块，降低了耦合性，容易维护和扩展。整个软件根据 MVC 架构分为底层数据库模块，业务处理模块，界面和资源模块。

1) 在每个页面逻辑的代码中，也体现了结构化的思想。每个 Activity 里面，凡是用于初始化控件的操作都放在 initView()函数中。

```
// init variables
private void initView() {
    dbHandler = DataBaseHelper.getInstance(this);
    _phone_edt = (EditText)findViewById(R.id.signup_phone_edt);
    _name_edt = (EditText)findViewById(R.id.signup_name_edt);
    _pw_edt = (EditText)findViewById(R.id.signup_password_edt);
    _pw_confirm_edt = (EditText)findViewById(R.id.signup_confirm_password_edt);
    _signup_btn = (Button)findViewById(R.id.signup_signup_btn);
}
```

2) 凡是用于设计控件交互动作的都放在 initEvent()函数中

```
private void initEvent() {
    _signup_btn.setOnClickListener(new OnClickListener() {

        @Override
        public void onClick(View v) {
            // TODO Auto-generated method stub
            startActivity(new Intent(LoginActivity.this, Sig
        }
    });

    _login_btn.setOnClickListener(new OnClickListener() {

        @Override
        public void onClick(View v) {
            // TODO Auto-generated method stub
            loginFunction();
        }
    });
    _forget_pw.setOnClickListener(new OnClickListener() {
```

3) 在 DataBaseHelper 中

(i)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。

```
public final static String TABLE_CINEMA = "CINEMA";
public final static String TABLE_LOCATION = "LOCATION";
public final static String TABLE_MOVIE = "MOVIE";
public final static String TABLE_PRODUCT_DESCRIPTION = "PRODUCTDESCRIPTION";
public final static String TABLE_RESERVATION = "RESERVATION";
public final static String TABLE_SCREENING_ROOM = "SCREENINGROOM";
public final static String TABLE_USER = "USER";
```

(ii) 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。

```
public long addUser(User user) {
        if (this.queryUser(user.getUser_id()) != null) return 0;
        SQLiteDatabase db = this.getWritableDatabase();
        ContentValues values = new ContentValues();
        values.put(USER_ID, user.getUser_id());
        values.put(USER_NAME, user.getName());
        values.put(USER_PASSWORD, user.getPassword());
        values.put(DIST_CODE, user.getDist_code());
        return db.insert(TABLE_USER, null, values);
}
```

4) 在具体编程中，代码清晰风格统一也属于结构化编程的原则之一。

我们的代码规范是:

1.所有函数以首字母小写后面大写命名。

```
// Login
private void loginFunction() {
```

```
// Update reservation info
public int updateReservation(Reservation reservation) {
```

2.所有类内部私有变量均以下划线为前缀。

```
public class LoginActivity extends Activity{
    private Button _login_btn, _signup_btn;
    private EditText _phone_edt, _password_edt;
    private TextView _forget_pw;
```

3.所有静态变量大写。

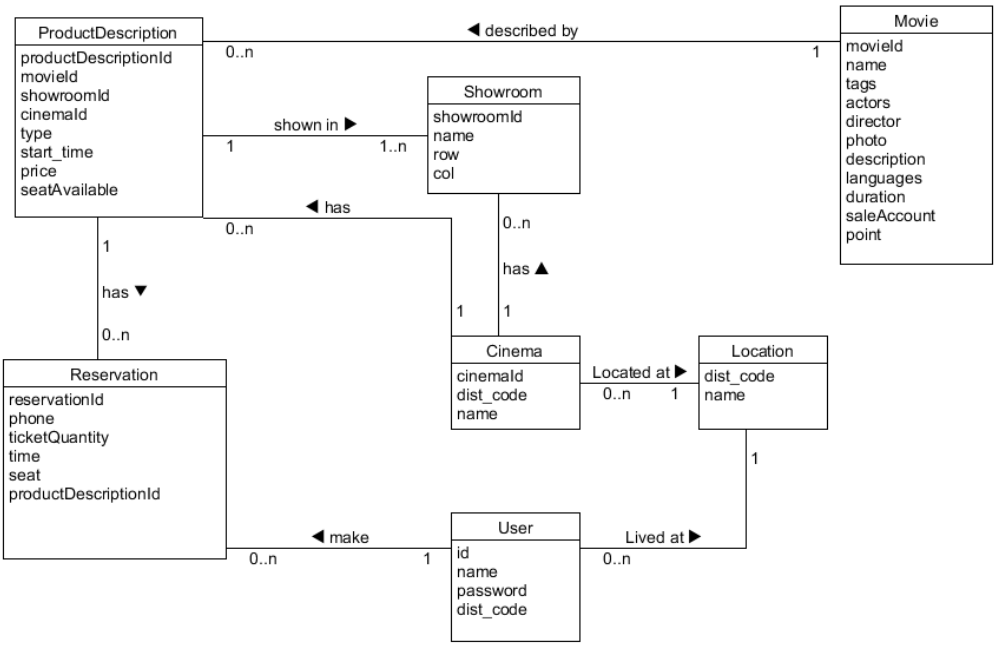
```
public final static String TABLE_CINEMA = "CINEMA";
public final static String TABLE_LOCATION = "LOCATION";
public final static String TABLE_MOVIE = "MOVIE";
public final static String TABLE_PRODUCT_DESCRIPTION = "PRODUCTI
public final static String TABLE_RESERVATION = "RESERVATION";
public final static String TABLE_SCREENING_ROOM = "SCREENINGROOM
public final static String TABLE_USER = "USER";
//列名
public final static String USER_ID = "UserId";
public final static String USER_NAME = "Name";
public final static String USER_PASSWORD = "Password";
public final static String DIST_CODE = "DistCode";
public final static String MOVIE_ID = "MovieId";
public final static String MOVIE_NAME = "Name";
public final static String MOVIE_TAG = "Tag";
public final static String MOVIE_ACTORS = "Actors";
public final static String MOVIE_DIRECTOR = "Director";
public final static String MOVIE_PHOTO = "Photo";
public final static String MOVIE_DESCRIPTION = "Description";
public final static String MOVIE_LANGUAGES = "Languages";
public final static String MOVIE_DURATION = "Duration";
```

4.良好的缩进风格。

2. 面向对象编程

我认为在整各软件设计过程中，编码并不是最重要的工作，需求分析和概要设计才是最重要。在明确了所有需求点（用例），重心应该放在领域模型的设计中。领域模型的设计很好地体现了面向对象编程的思想。让程序由现实中来，所有在程序里面的类应该抽象自现实生活中的具体物体和某个概念。

例如在这个电影订票系统里面，很显然的电影，电影院，用户，放映厅这些现实中存在的东西应该是一个类。但显然这还是不够的，有一些概念也需要抽象成类，比如说预订。预订在现实中是一个动作，但想想就知道软件的核心业务就是预订电影票，那每一次预订应该被记录，每一个预订都是具体的东西，所以预订应该是一个类。同样的如果在用例描述中出现的东西不是某个类的属性，那他很有可能就是一个概念类了。在确定了所有的概念类之后，找出概念类的属性和关系即完成领域模型的建模，我们的领域模型如下:



3. 设计模式

设计模型是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性、程序的重用性。

在实验中我们主要使用了两种设计模式。

1.单例模式

考虑对数据库管理类使用的频繁性和数据库操作的可能存在的并发性，在数据库管理类的实现中用了单例模式，并且用了 java 的 synchronized 关键字实现同步原语。

单例模式具体实现方式有很多，这里用了 3d 游戏课里面的一种方式：

首先将构造函数私有化，这样做的目的是禁止外部通过构造函数实例化单例模式下的类，要想得到这个类的实例对象只能调用静态的 getInstance 函数。

```
private DataBaseHelper(Context context) {
    super(context, DATABASE_NAME, null, DB_VERSION);
}
```

然后类里面定义了私有静态变量_instance，这个变量的类型就是单例模式的类。最后定义类静态函数 getInstance()，返回这个类的实例。

```
private static DataBaseHelper _instance = null;

public static synchronized DataBaseHelper getInstance(Context context) {
    if (_instance == null) {
        _instance = new DataBaseHelper(context);
    }
    return _instance;
}
```

2. 委托代理模式

由于 Java 里面没有 Delegate 关键字，实现委托代理模式主要通过定义接口，代理者实现接口。在页面嵌有几个 fragment 时，应该使用委托代理模式。fragment 的出现使得 activity 的耦合性进一步降低，让 activity 里面的代码不再那么臃肿。每个 fragment 相当于 activity 里面的一部分，有自己独立的生命周期。使用 fragment 可以有效分化一个页面里面的复杂业务过程和交互任务。但是由于一个页面里面有多个 fragment，当需要对主页面进行操作时，如果都在自己的 fragment 里面实现，那各个 fragment 之间就不再是相互独立的了，容易造成页面混乱。这里我使用了委托代理模式，所有对主页面的操作都定义成一个接口，主页面实现这个接口。当有需要时 fragment 只需要调用主页面的接口即可将任务“委托”给主页面。

在 fragment 类定义接口，传入要执行的任务名，以及参数。

```
/**
 * This interface must be implemented by activities that contain this
 * fragment to allow an interaction in this fragment to be communicated to
 * the activity and potentially other fragments contained in that activity.
 */
public interface OnFragmentInteractionListener {
    // TODO: Update argument type and name
    public void onFragmentInteraction(String commend, int arg);
}
```

主页面实现接口

```
public class ProductActivity extends Activity implements OnFragmentInteractionListener{
    @Override
    public void onFragmentInteraction(String commend, int arg) {
        // TODO Auto-generated method stub
        if (commend.equals("itemclick")) {
            Intent intent = new Intent(ProductActivity.this, Rese
            intent.putExtra("product", _products.get(arg));
            startActivityForResult(intent, 111);
        }
    }
```

fragment 调用

```
if (mListener != null) {
    int arg = (Integer) _listItems.get(arg2).get("product_index");
    mListener.onFragmentInteraction("itemclick", arg);
}
```

4.自适应界面布局

背景和难点

由于 Android 是谷歌的一个开源的操作系统，生产基于这个操作系统的手机的厂商众多，比较出名的有：华为、三星、小米等，这就导致了 Android 手机的尺寸千差万别，所以完成一个 android 应用的自适应功能比苹果应用要困难得多。

这里主要有几种方法去实现自适应布局：

1. 关于 UI 对象的尺寸必须用 dp 去定义，不要用 px。首先 px 是指像素，而 dp（dip）是指 device independent pixels(设备独立像素)。因为现在的手机都是高清屏，每个物理点表示的像素都超过一个。如果仍然用像素表示尺寸，那么在低 dpi(每英寸点数)的手机上可能看起来很大，但在高 dip 手机上就会很小了。所以推荐使用 dip 来描述尺寸大小，因为 dip 不依赖像素大小。

Dip 和 px 的换算公式是：

Px = dip * density / 160

特别的，当屏幕密度为 160 时，px = dip

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="50dp"
```

2. 在尺寸相差过大的时候应该用另一个布局文件，也就是为每一个尺寸范围类的屏幕提供一个布局文件，这样每个页面就会对应多个布局文件了，虽然有些麻烦，但这是解决自适应问题的最好方法。Eclipse 支持了这种做法，在资源文件夹（res）中存放布局文件的可以有多个文件夹：一般要有 layout-small, layout-large,layout-xlarge 三种。

3. 同样的，图片资源也要备多份，根据尺寸不同从小到大分别放在 drawable-ldpi, drawable-mdpi, drawable-hdpi, drawable-xhdpi, drawable-xxhdpi 五个文件夹中。

4. 尺寸也要备多份，虽然用 dp 作为尺寸单位很多地解决了低分辨率屏和高清屏的问题，但手机的物理尺寸大小还是会影响到布局的。所以对于物理尺寸小的手机用小尺寸，大的用大尺寸。这通过自定义尺寸实现，自定义尺寸用 xml 描述，放在不同的 values 文件夹就可以了。

5. 多用 match-parent 和 wrap-content，巧用 layout-weight，weight 属性是代表该控件在父容器中所占大小的比例，巧用 weight 可以让容器里的控件按比例分配大小。

```
<TextView
    android:id="@+id/product_today"
    android:layout_width="match_parent"
    android:layout_height="match_parent
    android:layout_weight="1"
    android:background="@drawable/date_
    android:text="今天"
    android:gravity="center"
    android:textSize="18sp"
    />
<TextView
    android:id="@+id/product_tomorrow"
    android:layout_width="match_parent"
    android:layout_height="match_parent
    android:layout_weight="1"
    android:background="@drawable/date_
    android:text="明天"
    android:gravity="center"
    android:textSize="18sp"
    />
```

6. 在 java 代码里面动态调整布局，虽然最后一个方法很多，但不是很提倡，因为这破坏了 Android 本来很好的 MVC 架构，增加了代码的耦合性。 由于在选座页面中每个放映厅的大小不同，座椅的行列数不同，所以无法通过静态文件定义界面，这里通过 java 动态添加界面元素实现。

```
private void initSelectZoom() {
    if (_product == null) return;

    _seat_state = _product.getSeat_availible().split(",");
    screeningRoom = dbHandler.queryScreeningRoom(_product.getSc

    if (screeningRoom == null) return;

    for (int i = 0; i < screeningRoom.getRow(); i++) {
        LinearLayout linearLayout = new LinearLayout(this);
        linearLayout.setOrientation(LinearLayout.HORIZONTAL);
        linearLayout.setLayoutParams(new LinearLayout.LayoutPar

        for (int j = 0; j < screeningRoom.getCol(); j++) {
            int index = i * screeningRoom.getCol() + j;
            ImageView imageView = new ImageView(this);
            LinearLayout.LayoutParams lp = new LinearLayout.Lay
            lp.weight = 1;

            imageView.setLayoutParams(lp);
            imageView.setAdjustViewBounds(true);
            }
        });

        if (_seat_state[index].equals("1")) {
            imageView.setImageResource(R.drawable.seat_selector);
        } else {
            imageView.setImageResource(R.drawable.ic_seat_have_selected);
            imageView.setClickable(false);
        }

        _select_items.add(imageView);
        linearLayout.addView(imageView);
    }

    _select_zoom.addView(linearLayout);
}
```