第3章 软件体系结构风格

内容

- 3.1 概述
- 3.2 数据流风格
- 3.3 过程调用风格
- 3.4 独立构件风格
- 3.5 层次风格
- 3.6 虚拟机风格
- 3.7 客户/服务器风格
- 3.8 表示分离风格
- 3.9 插件风格
- 3.8 微内核风格
- 3.11 SOA风格

表示分离风格

- 3.8.1 MVC风格
- 3.8.2 MVP风格
- 3.8.3 MVVM风格（自学）
- 3.8.4 案例分析
- 3.8.5 优缺点分析

“表示分离”的含义

- 在软件产品中，用户界面是最容易发生变化的部分。对于一款好的软件而言，更改用户界面不应影响系统核心功能，因为这部分功能应是稳定的，较少发生变化。此外，这种更改还应当比较容易，并且只局限在需修改的界面部分。
- 为满足用户界面的可变性需求，通常需要将交互式应用划分为多个独立部分，将表现与核心功能分离，并采用相应变化传播机制来确保各部分的协调一致。这样的设计风格即表示分离风格。

3.8.1 MVC风格

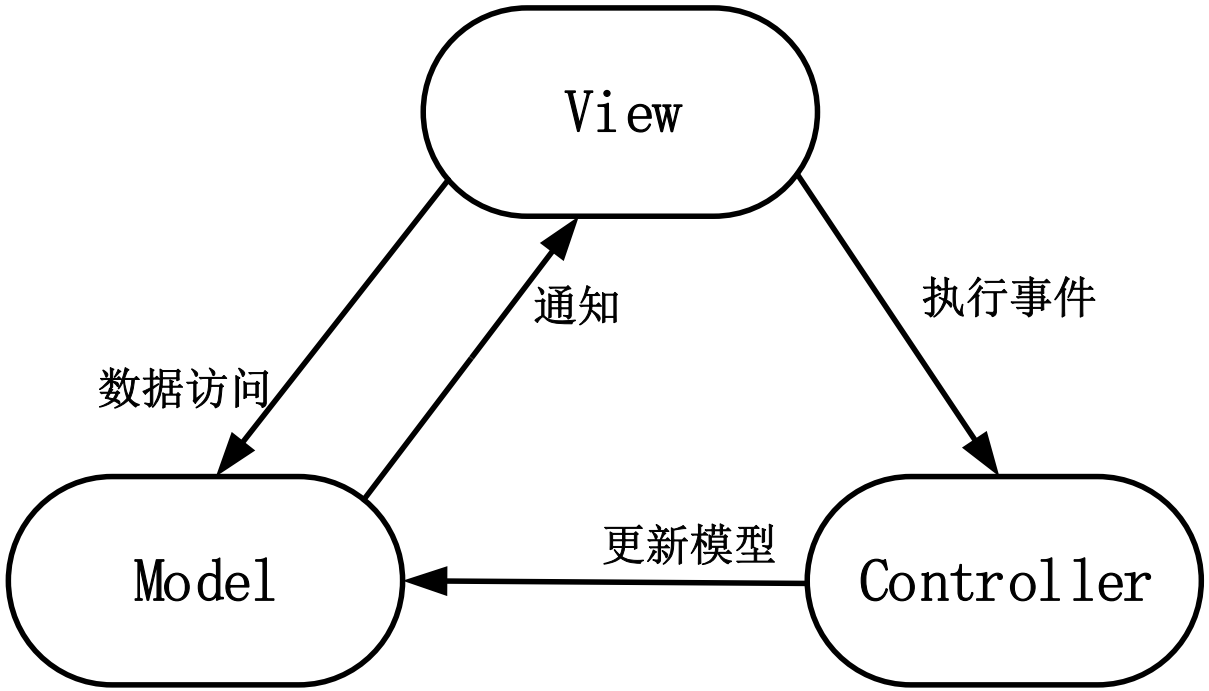
MVC是Model-View-Controller的简称，即模型-视图-控制器。

什么是MVC？

- MVC是一种架构模式，它将用户和应用之间的交互分离为3个角色：模型（业务逻辑），视图（用户接口），和控制器（用户输入）。这种关注点的分离有利于每个角色的开发、测试和维护。
- MVC模式最早由Trygve Reenskaug在1978年提出，是施乐帕罗奥多研究中心（Xerox PARC）在20世纪80年代为程序语言Smalltalk发明的一种软件设计模式。
- 在1990年后，MVC被普遍分类为一种架构模式
- MVC的目标是实现model和view的解耦，增加源代码灵活性和可维护性。

MVC的工作原理

- 构件：模型，视图，控制器
- 连接件：显式调用、隐式调用或其他机制（如HTTP协议）



模型（Model）

- 职责：
  - 提供应用程序的核心功能（数据和数据相关功能）
  - 注册对模型数据感兴趣的视图和控制器
  - 数据改变时通知注册者（主动模式）
- 模型提供访问数据的程序或方法。
- 这些程序和方法被控制器调用以响应用户的操作。
- 模型必须保持其数据为最新的，因此需要有内部更新数据的机制并将数据改变通知给相应视图。（这种改变的传播机制可使用发布者-订阅者模式实现）。
- 模型的变种：保持被动，不发布更新。这种模型中，视图和控制器主动询问模型来进行更新，而不是进行订阅等待更新。

视图（View）

- 职责：
  - 创建并初始化其控制器
  - 将信息显示给用户
  - 当有新数据从模型到达视图，更新自身
  - 从模型中取回数据
- 初始化过程中，所有视图在模型中进行注册，确保视图拥有最新的数据。
- 通常，视图和控制器存在一对一的关系。每个视图有一个控制器。每个视图也可能有多个子视图（按钮，滚动条，菜单都是子视图）。
- 一般来说，视图负责控制器的创建。

控制器（Controller）

- 职责：
  - 接收用户输入事件
  - 将事件翻译为对模型的请求或者视图的显示请求
  - 当有来自模型的新数据到达时，更新自身
- 控制器的行为有时依赖于模型的状态。这种情况下，控制器必须注册了模型的改变传播方法（change-propagation method）
- 视图可以拥有多个控制器。如：屏幕的一部分元素能够编辑，而其他部分不行。这种情况下，可以将这些元素的控制器进行分离。

MVC模式实现机制（1）

- **步骤1**：将核心功能从UI行为中分离出来。
  - **核心数据**部分是什么？
  - 在数据上有哪些需要**计算的功能**？
  - 系统期望的**输入**是什么？
  - 其次，需要设计**模型构件**以存储数据并执行核心计算功能，并设计一些视图将会用到的**数据访问函数**。
  - 此外，还需决定哪些数据和功能将被视图和控制器直接访问，并**定义访问接口**。

MVC模式实现机制（2）

- **步骤2**：建立改变传播机制。
  - 通过设计**注册表**，可以帮助模型记住哪些视图和控制器订阅的相应数据，与此同时也需要设计视图和控制器进行订阅和取消订阅的方法。
  - **模型发布更新**数据时，应该通知所有订阅的视图和控制器调用其更新函数。
  - 当模型中的数据没有改变，而视图或控制器需要访问它时，需要建立一个**独立的访问机制**。这个机制使得视图和控制器启动时能够请求数据的当前状态。

MVC模式实现机制（3）

- **步骤3**：设计和实现视图。
  - 对每个视图，需要设计其外观并创建需要的绘图软件来进行显示。绘图软件通过步骤 2中定义的方法访问数据。
  - 两种图形更新策略：第一是在模型中提供额外信息，描述改变的大小（当改变较小时，调用一些其他视图更新程序，而非完全更新整个视图）；第二种方式则是等待有较大的更新时，再一次性更新视图。

MVC模式实现机制（4）

- **步骤4**：设计并构建控制器。
  - 每个视图都有一个控制器，整个系统有多个控制器。每个**视图控制器**接收包含**UI指令的事件**，解释这些指令，并**将控制信息传送到与其交互的视图中**。
  - 控制器构件在初始化期间链接到一个模型和视图上。同时，控制器作为订阅者对需要控制的数据进行订阅。

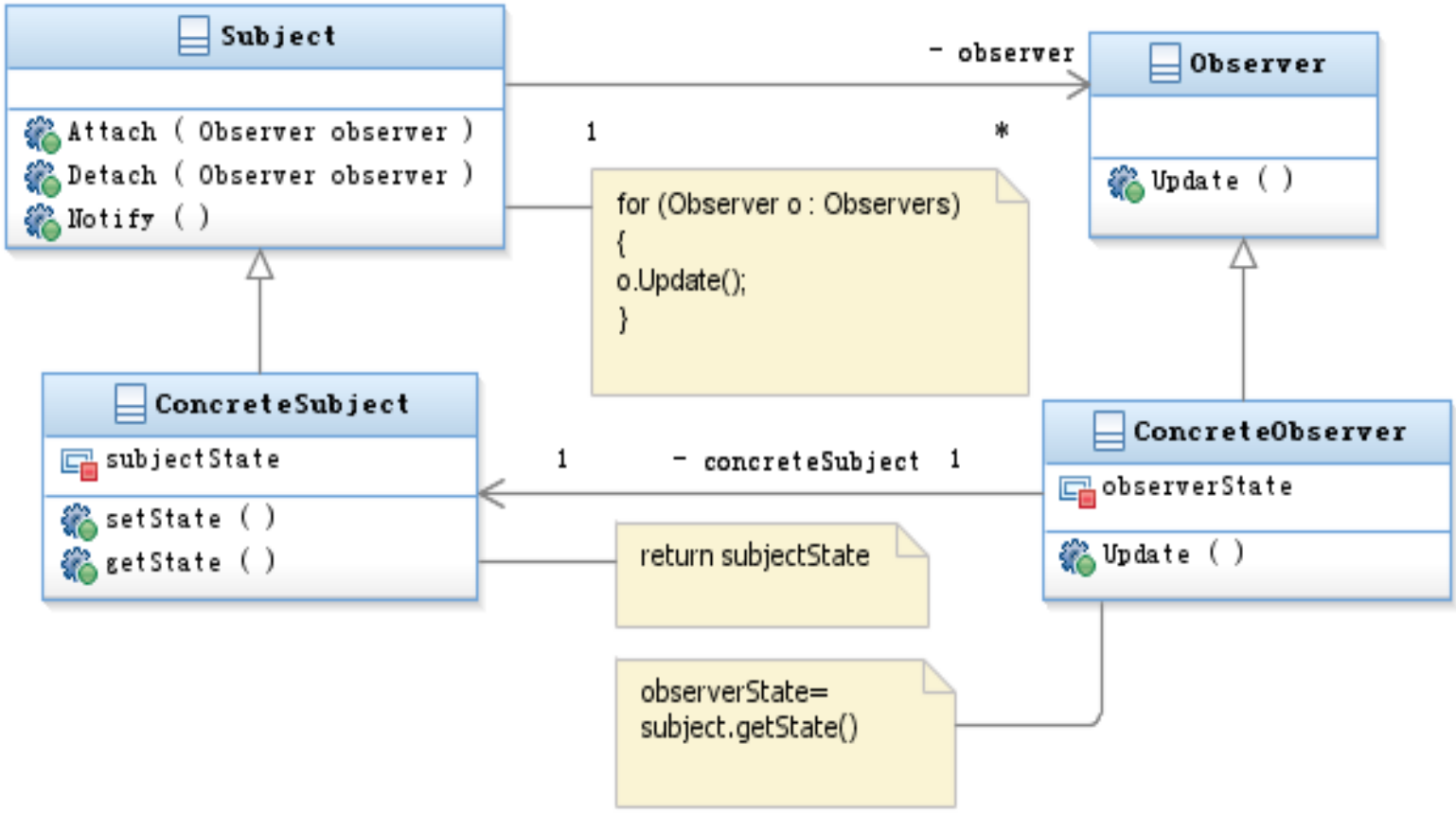
MVC模式实现机制（5）

- **步骤5**：建立视图和控制器之间的关系。
  - 每个视图在初始化期间都需要建立与其对应的控制器之间的关系。
  - 在视图类中，如果初始化代码中未创建控制器，应当定义一个makeController()方法来显式创建。

MVC模式实现机制（6）

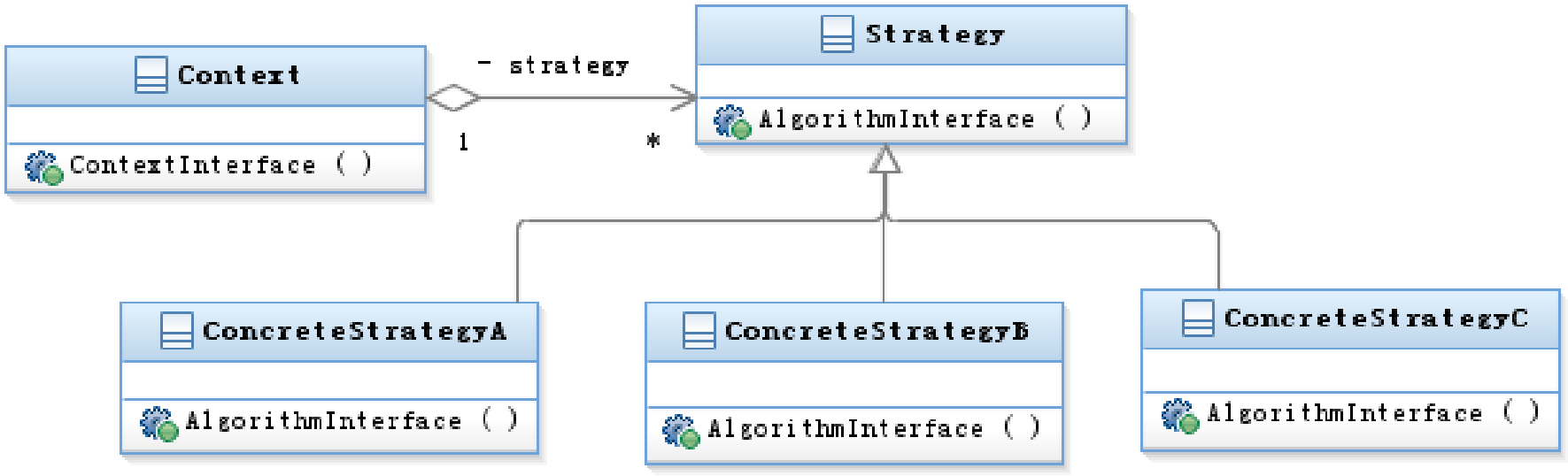
- **步骤6**：启动MVC。
- 有了多个视图和控制器，接下来需要绑定所有的元素并启动它们。
- 这部分最好在一个外部空间实现，例如一个Main程序中。MVC中的控制器依赖于到达的事件，控制器响应这些事件以触发视图或模型发生改变。
- 启动MVC的一个重要细节是启动事件处理，但事件处理机制并非MVC模式中的明确部分。

MVC中的观察者模式


Subject
Attach (Observer observer)
Detach (Observer observer)
Notify ( )
- observer
1
*
Observer
Update ( )
for (Observer o : Observers)
{
o.Update();
}
ConcreteSubject
subjectState
setState ( )
getState ( )
1
- concreteSubject
1
ConcreteObserver
observerState
Update ( )
return subjectState
observerState=
subject.getState()

MVC中的策略模式

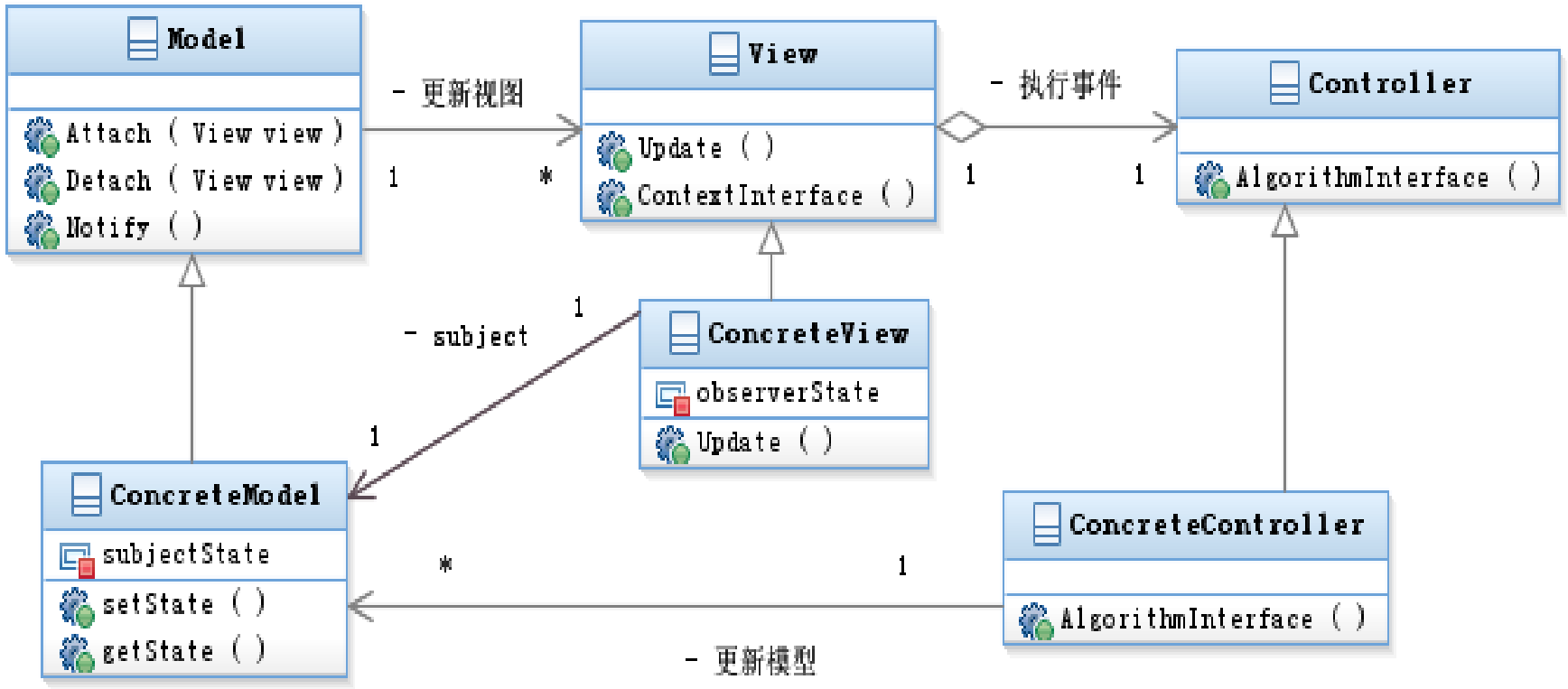
- 视图-控制器



MVC中的策略模式

```
public abstract class Strategy {
    public abstract void AlgorithmInterface();
}
public class ConcreteStrategyA extends Strategy {
    @Override
    public void AlgorithmInterface() {
        // code here
    }
}
public class Context {
    private Strategy Strategy;
    public Context(Strategy strategy)
    {
        Strategy = strategy;
    }
    public void ContextInterface()
    {
        Strategy.AlgorithmInterface();
    }
}
```

完整的MVC模式


Model
Attach (View view)
Detach (View view)
Notify ( )
- 更新视图
1
*
View
Update ( )
ContextInterface ( )
- 执行事件
1
1
Controller
AlgorithmInterface ( )
- subject
1
1
ConcreteView
observerState
Update ( )
ConcreteModel
subjectState
setState ( )
getState ( )
*
1
ConcreteController
AlgorithmInterface ( )
- 更新模型

3.8.2 MVP

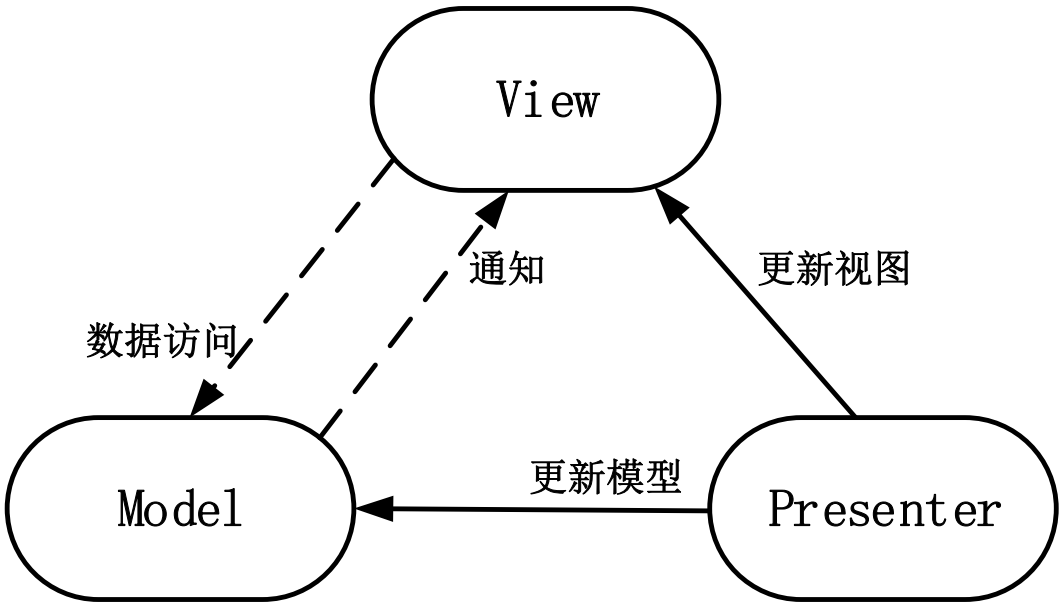
Model-View-Presenter (MVP)

MVP

- 1996年Taligent(IBM)公司的**Potel**首次提出并描述了MVP模式。Potel研究MVP时质疑了在MVC中是否需要控制器，他注意到现代操作系统的用户界面在视图中提供了大多数控制器功能，因此控制器看起来有点儿多余。

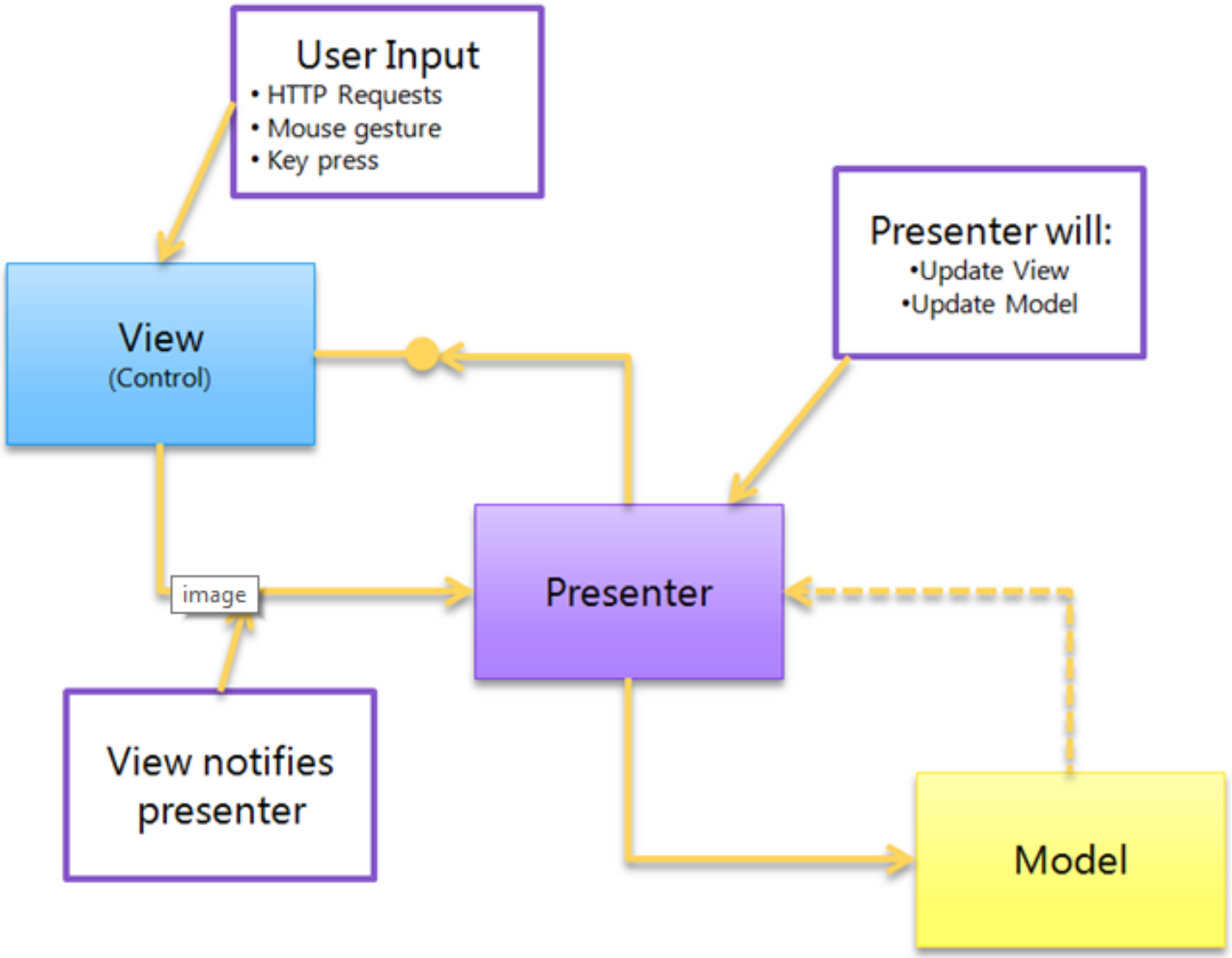
MVP工作原理

- MVP的基本思想是一种“变种MVC”，视图吸收控制器功能，增加一个主持者层（Presenter），主持者P能够直接访问视图和模型，模型-视图之间的关系仍然存在，但视图并不直接使用模型，所有的交互都发生在Presenter内部。



简化的MVP模型

MVP模式中各个构件的作用



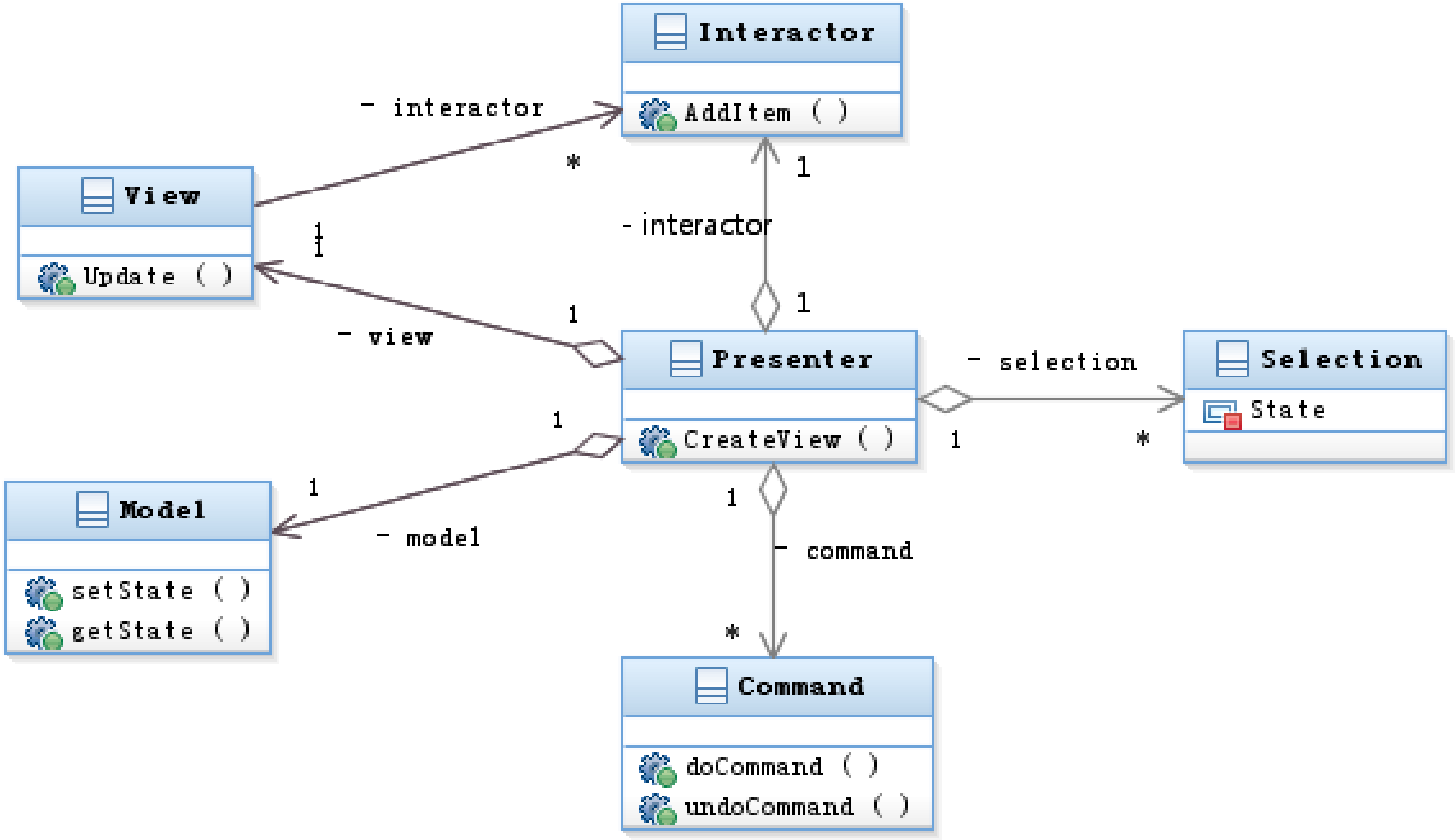
MVP模式中各个构件的作用

- 模型：类似于MVC模式中的模型
- 视图：通常是一个用户控件或组合了几个小控件的用户接口的一部分。用户可以在视图中与控件交互，但当要执行一些逻辑时，视图将其委派给主持者。
- 主持者：控制着视图的所有执行逻辑，还负责模型和视图之间的同步。当用户完成了一些操作（如按下按钮）时，由视图通知主持者，然后由主持者更新模型并同步模型和视图之间的变化。

MVP实现

- 模型与视图之间的交互类型，他把用户行为分类为选择、执行命令和触发事件。因此他定义了Selection类和Command类，正如类的名称一样，可以选择细化的模型、执行操作，并且引入了Interactor类（注意与Iterator“迭代器”区分），封装了可以改变数据的事件

MVP实现



来源：http://www.codeproject.com/Articles/42830/Model-View-Controller-Model-View-Presenter-and-Mod

3.8.3 MVVM

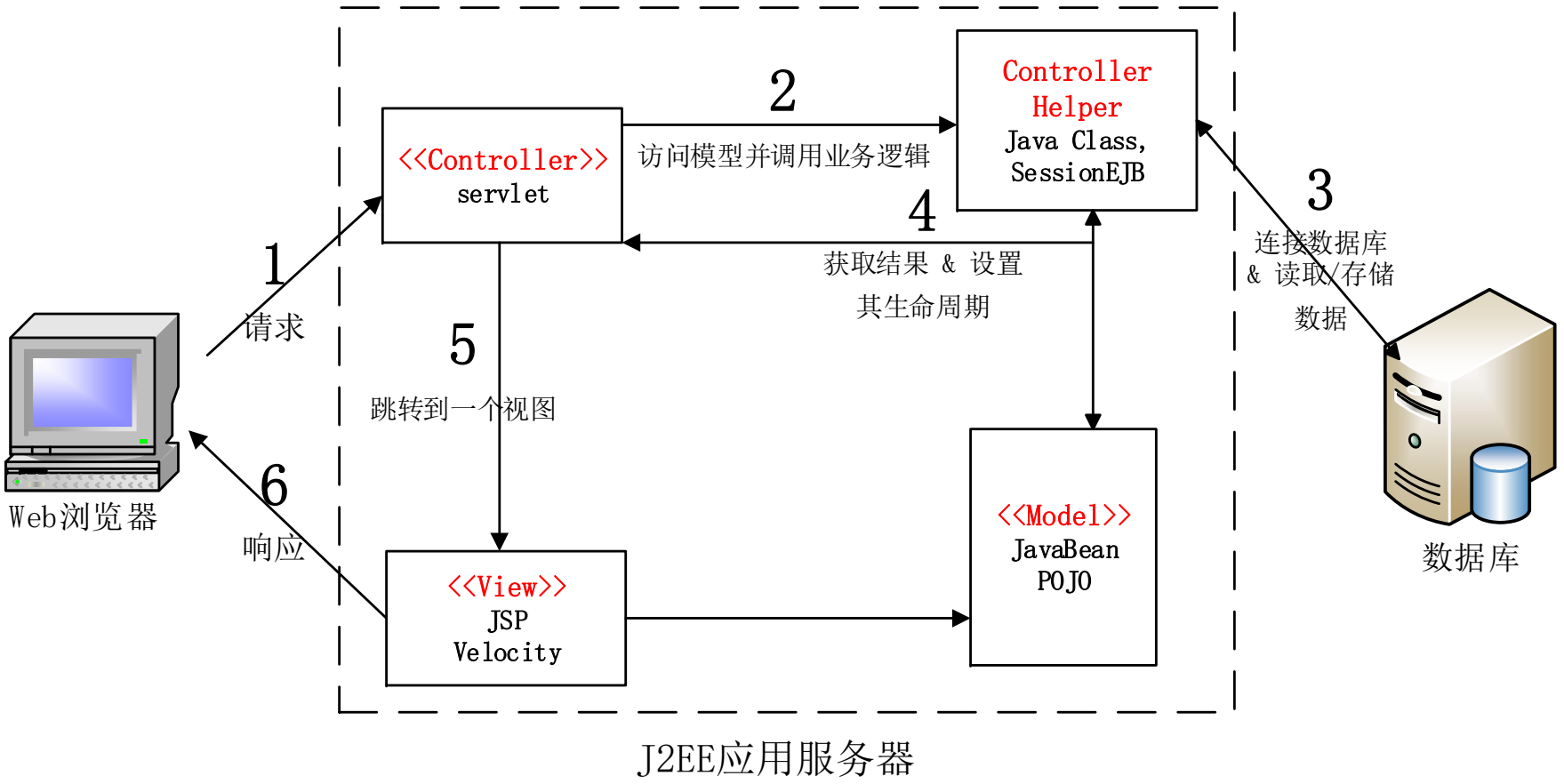
Model-View-ViewModel(MVVM)

2005年WPF架构师John Gossman在他的博客中首次提出Model-View-ViewModel

https://msdn.microsoft.com/en-us/magazine/dd419663.aspx

3.8.4 案例分析

案例1：Java Web程序中的MVC架构



案例2：一个简单的Java MVC架构小程序

3.8.5 优缺点分析

优点

- MVC架构最主要的优点是灵活的将数据（模型）从输出（视图）和输入（控制器）中解耦出来。首先，多个视图能共享一个模型，同一个模型可以被不同的视图重用，大大提高了代码的可重用性。
- 由于MVC的三个模块相互独立，改变其中一个不会影响其他两个，所以依据这种设计思想能构造良好的松耦合的构件。
- 此外，控制器提高了应用程序的灵活性和可配置性。控制器可以用来联接不同的模型和视图去完成用户的需求，这样控制器可以为构造应用程序提供强有力的手段。

缺点

- 需要构建并维护更多的构件，很大程度上增加了系统复杂性
- 控制器和视图组件对模型过于了解。模型的改变也许会同时需要控制器和视图的改变。
- 由于视图和模型数据的分离，并且必须使用模型的API，可能会产生低效的数据访问。

作 业

- 1.上机调试课堂上给出的Java MVC架构小程序，并试着将其改为MVP模式，分析MVC和MVP各自的优缺点和适用场合。
- 参考资料：
  - https://blogs.msdn.microsoft.com/erwinvandervalk/2009/08/14/the-difference-between-model-view-viewmodel-and-other-separated-presentation-patterns/
  - http://www.codeproject.com/Articles/42830/Model-View-Controller-Model-View-Presenter-and-Mod