# 思考题:

## Design Pattern（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

假想一个场景，需要通过迭代器来访问元素的后继(Binary Tree)

BinaryTree 类实现基本的二叉树的插入查找等功能:

*01* **public** **class** BinaryTree {  
*02* **public** Node head;  
*03*  
*04* **public** Node findMinNode(Node node) {  
05 Node temp = node;  
*06* **if** (temp == **null**) {  
*07* **return** **null**;  
*08* }  
*09* **while** (temp.getLeftNode() != **null**) {  
10 temp = temp.getLeftNode();  
*11* }  
*12* **return** temp;  
*13* }  
*14*  
15 **public** Node findMaxNode(Node node) {  
*16* Node temp = node;  
*17* **if** (temp == **null**) {  
*18* **return** **null**;  
*19* }  
20 **while** (temp.getRightNode() != **null**) {  
*21* temp = temp.getRightNode();  
*22* }  
*23* **return** temp;  
*24* }  
25  
*26* **public** Node searchNode(**int** x) {  
*27* Node node = head;  
*28* Node node1 = head;  
*29* **if** (head == **null**)  
30 **return** **null**;  
*31* **while** (node1 != **null**) {  
*32* node = node1;  
*33* **if** (node.getElement() == x) {  
*34* **return** node;  
35 } **else** **if** (node.getElement() > x) {  
*36* node1 = node.getLeftNode();  
*37* } **else** {  
*38* node1 = node.getRightNode();  
*39* }  
40 }  
*41* **return** **null**;  
*42* }  
*43*  
*44* **public** Node successor(Node x) {  
45 **if** ((x == **null**) || (searchNode(x.getElement()) == **null**)) {  
*46* **return** **null**;  
*47* }  
*48* **if** (x.getRightNode() != **null**) {  
*49* **return** findMinNode(x.getRightNode());  
50 }  
*51* Node y = x.getFather();  
*52* **while** ((y != **null**) && (x == y.getRightNode())) {  
*53* x = y;  
*54* y = y.getFather();  
55 }  
*56* **return** y;  
*57* }  
*58*  
*59* **public** **void** insert(**int** element) {  
60 Node temp = head;  
*61* Node temp1 = head;  
*62* **while** (temp1 != **null**) {  
*63* temp = temp1;  
*64* **if** (temp.getElement() == element) {  
65 **return**;  
*66* } **else** **if** (temp.getElement() > element) {  
*67* temp1 = temp.getLeftNode();  
*68* } **else** {  
*69* temp1 = temp.getRightNode();  
70 }  
*71* }  
*72* **if** (temp == **null**) {  
*73* head = **new** Node();  
*74* head.setElement(element);  
75 head.setFather(**null**);  
*76* } **else** {  
*77* Node node = **new** Node();  
*78* node.setElement(element);  
*79* node.setFather(temp);  
80 **if** (temp.getElement() > element) {  
*81* temp.setLeftNode(node);  
*82* } **else** {  
*83* temp.setRightNode(node);  
*84* }  
85 }  
*86* };  
*87* }

Node节点类是基本元素

*01* **public** **class** Node {  
*02* **private** Node father;  
*03* **private** Node leftNode;  
*04* **private** Node rightNode;  
05 **private** **int** element;  
*06*  
*07* **public** Node getFather() {  
*08* **return** father;  
*09* }  
10  
*11* **public** **void** setFather(Node father) {  
*12* **this**.father = father;  
*13* }  
*14*  
15 **public** Node getLeftNode() {  
*16* **return** leftNode;  
*17* }  
*18*  
*19* **public** **void** setLeftNode(Node leftNode) {  
20 **this**.leftNode = leftNode;  
*21* }  
*22*  
*23* **public** Node getRightNode() {  
*24* **return** rightNode;  
25 }  
*26*  
*27* **public** **void** setRightNode(Node rightNode) {  
*28* **this**.rightNode = rightNode;  
*29* }  
30  
*31* **public** **int** getElement() {  
*32* **return** element;  
*33* }  
*34*  
35 **public** **void** setElement(**int** element) {  
*36* **this**.element = element;  
*37* }  
*38*  
*39* }

下面是迭代器，主要实现基本的访问功能

*01* **public** **class** BinaryTreeIterator {  
*02* Object index;  
*03* BinaryTree bt;  
*04*  
05 **public** BinaryTreeIterator(BinaryTree bt) {  
*06* **this**.bt = bt;  
*07* index = bt.findMinNode(bt.head);  
*08* }  
*09*  
10 **public** Object first() {  
*11* **return** bt.head;  
*12* }  
*13*  
*14* **public** **void** next() {  
15 index = bt.successor((Node) index);  
*16* }  
*17*  
*18* **public** **boolean** isDone() {  
*19* **return** index == **null**;  
20 }  
*21*  
*22* **public** Object currentItem() {  
*23* **return** index;  
*24* }  
25 }

代理模式，封装一部分操作

*01* **public** **class** BinaryTreeProxy {  
*02* **private** BinaryTree bt;  
*03*  
*04* **public** BinaryTreeProxy(BinaryTree bt) {  
05 **this**.bt = bt;  
*06* }  
*07*  
*08* **public** **void** insert(**int** element) {  
*09* bt.insert(element);  
10 }  
*11*  
*12* **public** Node searchNode(**int** x) {  
*13* **return** bt.searchNode(x);  
*14* }  
15  
*16* **public** Node findMaxNode(Node node) {  
*17* **if** (node == **null**) {  
*18* **return** **null**;  
*19* }  
20 **if** (searchNode(node.getElement()) == **null**) {  
*21* **return** **null**;  
*22* }  
*23* **return** bt.findMaxNode(node);  
*24* }  
25  
*26* **public** Node findMinNode(Node node) {  
*27* **if** (node == **null**) {  
*28* **return** **null**;  
*29* }  
30 **if** (searchNode(node.getElement()) == **null**) {  
*31* **return** **null**;  
*32* }  
*33* **return** bt.findMinNode(node);  
*34* }  
35  
*36* **public** Node successor(Node x) {  
*37* **if** (x == **null**) {  
*38* **return** **null**;  
*39* }  
40 **if** (searchNode(x.getElement()) == **null**) {  
*41* **return** **null**;  
*42* }  
*43* **return** bt.successor(x);  
*44* }  
45 }

Mian函数用来测试

*01* **public** **class** Main {  
*02*  
*03* */\*\**  
*04* *\* @param args*  
05 *\*/*  
*06* **public** **static** **void** main(String[] args) {  
*07* BinaryTree bt = **new** BinaryTree();  
*08* bt.insert(7);  
*09* bt.insert(10);  
10 bt.insert(9);  
*11* bt.insert(13);  
*12* bt.insert(5);  
*13* bt.insert(2);  
*14* bt.insert(6);  
15 BinaryTreeIterator bti = **new** BinaryTreeIterator(bt);  
*16* **for**(;!bti.isDone();bti.next()){  
*17* System.out.println(((Node)bti.currentItem()).getElement());  
*18* }  
*19* }  
20  
*21* }

### 用Mediator Pattern 实现 CardShark游戏（P467）

* 下面是调停者模式

所有的参与者(Colleague)都继承该类

*01* package CardShark;  
*02*  
*03* **public** **class** Colleague {  
*04* **private** Mediator mediator;  
05  
*06* **public** Colleague(Mediator mediator) {  
*07* **this**.mediator = mediator;  
*08* }  
*09*  
10 **public** Mediator getMediator() {  
*11* **return** mediator;  
*12* }  
*13*  
*14* **public** **void** setMediator(Mediator mediator) {  
15 **this**.mediator = mediator;  
*16* }  
*17* }

参与者有cardsender，player，container

*01* package CardShark;  
*02*  
*03* **public** **class** Colleague {  
*04* **private** Mediator mediator;  
05  
*06* **public** Colleague(Mediator mediator) {  
*07* **this**.mediator = mediator;  
*08* }  
*09*  
10 **public** Mediator getMediator() {  
*11* **return** mediator;  
*12* }  
*13*  
*14* **public** **void** setMediator(Mediator mediator) {  
15 **this**.mediator = mediator;  
*16* }  
*17* }

*001* package CardShark;  
*002*  
*003* import java.io.BufferedReader;  
*004* import java.io.IOException;  
005 import java.io.InputStreamReader;  
*006*  
*007* **public** **class** Player **extends** Colleague {  
*008* String name;  
*009* status status;  
010 **double** stack;  
*011*  
*012* **public** Player(Mediator mediator) {  
*013* **super**(mediator);  
*014* status = status.Null;  
015 stack = 0;  
*016* }  
*017*  
*018* **public** **void** getStackBegin() {  
*019* status = status.PutStack;  
020 System.out.println(“玩家 “ + name + ” 获知开始下注”);  
*021* **this**.setStack();  
*022* **this**.noticeNextPlayerStack();  
*023* }  
*024*  
025 **public** **void** setStack() {  
*026* System.out  
*027* .print(“Hi “ + name + “,you should enter the Amount of fees:”);  
*028* BufferedReader bf = **new** BufferedReader(**new** InputStreamReader(System.in));  
*029* **try** {  
030 **double** stack = Double.parseDouble(bf.readLine());  
*031* **super**.getMediator().addAmount(stack);  
*032* System.out.println(“玩家 “ + name + “开始下注”);  
*033* **this**.stack = stack;  
*034* } **catch** (NumberFormatException e) {  
035 e.printStackTrace();  
*036* } **catch** (IOException e) {  
*037* e.printStackTrace();  
*038* }  
*039* }  
040  
*041* **public** **void** noticeNextPlayerStack() {  
*042* System.out.println(“玩家 “ + name + “通知下一个下注的玩家”);  
*043* **super**.getMediator().noticeNextPlayerStackByPlayer();  
*044* }  
045  
*046* **public** **void** getGamblingBegin(Player player, **double** total) {  
*047* System.out.println(“玩家 “ + name + “接到通知开始赌博”);  
*048* status = status.Gambling;  
*049* **if** (**this**.isLastOne()) {  
050 **this**.finished(player, total);  
*051* } **else** {  
*052* **this**.chooseOptioins(player, total);  
*053* }  
*054* }  
055  
*056* **public** **void** chooseOptioins(Player player, **double** total) {  
*057* String name = “暂无人下注”;  
*058* **if** (player != **null**)  
*059* name = player.name;  
060 System.out  
*061* .print(“The last add person is “  
*062* + name  
*063* + ” and the total number is “  
*064* + total  
065 + “. You should choose one option among add, withdraw or not set by 1,2,3(default 3)”);  
*066* BufferedReader bf = **new** BufferedReader(**new** InputStreamReader(System.in));  
*067* **try** {  
*068* **int** choice = Integer.parseInt(bf.readLine());  
*069* **switch** (choice) {  
070 **case** 1:  
*071* System.out.print(“input the amount:”);  
*072* **double** moreStack = Double.parseDouble(bf.readLine());  
*073* **this**.addAmount(moreStack);  
*074* **this**.noticeNextGamblingPlayer();  
075 **break**;  
*076* **case** 2:  
*077* **this**.withdraw();  
*078* **break**;  
*079* **default**:  
080 **this**.notSet();  
*081* **this**.noticeNextGamblingPlayer();  
*082* **break**;  
*083* }  
*084* } **catch** (NumberFormatException e) {  
085 e.printStackTrace();  
*086* } **catch** (IOException e) {  
*087* e.printStackTrace();  
*088* }  
*089*  
090 }  
*091*  
*092* **private** **void** addAmount(**double** moreStack) {  
*093* **this**.stack += moreStack;  
*094* **super**.getMediator().addStack(**this**);  
095 }  
*096*  
*097* **private** **void** withdraw() {  
*098* **super**.getMediator().withdraw(**this**);  
*099* }  
100  
*101* **private** **void** notSet() {  
*102* **super**.getMediator().notSet(**this**);  
*103* }  
*104*  
105 **public** **void** noticeNextGamblingPlayer() {  
*106* **super**.getMediator().noticeNextGamblingPlayerByPlayer();  
*107* }  
*108*  
*109* **public** **void** sendSecondPlayerByPlayer() {  
110 **super**.getMediator().noticeSecondGamblingPlayer();  
*111* }  
*112*  
*113* **public** **boolean** isLastOne() {  
*114* **return** **super**.getMediator().isLastOne(**this**);  
115 }  
*116*  
*117* **public** **void** finished(Player player, **double** total) {  
*118* String name = “暂无人下注”;  
*119* **if** (player != **null**)  
120 name = player.name;  
*121* System.out.print(“赌博结束，最后下注者为  “ + name  
*122* + ” 总数为 “ + total);  
*123*  
*124* }  
125 }

*01* package CardShark;  
*02*  
*03* **public** **class** Container **extends** Colleague {  
*04* status status;  
05 Player lastAddMan;  
*06* **double** totalStack;  
*07*  
*08* **public** Container(Mediator mediator) {  
*09* **super**(mediator);  
10 **this**.status = status.Null;  
*11* lastAddMan = **null**;  
*12* }  
*13*  
*14* **public** **void** startStack() {  
15 status = status.PutStack;  
*16* totalStack = 0;  
*17* System.out.println(“容器获得开始下注信息”);  
*18* }  
*19*  
20 *//开始下注或者开始加注*  
*21* **public** **void** addStack(**double** adds){  
*22* totalStack+=adds;  
*23* }  
*24*  
25 **public** **void** startGambling() {  
*26* status = status.Gambling;  
*27* System.out.println(“容器获得开始下赌博信息”);  
*28* }  
*29*  
30 **public** status getStatus() {  
*31* **return** status;  
*32* }  
*33*  
*34* **public** **void** setStatus(status status) {  
35 **this**.status = status;  
*36* }  
*37*  
*38* **public** Player getLastAddMan() {  
*39* **return** lastAddMan;  
40 }  
*41*  
*42* **public** **void** setLastAddMan(Player lastAddMan) {  
*43* **this**.lastAddMan = lastAddMan;  
*44* }  
45  
*46* **public** **double** getTotalStack() {  
*47* **return** totalStack;  
*48* }  
*49*  
50 **public** **void** setTotalStack(**double** totalStack) {  
*51* **this**.totalStack = totalStack;  
*52* }  
*53*  
*54* }

*01* package CardShark;  
*02*  
*03* **public** **class** CardSender **extends** Colleague {  
*04*  
05 **public** CardSender(Mediator mediator) {  
*06* **super**(mediator);  
*07* }  
*08*  
*09* **public** **void** startStack(){  
10 **this**.noticeStartStack();  
*11* **this**.noticeNextPlayerStack();  
*12* }  
*13*  
*14* *// 通知开发下注*  
15 **public** **void** noticeStartStack() {  
*16* System.out.println(“发牌人通知下注开始”);  
*17* **super**.getMediator().noticeBeginStack();  
*18* }  
*19*  
20 *//通知下一个玩家下注*  
*21* **public** **void** noticeNextPlayerStack() {  
*22* System.out.println(“发牌人通知下一个玩家开始下注”);  
*23* **super**.getMediator().noticeNextPlayerStackByCardSender();  
*24* }  
25  
*26* **public** **void** GetStackInfoFromUser(){  
*27* System.out.println(“发牌人受到下注信息，下注停止”);  
*28* }  
*29*  
30 **public** **void** startGambling(){  
*31* **this**.noticeStartGambling();  
*32* **this**.noticeNextPlayerGamblingBySender();  
*33* }  
*34*  
35 **public** **void** noticeStartGambling() {  
*36* System.out.println(“发牌人通知容器赌博开始”);  
*37* **super**.getMediator().noticeBeginGambling();  
*38* }  
*39*  
40 **public** **void** noticeNextPlayerGamblingBySender() {  
*41* System.out.println(“发牌人通知下一个玩家开始赌博开始”);  
*42* **super**.getMediator().noticeBeginGamblingBySender();  
*43* }  
*44*  
45 **public** **void** noticeSecondPlayerGamblingBySender(Player player){  
*46* System.out.println(“发牌人收到”+ player.name+“通知将信息发给该用户左边的用户”);  
*47* **super**.getMediator().noticeSecondPlayerGamblingBySender();  
*48* }  
*49* }

容器的状态类

*1* package CardShark;  
*2*  
*3* **public** **enum** status {  
*4* Null,PutStack,Gambling,End  
5 }

这个是调停者(Mediator)，主要是集中式控制

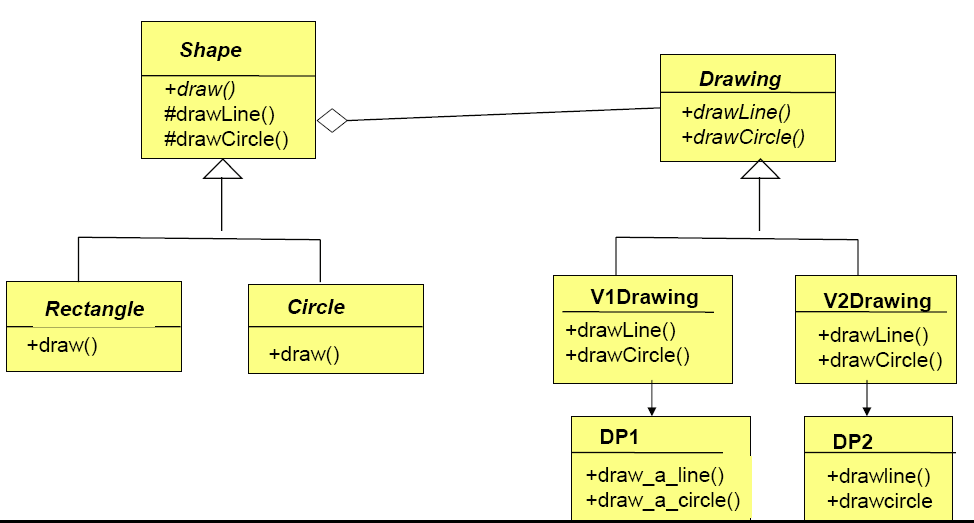
*001* package CardShark;  
*002*  
*003* import java.util.ArrayList;  
*004* import java.util.Iterator;  
005  
*006* **public** **class** Mediator {  
*007* CardSender cardSender;  
*008* ArrayList<Player> players;  
*009* Container container;  
010 Iterator<Player> iterator;  
*011*  
*012* **public** Mediator() {  
*013* players = **new** ArrayList<Player>();  
*014* }  
015  
*016* **public** **void** registerPlayer(Player player) {  
*017* **this**.players.add(player);  
*018* }  
*019*  
020 **public** **void** registerContainer(Container container) {  
*021* **this**.container = container;  
*022* }  
*023*  
*024* **public** **void** registerCardSender(CardSender cardSender) {  
025 **this**.cardSender = cardSender;  
*026* }  
*027*  
*028* **public** **void** noticeBeginStack() {  
*029* iterator = players.iterator();  
030 container.startStack();  
*031* }  
*032*  
*033* *// 由发牌人通知下一个玩家开始下注*  
*034* **public** **void** noticeNextPlayerStackByCardSender() {  
035 (iterator.next()).getStackBegin();  
*036* }  
*037*  
*038* *// 增加下注数量和下注金额*  
*039* **public** **void** addAmount(**double** amount) {  
040 container.addStack(amount);  
*041* }  
*042*  
*043* *// 由参与者通知下一个玩家开始下注*  
*044* **public** **void** noticeNextPlayerStackByPlayer() {  
045 **if** (iterator.hasNext()) {  
*046* (iterator.next()).getStackBegin();  
*047* } **else** {  
*048* cardSender.GetStackInfoFromUser();  
*049* }  
050 }  
*051*  
*052* **public** **void** noticeBeginGambling() {  
*053* iterator = players.iterator();  
*054* container.startGambling();  
055 }  
*056*  
*057* **public** **void** noticeBeginGamblingBySender() {  
*058* (iterator.next()).getGamblingBegin(**null**, container.totalStack);  
*059* }  
060  
*061* **public** **void** addStack(Player player) {  
*062* container.setLastAddMan(player);  
*063* container.addStack(player.stack);  
*064* }  
065  
*066* **public** **void** withdraw(Player player) {  
*067* **if**(isLastOne(player)){  
*068* player.finished(container.lastAddMan,container.totalStack);  
*069* }**else** **if**(isFirstOne(player)){  
070 cardSender.noticeSecondPlayerGamblingBySender(player);  
*071* }**else**{  
*072* Iterator<Player> temp = players.iterator();  
*073* Player previousOne = temp.next();  
*074* Player currentOne = previousOne;  
075 **while**(temp.hasNext()){  
*076* previousOne = currentOne;  
*077* currentOne = temp.next();  
*078* **if**(currentOne == player){  
*079* **break**;  
080 }  
*081* }  
*082* iterator = players.iterator();  
*083* **while**(iterator.hasNext()){  
*084* **if**(iterator.next()==previousOne){  
085 **break**;  
*086* }  
*087* }  
*088* previousOne.sendSecondPlayerByPlayer();  
*089* }  
090 }  
*091*  
*092* **public** **void** noticeSecondPlayerGamblingBySender(){  
*093* iterator.next().getGamblingBegin(**null**,container.totalStack);  
*094* }  
095  
*096* **public** **boolean** isFirstOne(Player player){  
*097* Iterator<Player> temp = players.iterator();  
*098* **return** temp.next()== player;  
*099* }  
100  
*101* **public** **boolean** isLastOne(Player player){  
*102* **boolean** has = **false**;  
*103* Iterator<Player> temp = players.iterator();  
*104* Player player2 =**null**;  
105 **while** (temp.hasNext()) {  
*106* player2 = temp.next();  
*107* }  
*108* **return** player2==player;  
*109* }  
110  
*111* **public** **void** notSet(Player player) {  
*112* System.out.println();  
*113* container.addStack(player.stack);  
*114* }  
115  
*116* **public** **void** noticeNextGamblingPlayerByPlayer(){  
*117* **if** (iterator.hasNext()) {  
*118* (iterator.next()).getGamblingBegin(container.lastAddMan,container.totalStack);  
*119* }**else**{  
120 System.out.println(“这是最后一个玩家，赌博结束，最后总的赌注为” + container.getTotalStack());  
*121* }  
*122* }  
*123*  
*124* **public** **void** noticeSecondGamblingPlayer(){  
125 iterator.next();  
*126* iterator.next().getGamblingBegin(container.lastAddMan,container.totalStack);  
*127* }  
*128* }

测试类

*01* package CardShark;  
*02*  
*03* **public** **class** Main {  
*04*  
05 */\*\**  
*06* *\* @param args*  
*07* *\*/*  
*08* **public** **static** **void** main(String[] args) {  
*09* *//初始化*  
10 *//新建调停者*  
*11* Mediator mediator = **new** Mediator();  
*12*  
*13* *//注册容器*  
*14* Container container = **new** Container(mediator);  
15 mediator.registerContainer(container);  
*16*  
*17* *//注册会员*  
*18* Player player1 = **new** Player(mediator);  
*19* player1.name = “Tom”;  
20 mediator.registerPlayer(player1);  
*21*  
*22* Player player2 = **new** Player(mediator);  
*23* player2.name = “Mike”;  
*24* mediator.registerPlayer(player2);  
25  
*26* Player player3 = **new** Player(mediator);  
*27* player3.name = “Jim”;  
*28* mediator.registerPlayer(player3);  
*29*  
30 Player player4 = **new** Player(mediator);  
*31* player4.name = “Hebe”;  
*32* mediator.registerPlayer(player4);  
*33*  
*34*  
35 *//注册发牌人*  
*36* CardSender cardSender = **new** CardSender(mediator);  
*37* mediator.registerCardSender(cardSender);  
*38*  
*39* *//开始下注*  
40 cardSender.startStack();  
*41*  
*42* *//赌博开始*  
*43* cardSender.startGambling();  
*44* }  
45  
*46* }

### 实现前面Bridge Pattern的示例

类图：



代码：

public class Drawing {

public void drawLine(){

}

public void drawCircle(){

}

}

public class V1Drawing extends Drawing {

private DP1 dp1;

public V1Drawing(){

dp1 = new DP1();

}

public void drawLine(){

dp1.draw\_a\_line();

}

public void drawCircle(){

dp1.draw\_a\_circle();

}

}

public class V2Drawing extends Drawing {

private DP2 dp2;

public V2Drawing(){

dp2 = new DP2();

}

public void drawLine(){

dp2.draw\_a\_line();

}

public void drawCircle(){

dp2.draw\_a\_circle();

}

}

public class Shape {

private Drawing drawing;

public Shape(Drawing d){

this.drawing = d;

}

public void draw(){

}

protected void drawLine(){

drawing.drawLine();

}

protected void drawCircle(){

drawing.drawCircle();

}

}

public class Rectangle extends Shape {

super(new V1Drawing());

public void draw(){

drawLine();

drawLine();

.

.

}

}

public class Circle extends Shape {

super(new V2Drawing());

public void draw(){

drawCircle();

}

}

## Design Pattern（2）

### 考虑实验中你所负责的模块，是否需要利用Façade模式？

不考

### 用Decorater解决下列新增问题

**public** **class** WritePrint {

**public** **void** write(**int** c){}

}

**public** **abstract** **class** Decorator **extends** WritePrint{

**private** WritePrint wp;

**public** **void** setWritePrint(WritePrint wp) {

**this**.wp = wp;

}

@Override

**public** **void** write(**int** c) {

**if** (wp != **null**) {

wp.write(c);

}

}

}

**public** **class** WriteDisk **extends** Decorator {

@Override

**public** **void** write(**int** c) {}

}

Adapter

**public** **interface** Enumeration {

**public** **void** hasMoreElements();

**public** **void** nextElement();

}

**public** **class** Adapter **implements** Enumeration {

**private** Iterator iterator = **new** ConcreteIterator();

**public** **void** hasMoreElements() {

iterator.hasNext();

}

**public** **void** nextElement() {

iterator.next();

}

}

**public** **interface** Iterator {

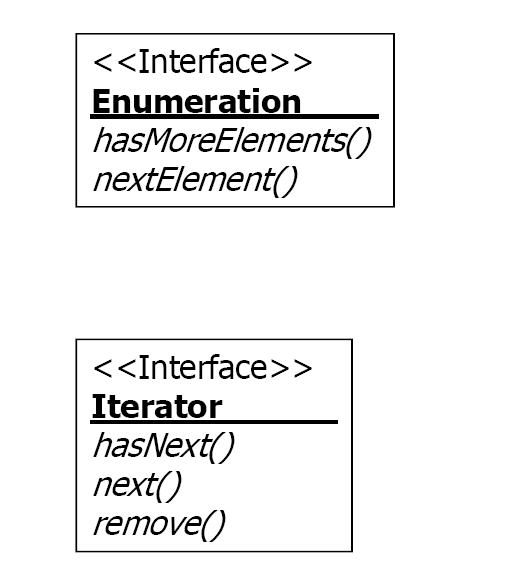
**public** **void** hasNext();

**public** **void** next();

**public** **void** remove();

}

### Adapt enumeration to iterator



### 策略模式思考题：

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？

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？

如果一个对象集除了接口之外，全是差异行为，如何处理？

从共性和差异性角度，如何解释Bridge模式？

* =>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？

=>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？

=>如果一个对象集除了接口之外，全是差异行为，如何处理？

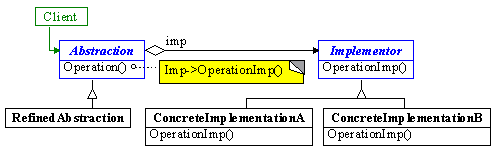
=>从共性和差异性角度，如何解释Bridge模式？

1）将与其他对象连接的部分分离出去

2）每个行为都封装出去

3）保留接口，其他全部做成策略

4）（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。

### 状态模式思考题：

比较strategy 与 state

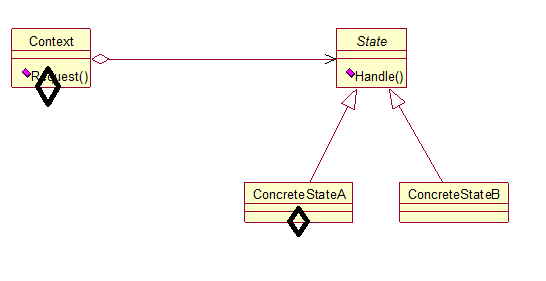
1 of N or M of N?

* 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?

* 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）



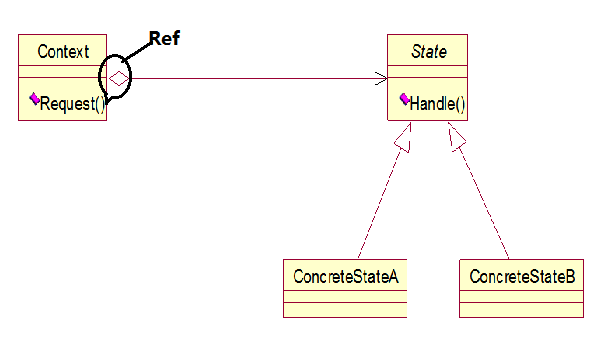
How to change?

Fixed rules or table-driven (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



“Context decide changing of ConcreteState object ”与“ ConcreteState decide changing of ConcreteState object”有何不同？

* 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

## Design Pattern（3）

### 以singleton为基础，编写程序解决Limited instance permitted

//实例的状态

Public enum Status {

Busy,//被客户占用

Free//空闲态

}

public class SingletonN {

private Status status;

private static ArrayList<SingletonN> instances;

public final static int N = 3;

//

private SingletonN() {

status = Status.Free;

}

//

public static SingletonN getInstance() {

if (instances == null) {

instances = new ArrayList<SingletonN>();

}

for (int i = 0; i < instances.size(); i++) {

if (instances.get(i).status == Status.Free) {

instances.get(i).status = Status.Busy;

return instances.get(i);

}

}

if (instances.size() < N) {

SingletonN s = new SingletonN();

s.status = Status.Busy;

instances.add(s);

return s;

}

return null;//表示没有实例可以用

}

//

public static boolean destory(SingletonN s) {

if (instances != null) {

for (int i = 0; i < instances.size(); i++) {

if (instances.get(i).equals(s)) {

instances.get(i).status = Status.Free;

return true;

}

}

}

return false;//没有destory成功

}

}

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

public class FactoryPizza {

public Pizza createPizza(){}

}

public abstract class Decorator extends FactoryPizza{

private FactoryPizza fp;

public void setFactoryPizza(FactoryPizza fp) {

this.fp = fp;

}

@Override

public Pizza createPizza() {

if(fp != null) {

return fp.createPizza();

}

}

}

public class T1Factory extends Decorator {

@Override

public Pizza createPizza() {

return new T1Pizza();

}

}

public class Pizza {

public void prepare() {}

public void bake() {}

public void cut() {}

public void box() {}

}

public class T1Pizza extends Pizza {}

### 工厂模式思考题

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。