* 有一个数据列表DataList，其基本类型是3维向量ThreeD<x,y,z>
* 1）用Java语言实现该数据列表的数据结构
* 假设有三个外部对象A,B,C，分别对其x、y、z维度感兴趣，希望访问DataList在相应维度的数据并进行处理
* 2）请定义其对外的数据接口
  + 分别使用迭代器与Proxy

(这里可以再做一层继承，这个就可以实现多种迭代器了)

Class DataList{

Set<ThreeD<x,y,z> list;

Iterator createIterator(){

Return new RealIterator(list);

}

}

Iterator{

Boolean hasnext();

ThreeD<x,y,z> next();

}

RealIterator implement Iterator{

Set<ThreeD<x,y,z> list;

RealIterator(Set<ThreeD<x,y,z> list){

This.list=list;

}

Boolean hasNext(){…}

ThreeD<x,y,z> next(){..}

}

IteratorX {

Iterator iterator;

IteratorX (Iterator iterator){

This.itreator=iterator;

}

hasNext(){

iterator.hasnext();

}

X next(){.

Return iterator.next.getX();

}

}

Main{

Iterator iterator=new DataList().createiterator();

IteratorX xt=new IteratorX(iterator);

…. 可以访问了。

}

DataListImpl2 实现的所有了 hasNext,next()

Proxy 调用 dataListimpl2 的东东，访问next 中的x,y,z

Proxy:

Public proxyX{

DataList subject;

Public Object getNextX(){

Return subject.hasNext()==false?null:subject.next().getX();

}

}

Public proxyY{

DataList subject;

Public Object getNextY (){

Return subject.hasNext()==false?null:subject.next().getY ();

}

}

Z 类似。

Mediator proxy

* 现在有两个类：
  + C1：方法f1调用C2的f2;方法p1调用C2的p2
  + C2:有方法f2和p2
* 现在C1被部署到网络节点1，C2被部署到网络节点2
* 请写程序解决C1与C2之间的协作问题

这里使用remote proxy

C1 依赖于 proxyC1 -----ProxyC2 C2

* 用Mediator Pattern 模拟实现麻将游戏的框架
  + 四人，其中一个为庄家
  + 顺时针游戏顺序
  + 只模拟摸牌、出牌、吃牌、碰牌、槓牌和胡牌行为
  + 不要求详细的行为判断规则
* 使用Bridge Pattern实现一个画图程序的主体框架
  + 图形：点、线、矩形、圆…
  + 线形：实线、虚线、双线…
  + 线粗细：1x、2x、4x
  + 颜色：RGB

实现与抽象分开

* 请给出Facade模式的结构；
* 比较facade与collaboration design的异同

Collaboration有三种方式，一种是委托，一种是集成式，一种式分散式。Façade 类似于collaboration 中的controller,不同的是，façade 更多关注的是区别对待，而controller 主要是为了决策。

* 比较Facade. Mediator,在collaboration design方面异同：

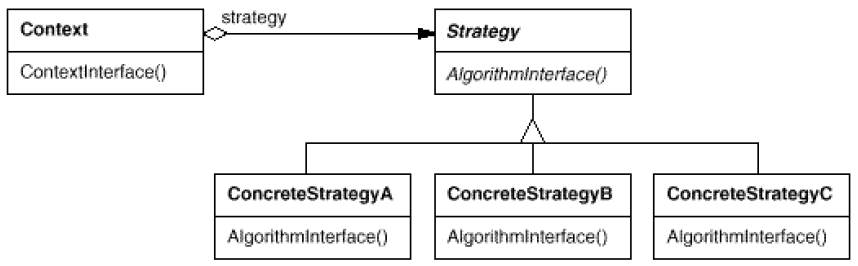
Façade mediator collaboration design 目标都是解耦，方式与强调的方面不同。

Façade 是将外界与内界之间进行解耦，而mediator 强调的是系统内部的。

sub system components are not aware of the facade, where as components are obviously aware of the mediator. The **facade** only exposes the existing functionality from a different perspective.The **mediator** "adds" functionality because it combines different existing functionality to create a new one.

collaboration design 强调的是决策。

* 在策略模式中， 为什么使用“聚合”而不是 “关联”关系？



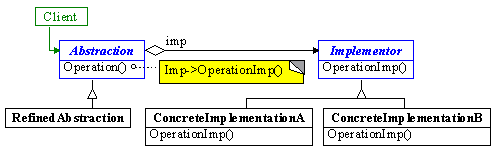
* 如果一个对象集之间除共性外，有超过２个的差异行为，如何处理？
* 如果一个对象集的一个行为需要协作对象来完成，但是它们的协作对象存在差异性，如何处理？
* 如果一个对象集的部分行为组存在差异性，如何处理？
* 如果一个对象集的部分属性（以及依赖于这些属性的方法）存在差异性，如何处理？

前四个都是将变化的部分移出去，采用策略模式。

* 如果一个对象集的行为因为属性的取值而存在差异性，如何处理？（采用状态模式）
* 从共性和差异性角度，如何解释Bridge模式？

将与其他对象连接的部分分离出去;每个行为都封装出去;保留接口，其他全部做成策略;

（Bridge非常典型的只有Interface）暴露给Client的是不变的接口，Implementor给出了一些共同操作的接口，但是Abstraction可以有不同的实现，实现也有不同的实现。接口是共性，而实现是差异性，利用了继承和聚合。



* 比较strategy 与 state
  + “1 of N” or “M of N”?

策略模式用在M of N 的情况下，状态模式也可以，但是通常用在1 of N 的情况下。

* + Who control the changing of variation?
    - “Context decide changing of ConcreteState object ”与“ ConcreteState decide changing of ConcreteState object”有何不同？

Front:谁控制修改是不固定的，谁拥有改变规则就控制修改。参考信息专家，将职责分配给拥有足够的信息去完成整个职责的类。 例如，状态模式：

Context变量放在哪个里面，哪个就可以控制修改。（下图举例标出了Context类型变量可放的位置）

Later: 在状态模式中，状态的变化是有对象的内部条件决定，外界只需关心其接口，不必关心其状态对象的创建和转化；而策略模式里面，采取何种策略由外部条件决定的。

其实，两种模式的关键不同在于，状态模式注重给客户对象提供在不同状态间切换不同的行为。重在切换；而策略模式注重给客户对象提供多种不同的选择，一般来说，用户不会经常切换来切换去。

前者是指用context来操作不同状态类的自动转换，或者是指用具体某个状态的某个event来自动实现各个状态之间的跳转。

在现实世界中，策略（如促销一种商品的策略）和状态（如同一个按钮来控制一个电梯的状态，又如手机界面中一个按钮来控制手机）是两种完全不同的思想。当我们对状态和策略进行建模时，这种差异会导致完全不同的问题。例如，对状态进行建模时，状态迁移是一个核心内容；然而，在选择策略时，迁移与此毫无关系。另外，策略模式允许一个客户选择或提供一种策略，而这种思想在状态模式中完全没有

* + How to change?
    - Fixed rules or configuration files

按实际情况，如果需求可以固定为固定的规则，则用fixed rules， 如果是外部控制需求改变，则是table-driven

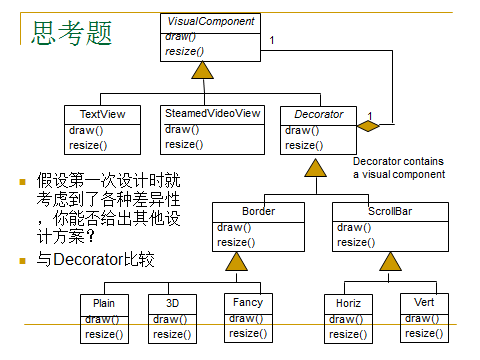
Creating and destroying policy?视具体情况而定，一般是在ref那里创建

如果有多个其他类实例的创建类型都需要子类来决定怎么办？

使用多个工厂方法，有一个类实例就建一个。

如果多个其他类实例之间存在类型依赖该怎么办？

使用抽象工厂模式。



直接继承。因为考虑了所有的差异性了。