1. 设计模式部分所有的思考题
   1. 寻找一个需要访问集合变量的简单场景： 编程实现: Iterator Pattern and Proxy Pattern

//学生打印出所有的答案

interface IStudent

{

void printAnswer();

}

//集合类

interface IAggregate{

public Iterator createIterator();

}

//代理模式中的realsubject

class Student implements IStudent{

Answers ans;

public void printAnswer() {

String[] str={"a","b","c"};

ans=new Answers(str);

visit(ans.createIterator());

}

public void visit(Iterator iterator){

while(iterator.hasNext())

System.out.println(iterator.next());

}

}

//学生代理

class StudentProxy implements IStudent{

Student st;

public void printAnswer() {

System.out.println("do something before invoke");

if(st==null)

st=new Student();

st.printAnswer();

}

}

//答案的枚举类

class AnswerIterator implements Iterator{

Answers ans;

int index=0;

AnswerIterator(Answers ans){

this.ans=ans;

}

public boolean hasNext() {

if(index==3)

return false;

else return true;

}

public Object next() {

String an=(String)ans.answers[index];

index++;

return an;

}

public void remove() {

}

}

//答案集合类，实现生成答案迭代器的方法

class Answers implements IAggregate{

String[] answers;

Answers(String[] answers){

this.answers=answers;

}

public Iterator createIterator() {

return new AnswerIterator(this);

}

}

//客户要求打印答案

class Client {

public static void main(String[] args){

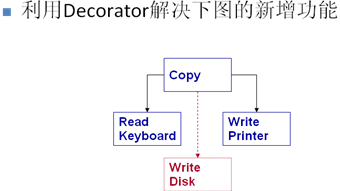
IStudent sp=new StudentProxy();

sp.printAnswer();

}

}

* 1. 用Mediator Pattern 实现 CardShark游戏
  2. 用Decorator 解决下图的新增功能



interface WritePrinter{

public void write(int c);

}

class DiskWriter implements WritePrinter{

PrinterWriter pw;

DiskWriter(PrinterWriter pw){

this.pw=pw;

}

public void write(int c) {

pw.write(c);

//disk write

System.out.println("新增：write disk");

}

}

class PrinterWriter implements WritePrinter{

public void write(int c) {

//printer write

}

}

public class Main {

public static void main(String[] args){

PrinterWriter pw=new PrinterWriter();

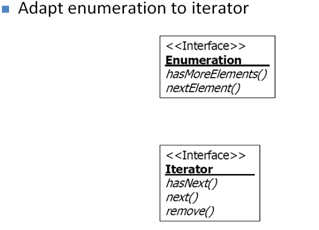
DiskWriter dw=new DiskWriter(pw);

dw.write(1);

}

}

d)



public class EnumerationIterator implements Iterator  
{  
  Enumeration enum;  
  public EnumerationIterator(Enumeration enum){  
     this.enum=enum;  
  }  
  public boolean hasNext(){  
     return enum.hasMoreElements();  
  }  
  public Object next(){  
     return enum.nextElement();  
  }  
  public void remove(){  
     throw new UnsupportedOperationException();  
  }  
}

* 1. 如果一个对象集的一个行为需要协作对象来完成，但是它们在协作对象上存在差异性，如何处理？

使用strategy模式，把这个行为抽象出来 //自己yy的

* 1. 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？
  2. 如果一个对象集除了接口之外，全是差异行为，如何处理？
  3. 从共性和差异性角度，如何解释Bridge模式？
  4. strategy 与 state
  5. 1 of N or M of N?
  6. Who control the changing?
  7. How to change?
     1. Fixed rules or table-driven (configuration files)
  8. Creating and destroying policy?
  9. ntext decide changing of ConcreteState object ”与“ ConcreteState decide changing of ConcreteState object”有何不同？
  10. 以singleton为基础，编写程序解决Limited instance permitted

class Instance {

private static Instance ins;

private static int count=0;

private static int limit=3;

private Instance(){ }

public static Instance getInstance(){

if(count<limit){

count++;

return new Instance();

}

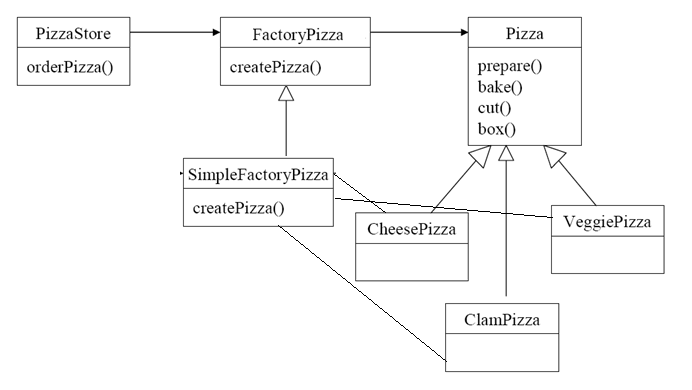
else return null;

}

}

* 1. 如果增加pizza类型t1,请基于Decorate模式完成对factory的修改

修改前的类图：



**interface Pizza**{

void prepare();

void bake();

void cut();

void box();

}

**interface FactoryPizza**{

Pizza createPizza(String type);

}

**class SimpleFactoryPizza** implements FactoryPizza{

public Pizza createPizza(String type){

Pizza pizza=null;

if(type=="cheesePizza")

pizza= new CheesePizza();

else if(type=="veggePizza")

pizza= new VeggePizza();

else if(type=="clamPizza")

pizza= new ClamPizza();

return pizza;

}

}

**class Decorator** implements FactoryPizza{

SimpleFactoryPizza simfac;

Decorator(SimpleFactoryPizza simfac){

this.simfac=simfac;

}

public Pizza createPizza(String type) {

Pizza pizza;

if(!type.equals("t1"))

pizza=simfac.createPizza(type);

else pizza=new T1Pizza();

return pizza;

}

}

**public class PizzaStore** {

public void orderPizza(String type){

SimpleFactoryPizza simfac=new SimpleFactoryPizza();

Decorator dec=new Decorator(simfac);

Pizza pizza=dec.createPizza(type);

pizza.prepare();

//...

}

public static void main(String[] args){

PizzaStore store=new PizzaStore();

store.orderPizza("t1");

}

}

**class CheesePizza** implements Pizza{

public void prepare() {}

public void bake() {}

public void cut() {}

public void box() {}

}

**class VeggePizza** implements Pizza{

public void prepare() {}

public void bake() {}

public void cut() {}

public void box() {}

}

**class ClamPizza** implements Pizza{

public void prepare() {}

public void bake() {}

public void cut() {}

public void box() {}

}

**class T1Pizza** implements Pizza{

public void prepare() {}

public void bake() {}

public void cut() {}

public void box() {}

}

* 1. 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？
  2. 如果多个其他类实例之间存在类型依赖该怎么办？