说说Android的MVP模式

http://toughcoder.NET/blog/2015/11/29/understanding-Android-mvp-pattern/

安卓应用开发是一个看似容易，实则很难的一门苦活儿。上手容易，看几天Java，看看四大组件咋用，就能整出个不太难看的页面来。但是想要做好，却是很难。系统框架和系统组件封装了很多东西，开发者弄几个Activity，用LinearLayout把布局组合在一起，添加点事件监听，一个应用就成型了。红海竞争，不管多么复杂的UX和业务逻辑都是一个月快速上线，二周一个迭代，领导和产品早上改需求，晚上改设计，再加上产品经理和设计师都按照iOS来设计，这一系列原因导致很多安卓应用不但体验差，不稳定，性能低，而且内部代码相当之混乱，即使BAT也是如此。

反观国外市场（谷歌应用市场）上面的大部分应用都还是比较好的，表现在符合安卓设计规范，性能和稳定上表现不俗，体验上更符合安卓系统，而且会发现他们的代码也是很有设计思想的。GitHub上面的很多安卓开源项目也都是源自国外的优秀开发者以及他们的项目。

安卓应用也是软件，代码结构合理，层次清晰不但容易维护而且还容易做自动化测试和单元测试，这是开发者的美好愿望，也是提升效率的必然之路。

安卓由于系统架构特性，UI组件Activity中融合了View的处理，事件处理和逻辑处理，随着业务的越来越复杂，导致Activity也越来越雍肿，几千行的Activity随处可见，Fragment也不能解决问题，千行以上的Fragment也不在少数，这个时候就完全不要谈什么可维护性，可测试性了。能完成需求就算高手了。

MVP便应运而生，就来解决这些问题的。

什么是MVP模式

MVP是针对有GUI存在的应用程序，比如像安卓，像水果以及PC的客户端软件中用以划分组织代码的一种设计模式，是由MVC模式升级演进出来的，目的在于，对于GUI层来说，把UI展示与逻辑分开。

- Model – 为UI层提供的数据，或者保存UI层传下来的数据
- View – 单纯的展示数据，响应用户操作并都转发给Presenter来做具体的处理
- Presenter – 逻辑控制层，从Model处取数据，运算和转化，最后用View来展示；并处理View传过来的用户事件，并做处理

需要注意的是MVP仅用于应用中的GUI部分，它并不是整个应用的架构方式。一个应用的主要的架构应该包括基础组件，业务逻辑层和GUI展示层，而MVP仅是用于展示层的设计模式。另外，它是一个方法论的东西，没有固定的实现方式，只要能体现出它的方法就可以算是MVP。

虽然是方法论，但是也有一些指导性的原则来约束实现：

- Model与View不能直接通信，只能通过Presenter
- Presenter类似于中间人的角色进行协调和调度
- Model和View是接口，Presenter持有的是一个Model接口和一个View接口
- Model和View都应该是被动的，一切都由Presenter来主导
- Model应该把与业务逻辑层的交互封装掉，换句话说Presenter和View不应该知道业务逻辑层
- View的逻辑应该尽可能的简单，不应该有状态。当事件发生时，调用Presenter来处理，并且不传参数，Presenter处理时再调用View的方法来获取。

从这里可以看的出来，其实，MVP的目的就是把GUI的逻辑都集中在Presenter层，又把View层和Model与其用接口分离，让View尽可能的简单，这样可以加强移植性。因为View层是肯定不能移植的，不同的平台GUI的窗口部件肯定不一样，Model也是不太好移植的，因为每个平台的IO也都是不一样的。但是，MVP中的P肯定是可以移植的，因为它里面只有逻辑，且View和Model都是接口，所以很容易移植。同时，因为View和Model都是接口，这个Presenter也非常好测试，只要实现一个View的接口和Model的接口，就可以单独的测试Presenter了。

严格来讲，View只是被动的显示，提供方法由Presenter来调用，数据等都是由Presenter来提供，内部不能任何的逻辑与状态，逻辑和状态都应该是在Presenter中。UI事件发生时，调用Presenter的方法来处理，不能传参数，也

不能有返回值，在Presenter中处理后再调用View来更新数据和状态。

MVP与MVC的区别

MVC之中逻辑是放在了Model里，Controller负责桥接View和Model，View发生变化时通知Controller，Controller再通知Model，Model进行逻辑处理，更新数据，然后通知View来刷新。可以看到MVC中三者之间都有联系，如果处理不好，或者当View比较复杂时，三者之间都会双向关联。MVC在命令行应用，以及WEB中有大量的应用，但对于客户端（PC和移动端）的GUI应用，MVC往往解决不了复杂性，移植性上以及可测试性上也没有优势。

MVP的改进在于：

- 逻辑放在Presenter中
- View和Model抽象成为接口

这样就带了二个好处：

- 代码更加容易移植
- 代码更加容易加入Unit Testing

如何在安卓中实践MVP

MVP是一个方法论的东西，也就是没有任何固定的具体的实现形式，只要能够把View跟Model解除联系，把逻辑都放在Presenter中，那么就能算得上是MVP，一些具体的实践的指导性原则：

- View是一个接口，负责被动的把处理好的数据显示出来
- Model也是一个接口，负责获取数据和存储数据
- View调用Presenter处理用户事件也是一个接口，称为事件Delegate
- Presenter持有的是View的接口和Model接口

安卓的Activity是一个比较奇葩的角色，在MVP中，既可以用作V，因为一个应用的根布局总是由Activity来创建的。当然也可以当作P，因为Activity是一个应用的入口，也是出口，再加上一些关键的系统事件也都是通过Activity的方法来通知的（比如configChange, saveInstance）。其实，都可以。因为MVP是方法论，并没有固定的形式，只要是把数据处理的逻辑都封装在

Presenter里，让其去控制View和Model，让Activity来承担View还是Presenter，其实都可以。

MVP不是银弹，仅是展示层的一种范式而已

最重要的一点就是要明白，MVP不会拯救你的应用，不要以为使用了MVP就能让代码更容易维护，更少的Bug，添加新功能会更容易。MVP仅是GUI层的一种编程范式而已，且因为它是方法论的东西，对实现方式并没有固定的形式，所以会被滥用，如果没有深刻理解MVP的思想，更加会导致灾难性的结果。

软件，移动应用也不例外，如果功能简单，业务简单，那么代码怎么写其实也都无大碍，但当功能越来越多，业务越来越复杂的时候，就必须要采取必要的方法来应对复杂度和软件的可开发性，可维护性。比如，说的夸张一点，一个helloworld式的应用，你怎么写都可以。但当功能复杂到一个Activity几千行代码的时候，你再怎么MVP，MVC或者MVVM都不能解决问题，再怎么把Activity当成P或者当成V都没有用。

要知道MVP仅是解决GUI应用程序中展示层的问题，并且它带来的最大的好处是方便测试和移植，因为逻辑都在P里面，P持有的又仅是View和Model的接口，所以P是可测试的，Mock一个View的实现，和Mock一个Model的实现，就可以完全脱离平台和框架的限制来自由的测试P。同样，移到一个新的框架和平台后，只需要实现View和Model就可以了，P是不需要改变的。

分层和模块化才是解决应用越来越复杂之道

分层

所谓分层，也就是应用程序的架构方法，把应用程序分成好多层，可以参考Bob大叔的The Clean Architecture。

至少应该分层三层，最底层是平台适配层，把用到的平台的组件，控件，工具，比如UI组件，数据库等等，进行封装；中间层就是业务层，就是你应用的核心的业务逻辑，或者说你的应用解决了用户什么样子的问题，这一层是不会随着平台和UI的改变而改变的。比如新闻阅读类，那么从服务器拉取数据，解析数据，缓存数据，为上层提供数据这些事情都属于业务层；最上面

就是展示层或者叫做UI层。展示层是可以调用业务层的方法和数据。这样分层，可以让展示层只是负责与用户交互，展示业务数据，展示层会变得简单很多，同时业务层因为不涉及具体的平台和UI的细节，就非常容易移植，当移植到新平台或者要做UI改版也是非常容易做的。

模块化

另外一个就是模块化，其实这是软件开发的一个非常基本的方法，也是非常有用的一个方法。模块划分的方法非常简单就是按照功能来划分。让模块处理好自己的事情，暴露统一的接口给外部，定义好输入与输出。输入就以参数和方法形式暴露，输出最好以Delegate方式，这样能把耦合降到最低。再由一个统一的顶层类来管理各个模块，顶层直接调用各模块，各模块通过Delegate方式来回调管理者。

对于业务层，模块化相对比较容易，因为这里并不涉及UI和平台的特性，业务层都应该是独立的，可移植的，全都是自己写的类。

但对于展示层，通常没有那么的容易，因为有平台的限制。比如说安卓，根布局必须由Activity来创建。首先，模块的划分也要以功能为界限。然后，就是Activity的布局，要把布局按功能区域来管理，然后把每个功能模块的top Container传给模块，具体内部如何布局，如何填充数据，就由模块自己负责。Activity就起管理各个模块的作用。再有，模块间的通信，可以都通过Activity来，比如模块1有模块2的入口按扭，但是模块1与模块2之间没有交集，这个时候的处理方式就是模块1Delegate给Activity，然后Activity再调用模块2来显示和隐藏。如果模块多到Activity的管理工作也变得庞大复杂时就要拆出子Controller来管理模块，也就是三层，甚至还可以四层。模块的原则就是做好封装，让外层管理变得简单，这样外层管理的复杂度就会降下来，就好比公司人员的组织架构一样。

```
<LinearLayout>
  <LinearLayout id="module1" />
  <RelativeLayout id="module2" />
  <ListView id="module3" />
</LinearLayout>
```

```
public class DemoActivity extends Activity implements Module1Delegate, Modu
```

```
  @Override
  public void onCreate(Bundle bundle) {
    setContentView(R.layout.demo_activity);
    Module1 module1 = new Module1(findViewById(R.id.module1), this);
    Module2 module2 = new Module2(findViewById(R.id.module2), this);
    Module3 module3 = new Module3(findViewById(R.id.module3));
    module1.render();
    module2.render();
  }

  @Override
  public void onModule1() {
    Log.e("Demo", "module1 say hello to the world.");
  }

  @Override
  public void onModule2(boolean show) {
    if (show) {
      module3.show();
    } else {
      module3.hide();
    }
  }
```

其实，还可以做的更彻底一些，那就是Activity中的布局都由ViewStub来组装，然后由各个子模块来决定如何布局。

对于多层全屏层叠的应用来说，要简单一些，对于每一层都可以由Activity或者Fragment来实现，如果业务层已经抽离出来，就都可以直接调用业务层来获取数据，因此也不会有传递数据的麻烦。

做好了分层和模块化，我相信，能解决绝大多数应用遇到的问题。至于模块内部用什么MVP，MVC，MVVM，其实真的无大害，因为模块内部的实现方式不影响其他模块，也不影响外部管理和level更高的类。

把基本的原则做到就够了

编程是一项社会活动，所以人和人与人之间的关系才是核心，优秀的人，你发现他也没有用什么MVP，什么MVC，什么高大上的设计模式和算法，但是他的代码是很清晰，很容易看懂。有些即使号称用什么高大上的，最先进的设计模式，但是代码仍是一坨坨的，可能连他自己都看不懂。

把基本的抽象和封装真正做到位了，就够了，代码水平可以的话，再能做到命名见名知义，小而活的方法，小而活的类，一个方法只做一件事，一个类

只做一件事情。做到这些，也就够了。

至于什么高大上的MVP，什么XP，什么TDD，什么结对，其实都是浮云，如果你的水平比较高，代码sense较高，那么用不用这些方法差别不大。

MVP的核心目的是方便UT，因为把展示层的逻辑都集中在P，而P又不依赖于具体的View和Model，所以可以随便Mock一个View和一个Model来测试P，甚至P可以独立于平台的限制来单独的测试。所以，如果你不搞UT，以不以MVP方式来实现，其实没啥影响，甚至网上不少人还专门为MVP而弄出几个抽象的类，把Activity啥的封装了一下，号称MVP框架，毫无实用价值。软件方法，切忌生搬硬套，一定要先理解透彻方法，再理解透彻你的问题和环境限制，然后灵活运动，什么叫理解透彻呢？就是你能给别人讲明白时。这说起来还是太抽象，只能在实际运用中慢慢领悟。

再有就是Unit Testing这玩意儿，实际的意义也没有那么大，要知道写测试代码通常要比生产代码花更多的精力，前提还是你的代码写的可测，可测性比可读性还要难一点，说白了这对开发者水平的要求相当的高，不是看了一遍书，学习一下JUnit就能搞得好的。还有就是如果你的需求经常变动，移动互联时代这是家常便饭，那么做UT会让开发量double甚至tripple，因为之前写的UT全没有用。

还想说一点就是，软件开发方法这东西必须是由上向下推动，也就是由老板带头来推动，否则技术小组长或者开发者自己是很难推得动的。特别是像UT，Code Review或者结对之类的会“降低开发效率”的方法。这些方法短期内不会提升效率和质量，只会降低需求的产出率，平均开发水平比较高的团队也至少要几个月后才能真正的适应这些方法，然后才有可能提高效率和提高质量。如果不是老板主动推动，谁能受得了呢？KPI咋整？

结论

MVP或者MVVM带来最大的好处是：

- 方便移植
- 方便UT

另外，要注意MVP仅是展示层的方法论。应用整体还是要进行分层和模块化。如果分层和模块化进行的彻底，并且在移植和UT没有强烈的需求，其

实MVP与不P真的不重要。

参考资源

- Android中的MVP
- MVP模式在Android开发中的应用