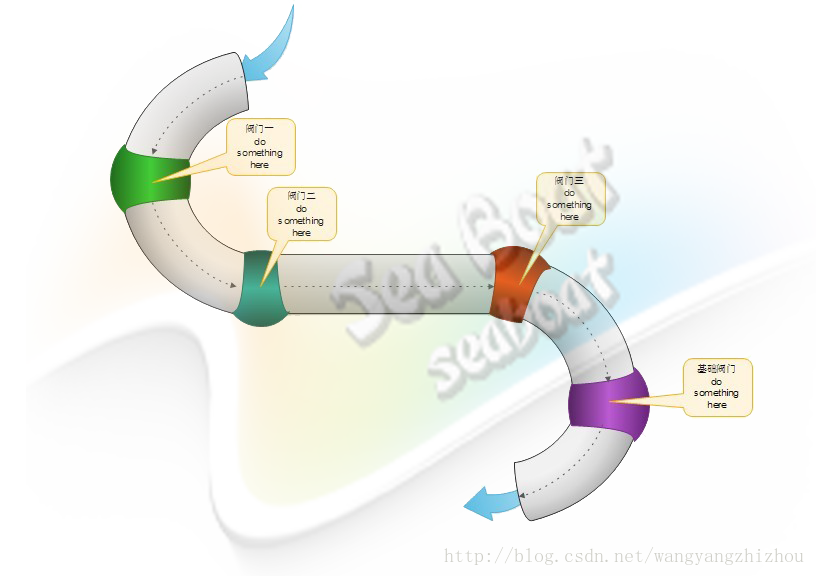
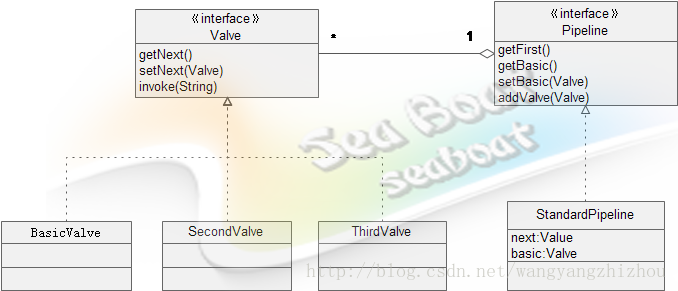
在一个比较复杂的大型系统中，假如存在某个对象或数据流需要被进行繁杂的逻辑处理的话，我们可以选择在一个大的组件中进行这些繁杂的逻辑处理，这种方式确实达到了目的，但却是简单粗暴的。或许在某些情况这种简单粗暴的方式将带来一些麻烦，例如我要改动其中某部分处理逻辑、我要添加一些处理逻辑到流程、我要在流程中减少一些处理逻辑时，这里有些看似简单的改动都让我们无从下手，除了对整个组件进行改动。整个系统看起来没有任何可扩展性和可重用性。  
 是否有一种模式可以将整个处理流程进行详细划分，划分出的每个小模块互相独立且各自负责一段逻辑处理，这些逻辑处理小模块根据顺序连起来，前以模块的输出作为后一模块的输入，最后一个模块的输出为最终的处理结果。如此一来修改逻辑时只针对某个模块修改，添加或减少处理逻辑也可细化到某个模块颗粒度，并且每个模块可重复利用，可重用性大大增强。这种模式就是此章节要进行讨论的管道模式。  
 顾名思义，管道模式就像一条管道把多个对象连接起来，整体看起来就像若干个阀门嵌套在管道中，而处理逻辑就放在阀门上，如下图，需要处理的对象进入管道后，分别经过阀门一、阀门二、阀门三、阀门四，每个阀门都会对进入的对象进行一些逻辑处理，经过一层层的处理后从管道尾处理，此时的对象就是已完成处理的目标对象。

  
既然管道模式这么有用，我们希望能在程序中适当地考虑使用，为了实现此模式需要多个对象协作，可参考如下类图，Valve接口定义了阀门的调用方法，由于阀门与阀门使用单链表结构连接所以需提供对next的操作，实现一个阀门对其进行扩展即可；Pipeline接口定义了管道操作阀门的方法，包括获取第一个阀门、获取基础阀门、添加阀门等方法，管道需对其扩展。  
   
往下看如何简单实现一个管道模式：  
① 阀门接口  
public interface Valve {  
  public Valve getNext();  
  public void setNext(Valve valve);  
  public void invoke(String handling);  
}  
② 管道接口  
public interface Pipeline {  
public Valve getFirst();  
public Valve getBasic();  
public void setBasic(Valve valve);  
public void addValve(Valve valve);  
}  
③ 基础阀门，处理逻辑仅仅是简单的将传入的字符串中”aa”替换成”bb”  
public class BasicValve implements Valve {  
protected Valve next = null;   
public Valve getNext() {  
return next;  
}  
public void invoke(String handling) {  
        handling=handling.replaceAll("aa", "bb");  
System.out.println("基础阀门处理完后：" + handling);  
}  
public void setNext(Valve valve) {  
this.next = valve;  
}  
}  
④ 第二个阀门，将传入的字符串中”11”替换成”22”  
public class SecondValve implements Valve {  
protected Valve next = null;   
public Valve getNext() {  
return next;  
}  
public void invoke(String handling) {  
handling = handling.replaceAll("11", "22");  
System.out.println("Second阀门处理完后：" + handling);  
getNext().invoke(handling);  
}  
public void setNext(Valve valve) {  
this.next = valve;  
}  
}  
⑤ 第三个阀门，将传入的字符串中”zz”替换成”yy”  
public class ThirdValve implements Valve {  
protected Valve next = null;   
public Valve getNext() {  
return next;  
}  
public void invoke(String handling) {  
handling = handling.replaceAll("zz", "yy");  
System.out.println("Third阀门处理完后：" + handling);  
getNext().invoke(handling);  
}  
public void setNext(Valve valve) {  
this.next = valve;  
}  
}  
⑥ 管道，我们一般的操作是先通过setBasic设置基础阀门，接着按顺序添加其他阀门，执行时的顺序是：先添加进来的先执行，最后才执行基础阀门。  
public class StandardPipeline implements Pipeline {  
protected Valve first = null;   
protected Valve basic = null;   
public void addValve(Valve valve) {  
if (first == null) {  
first = valve;  
valve.setNext(basic);  
} else {  
Valve current = first;  
while (current != null) {  
if (current.getNext() == basic) {  
current.setNext(valve);  
valve.setNext(basic);  
break;  
}  
current = current.getNext();  
}  
}  
}  
public Valve getBasic() {  
return basic;  
}  
public Valve getFirst() {  
return first;  
}  
public void setBasic(Valve valve) {  
this.basic = valve;  
}  
}  
⑦ 测试类  
public class Main {  
public static void main(String[] args) {  
String handling="aabb1122zzyy";  
StandardPipeline pipeline = new StandardPipeline();  
BasicValve basicValve = new BasicValve();  
SecondValve secondValve = new SecondValve();  
ThirdValve thirdValve = new ThirdValve();  
pipeline.setBasic(basicValve);  
pipeline.addValve(secondValve);  
pipeline.addValve(thirdValve);  
pipeline.getFirst().invoke(handling);  
}  
}  
输出的结果如下：  
Second阀门处理完后：aabb2222zzyy  
Third阀门处理完后：aabb2222yyyy  
基础阀门处理完后：bbbb2222yyyy  
这就是管道模式，在管道中连接一个或多个阀门，每个阀门负责一部分逻辑处理，数据按规定的顺序往下流。此模式分解了逻辑处理任务，可方便对某任务单元进行安装拆卸，提高了流程的可扩展性、可重用性、机动性、灵活性。