对于大部分面向最终用户的应用来说，它们都需要具有一个可视化的UI与用户进行交互，我们将这个UI称为视图（View）。在早期，我们倾向于将所有与视图相关的逻辑糅合在一起，这些逻辑包括数据的呈现、用户操作的捕捉与相应以及和针对数据存储（比如数据库）的操作。我们将这种设计模式称为自治视图（AV，Autonomous View）。

目录   
一、自治视图   
二、MVC模式   
三、多层架构中的MVC

**一、自治视图**

说到自治视图，可能很多人会感到模式，但是我想很多人（尤其是.NET开发人员）可能经常在采用这种模式来设计我们的应用。Windows Forms和ASP.NET Web Forms虽然分别属于GUI和Web开发框架，但是它们都采用了事件驱动的开发方式。所有与UI相关的逻辑都可以定义在针对视图（Windows Form或者Web Form）的后台代码（Code Behind）中，并最终注册到视图本身或者视图元素（控件）的相应事件上。

一个典型的人机交互应用具有三个主要的关注点，即数据在可视化界面上的呈现、UI处理逻辑（用于处理用户交互式操作的逻辑）和业务逻辑。对于自治视图模式来说，它实际上这三种混合在一起，势必会带来如下一些问题：

首先，业务逻辑是与UI无关的，应该最大限度地被重用。由于业务逻辑定义在自治视图中，相当于完全与视图本身绑定在一起。如果我们能够将UI的行为抽象出来，基于抽象化UI的处理逻辑也是可以被共享的，定义在自治视图的UI处理逻辑完全丧失了重用的可能。

其次，业务逻辑具有最强的稳定性，UI处理逻辑次之，而可视化界面上的呈现最差，比如我们经常会为了更好的呈现效果来调整HTML。将具有不同稳定性的元素融为一体，具有最差稳定性的元素决定了整体的稳定性，这是“短板理论”在软件设计中的体现。

再次，任何涉及到UI的组件都不易测试。UI是呈现给人看的，并且用于与人进行交互，用机器来模拟活生生的人来对组件实施自动化测试不是一件容易的事，自治视图严重损害了组件的可测试性。

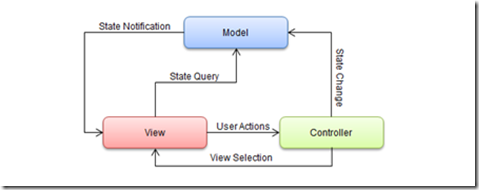
为了解决自治视图导致的这些问题，我们需要采用采用关注点分离（SoC， Seperation of Concerns）的方针将可视化界面呈现、UI处理逻辑和业务逻辑三者分离出来，并且采用采用合理的交互方式将它们之间的影响降到最低。由于将三者“分而治之”，自然也使UI逻辑和业务逻辑编程的容易被测试的组件，使测试驱动设计与开发变成了可能。这里用于进行关注点分离的模式就是MVC。

**二、MVC模式**

MVC的创建者是Trygve M. H. Reenskau，他是挪威的计算机专家，同时也是奥斯陆大学的名誉教授。MVC是他在1979年访问施乐帕克研究中心（Xerox PARC，Xerox Palo Alto Research Center）期间是提出一种主要针对GUI应用的软件架构模式。MVC最初用于SmallTalk，Trygve最初对MVC的描述记录在《Applications Programming in Smalltalk-80(TM):   
How to use Model-View-Controller (MVC)》这篇论文中，有兴趣的读者可以通过地址<http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html>阅读这篇论文。MVC体现了关注点分离这一基本的设计方针，它将构成一个人机交互应用涉及到的功能分为Model、Controller和View三部分，三者各自具有的基本职责或者功能范围如下：

* Model：是对应用状态和业务功能的封装，可以看成是同时包含数据和行为的领域模型（Domain Model）。Model接受Controller的请求执行相应的业务功能，并在状态改变的时候通知View。
* View：实现可视化界面的呈现，捕捉最终用户的交互操作（比如鼠标和键盘操作）。
* Controller：View捕获到用户交互操作后会直接转发给Controller，后者完成相应的UI逻辑。如果需要涉及业务功能的调用，Controller会直接调用Model。在完成UI处理之后，Controller会根据需要控制原View或者创建新的View对用户交互操作予以响应。

下图揭示了MVC模式下Model、View和Controller之间的交互。对于传统的MVC模式，很多人认为Controller仅仅是View和Model之间的中介，实则不然，View和Model存在直接的联系。View可以直接调用Model查询其状态信息。当Model状态发生改变的时候，它也可以直接通知View。比如在一个提供股票实时价位的应用，维护股价信息的Model在股价变化的情况下可以直接通知相关的View改变其显示信息。

[](http://images.cnblogs.com/cnblogs_com/artech/201203/201203081758064689.png)

从消息交换模式的角度来讲，Model针对View的状态通知和View针对Controller的用户交互通知都是单向的，我们推荐采用事件机制来实现这两种类型的通知。从设计模式的角度来讲就是采用观察者（Observer）模式通过注册/订阅的方式来实现它们，即View作为Model的观察者通过注册相应的事件来检测状态的改变，而Controller作为View的观察者通过注册相应的事件来处理用户的交互操作。

**三、多层架构中的MVC**

我看到很多人将MVC和所谓的“三层架构”进行比较，其实两者并没有什么可比性，MVC更不是分别对应着UI、业务逻辑和数据存取三个层次。不过两者也不能说完全没有关系，我们现在就来讨论这个问题。

Trygve M. H. Reenskau当时提出MVC的时候实际上将其作为构建整个GUI应用的架构模式，而Model维护着整个应用的状态并实现了所有的业务逻辑，它更多地体现为一个领域模型。而对于多层架构来说（比如我们经常提及的三层架构），MVC是被当成是UI呈现层（Presentation Layer）的设计模式，而Model则更多地体现为访问业务层的入口（Gateway）。如果采用面向服务的设计，将业务功能定义成相应服务并通过接口（契约）的形式暴露出来，这里的Model甚至还可以表示成进行服务调用的代理。