**设计模式**

**软件设计的 开-闭原则：**

开闭原则是：一个系统要对扩展开放，而对修改关闭。这里的所指的意思就是在不修改以前已经写好的代码或者尽量不改的情况下来扩展系统的功能。  
由于从抽象层导出一个或多个新的具体类可以改变系统的行为，因此系统的设计对扩展是开放的，这就满足了开闭原则的第一条。  
在语言里，可以给出一个或多个抽象类或者接口，规定出所有的具体类必须提供的方法的特征作为系统设计的抽象层。这个抽象层预见了所有的可扩展性，因此，在任何扩展情况下都不会改变。这就使得系统的抽象不需要修改，从而满足了开闭原则的第二条，对修改关闭。

**Design pattern 分类**

**(1)、创建型，一般通过用来创建对象 Factory Method Pattern、Builder Pattern、Prototype Pattern、Singleton Pattern.**

**(2)、结构型，对象组合来获取更大的结构 Adapter Pattern、 Proxy Pattern、 Facade Pattern、Bridge Pattern、Decorator Pattern。**

**(3)、行为模型，处理对象如何交互 Observer Pattern、 Template Pattern、 Strategy Pattern、 State Pattern、 Command Pattern、 Iterator Pattern。**

**一：单态模式(Single Pattern)**

**注意：构造函数可见性为private。这样使得外面的类不能通过引言来产生对象。**

**构造器（Constructor）声明为private的话，外面不能实例化，典型的单例模式和一些工具类（提供静态方法），都把构造器声明为private。所以要得到这个类的实例只能是类名。method()取的它实例。**

**[java]** [view plain](http://blog.csdn.net/cdl2008sky/article/details/3335385) [copy](http://blog.csdn.net/cdl2008sky/article/details/3335385)

1. **public** **class** Singleton{
3. **private** **static** Singleton instance=**new** SinglePattern();
5. **private** Singleton(){}
7. **public** **static** Singleton getInstance(){
9. **return** instance;
11. }
13. }

**线程安全的Singleton**

**[java]** [view plain](http://blog.csdn.net/cdl2008sky/article/details/3335385) [copy](http://blog.csdn.net/cdl2008sky/article/details/3335385)

1. **public** **class** Singleton{
3. **private** Singleton instance = **null**;
5. **public** **static** Synchronizeed  Singleton getInstance(){
7. **if** (instance == **null**)  {
9. instance = **new** Singleton();
11. }
13. **return** instance;
15. }
17. }

为什么需要使用Singleton:单态模式是保证一个类有且仅有一个对象实例。单态模式经常使用在控制资源的访问上。例如配置文件管理类、数据工具类或Socket连接等。单态模式可以控制在某个时刻只有一个线程访问资源。

应用场景：

1. Windows的Task Manager（任务管理器）就是很典型的单例模式（这个很熟悉吧），想想看，是不是呢，你能打开两个windows task manager吗？ 不信你自己试试看哦~

2. windows的Recycle Bin（回收站）也是典型的单例应用。在整个系统运行过程中，回收站一直维护着仅有的一个实例。

3. 网站的计数器，一般也是采用单例模式实现，否则难以同步。

4. 应用程序的日志应用，一般都何用单例模式实现，这一般是由于共享的日志文件一直处于打开状态，因为只能有一个实例去操作，否则内容不好追加。

5. Web应用的配置对象的读取，一般也应用单例模式，这个是由于配置文件是共享的资源。

6. 数据库连接池的设计一般也是采用单例模式，因为数据库连接是一种数据库资源。数据库软件系统中使用数据库连接池，主要是节省打开或者关闭数据库连接所引起的效率损耗，这种效率上的损耗还是非常昂贵的，因为何用单例模式来维护，就可以大大降低这种损耗。

7. 多线程的线程池的设计一般也是采用单例模式，这是由于线程池要方便对池中的线程进行控制。

8. 操作系统的文件系统，也是大的单例模式实现的具体例子，一个操作系统只能有一个文件系统。

9. HttpApplication 也是单位例的典型应用。熟悉ASP.Net(IIS)的整个请求生命周期的人应该知道HttpApplication也是单例模式，所有的HttpModule都共享一个HttpApplication实例.

总结以上，不难看出：

　　单例模式应用的场景一般发现在以下条件下：

　　（1）资源共享的情况下，避免由于资源操作时导致的性能或损耗等。如上述中的日志文件，应用配置。

　　（2）控制资源的情况下，方便资源之间的互相通信。如线程池等。

大家可以追加其他常见的应用场景哈~~非常欢迎~~

**[java]** [view plain](http://blog.csdn.net/cdl2008sky/article/details/3335385) [copy](http://blog.csdn.net/cdl2008sky/article/details/3335385)

1. **public** **class** DBUtil {
3. //单一实例
4. **private** **static** **final** DBUtil instance = **null**;
6. //数据源的JNDI
7. **private** **static** **final** String datasource = "java:comp/env/jdbc/dataS";
9. //私有构造方法,防止外部实例化
10. **private** DBUtil() {
12. }
14. //数据库工具类实例工厂
15. **public** **static** **synchronized** DBUtil getInstance() {
16. **if**(instance == **null** ){
17. instance = **new** DBUtil();
18. }
19. **return** instance;
20. }

23. //业务方法:用于获取数据库连接
24. **public** Connection getConnection() {
25. Connection conn = **null**;
26. **try** {
27. Context ctx = **new** InitialContext();
28. DataSource ds = (DataSource) ctx.lookup(datasource);
29. conn = ds.getConnection();
30. } **catch** (NamingException e) {
31. System.out.println("获取数据源异常,请AppServer的JNDI数据源配置!");
32. e.printStackTrace();
34. } **catch** (SQLException e) {
35. System.err.println("获取数据库连接发生异常!");
36. e.printStackTrace();
37. }
38. **return** conn;
39. }
41. }

**二：工厂模式(Factory Pattern)**

    (a):简单工厂模式(Simple Factory Pattern)

**[java]** [view plain](http://blog.csdn.net/cdl2008sky/article/details/3335385) [copy](http://blog.csdn.net/cdl2008sky/article/details/3335385)

1. **public** **interface** AB { **public** **void** drive();};
3. **public** A **implements** AB{ **public** **void** drive(){ System.out.println("My is A!") }};
5. **public** B **implements** AB{ **public** **void** drive(){ System.out.println("My is B!") }};
7. **public** **class** Factory{
9. **public** **static** AB createFactory(**int** i){  //返回的是静态的抽象产品角色
11. **switch**(i){
13. **case** 1:
15. **return** **new** A();
17. **case** 2:
19. **return** **new** B();
21. }
23. }
25. **public** operatorFactory{
27. AB ab=Factory.createFactory(1);
29. ab.dirive();//输出A
31. }

  简单工厂模式符合设计开闭原则的第一个原则，对扩展开发，但不符合第二个原则，如果再增加一个产品，则在工厂里增加相应的业务逻辑和判断逻辑。

改进版，通过类的反射机制来实现

**[java]** [view plain](http://blog.csdn.net/cdl2008sky/article/details/3335385) [copy](http://blog.csdn.net/cdl2008sky/article/details/3335385)

1. **public** **class** Factory{
3. **public** **static** AB createFactory(String className){
5. Class c = Class.forName(className);
7. AB ab = (AB)c.newInstance();
9. **return** ab;
11. }
13. Client:
14. AB a = createFactory("cn.com.chenlly.A");

    (b):工厂方法模式(Abstract Factory Pattern)：当添加一个操作时，需要修改静态工厂方法不符合开闭原则（对扩展开放，对修改封闭)。把工厂里的静态工厂方法由不同的工厂子类来分担。

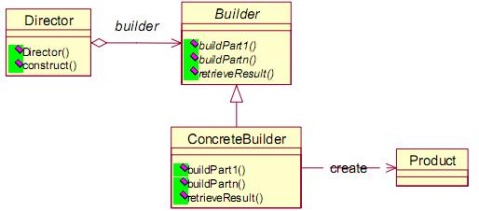
**[java]** [view plain](http://blog.csdn.net/cdl2008sky/article/details/3335385) [copy](http://blog.csdn.net/cdl2008sky/article/details/3335385)

1. **public** **interface** AB {
2. **public** **void** drive();
3. }
5. **public** A **implements** AB{
6. **public** **void** drive(){
7. System.out.println("My is A!");
8. }
9. }
11. **public** B **implements** AB{
12. **public** **void** drive(){
13. System.out.println("My is B!");
14. }
15. }
17. // 把静态工厂方法抽象出来
19. **public** **interface** CreateAbstractorFacto{
20. pubic AB createFactory();
21. }
23. **public** createAFactory **implements** CreateAbstractorFactory{
24. **public** AB createFactory(){
25. **return** **new** A();
26. }
27. }
29. **public** createBFactory **implements** CreateAbstractorFactory{
30. **public** AB createFactory(){
31. **return** **new** B();
32. }
33. }
35. **public** operatorFactory{
36. CreateAbstractorFactory  caf = **new** createAFactory ();
37. A  a = caf.createFactory();// 有A的工厂类制造出A
38. a.drive();  // 输出A
39. }

**三 建造模型（Builder Pattern）**

**客户首先创建一个指导者对象和一个建造者角色，并将建造者角色传入指导者对象进行配置。然后，指导者按照步骤调用建造者的方法创建产品。最后客户从建造者或指导者那里得到产品。建造模式将复杂对象的组件和表示方式封装在一个具体的建造角色中，而由指导者来协调建造者角色来得到具体的产品实例。**

**类图：**

****

**[java]** [view plain](http://blog.csdn.net/cdl2008sky/article/details/3335385) [copy](http://blog.csdn.net/cdl2008sky/article/details/3335385)

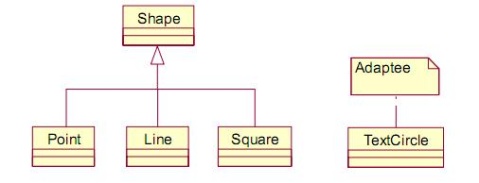
1. Interface Builder{
3. **public** buildPart1();
5. **public** buildPart2();
7. **public** buildPart3();
9. **public** Procedure getResult();
11. }
13. **public** **class** CreateBuilder **implements** Builder{
15. Part1 part1;
17. Part2 part2;
19. Part3 part3;
21. **public** **void** buildPart1{//构建Part1代码}
23. **public** **void** buildPart2{//构建Part2代码}
25. **public** **void** buildPart3{//构建Part3代码}
27. **public** Procedure getResult(){
29. //产品生产
31. }
33. }
35. **public** **class** Director{
37. **private** Builder build;
39. **public** Director(build){
41. **this**.build=build;
43. }
45. **public** **void** construct(){
47. // 将部件partA partB partC最后组成复杂对象
48. //这里是将车轮 方向盘和发动机组装成汽车的过程
50. build.buildPart1();
52. build.buildPart2();
54. build.buildPart3();
56. }
58. }


62. //调用：
64. CreateBuilder  cb=**new** CreateBuilder();
66. Direct dt=**new** Direct(cb);
68. dt.construct();
70. cb.getResult();
72. **interface** Produce {}  //对象产品
74. **interface** Part(){}        //对象附件

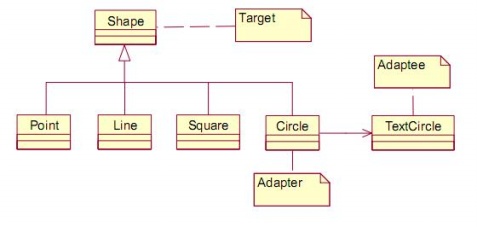
**四 适配器模式（Adapter pattern）**

**将一个类的接口转换为客户希望的另一个类的接口**

**A接口使用了方法1来实现某个功能，B接口使用了方法2来实现某个功能。修改任何一个接口中的方法都要修改其实现和使用到的地方。此模式是为了在面向接口编程中更好的复用。**



形状接口定义了它所对应的绘制图形所有的基本方法，现在要绘制椭圆，庆幸自己发现系统中已经有了绘制图形的实现。但是发现Shape接口中定义的方法和TextCircle中定义的方法不一样。



看在Circle中如何实现

**[java]** [view plain](http://blog.csdn.net/cdl2008sky/article/details/3335385) [copy](http://blog.csdn.net/cdl2008sky/article/details/3335385)

1. **public** **class** Circle **extends** Shape{
3. **private** TextCircle tc;
5. **public** Circle(){
7. tc=**new** TextCircle;
9. }
11. **public** **void** display(){
13. tc.displayCircle();
15. }
17. }

**五  桥接模式（bridge pattern）**

**桥梁模式主要的手段用动态的聚合关系，代替为静态的继承关系，从而达到系统的重构，使系统具有更高的复用.**用意是将问题的抽象和实现分离开来实现，通过用聚合代替继承来解决子类爆炸性增长的问题。

**当系统有两个可变因素时 我就应该考虑到桥梁模式，至少它应该在你的脑子里闪过**

 如 形状接口中，可能还会增加一个形状，画法可能除了画法一还有画法二，这就要要考虑bridge pattern

**1：找出变化并且封装**

**2：优先考虑聚合而不是继承**

**eg:**

抽象形状(抽象角色 形状)

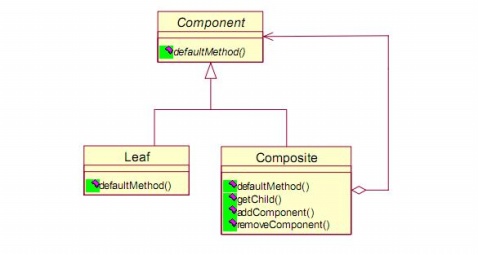
**[java]** [view plain](http://blog.csdn.net/cdl2008sky/article/details/3335385) [copy](http://blog.csdn.net/cdl2008sky/article/details/3335385)

1. **public** **interface** Shape{
3. **public** **void** doDraw();
5. }
7. **public** **class** Circle **implements** Shape{}
9. **public** **class** Rectangle **implements** Shape({}
11. 抽象方法（实现角色 方法）
13. **public** **interface** Drawing {
15. **public** **void** draw();
17. }
19. **public** **class** draw1 **implements** Drawing{}
21. **public** **class** draw2 **implements** Drawing {}
23. //实现抽象角色
25. **public** **class** Circle **implements** Shape(){
27. **private** Drawing drawing;
29. //get() set()
31. **public** **void** doDraw(){
33. drawing.draw();
35. }
37. }
39. //客户端调用
41. **public** **void** main(String []args){
43. Shape circle=**new** Circle(**new** drawing3);
45. circle.doDraw();
47. }

仔细体会了一下桥梁模式，感觉凡是**‘调用和实现’**之间的问题都可以用桥梁模式解决。

**六 组合模型 （composite）**

    Composite模式的一个重要思想是递归组合，关键是一个抽象类，它既可以代表组合对象，又可以代表一个被组合的对象。经常在Java开发中遇到**的容器**就使用了组合模式，一方面组件可以放在容器中，另一方面容器也可以作为组件放在另外的容器中。



  removeComponent(Component com) com 可以是Component类的子类对象也可以是它本身。

**七 装饰模式（decorator Pattern 或 包装器模式Wrapper Pattern）**

应用：Java 的I/O API就是使用Decorator实现的 ,I/O变种很多,如果都采取继承方法,将会产生很多子类,显然相当繁琐。如：

FileReader fr = new FileReader(filename);  
BufferedReader br = new BufferedReader(fr);

      定义：动态给一个对象添加一些额外的职责。这些功能需要由用户动态决定加入的方式和时机.Decorator提供了"即插即用"的方法,在运行期间决定何时增加何种功能。使用继承实现功能拓展,我们必须可预见这些拓展功能,这些功能是编译时就确定了,是静态的.

eg:

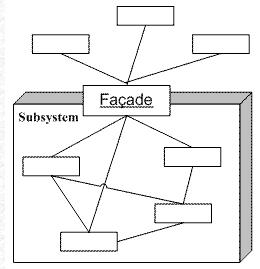
**[java]** [view plain](http://blog.csdn.net/cdl2008sky/article/details/3335385) [copy](http://blog.csdn.net/cdl2008sky/article/details/3335385)

1. **public** **interface** Work{
3. **public** **void** insert();
5. }
7. **public** **class** SeqarePeg **implements** Work{
9. **public** **void** insert(){
11. System.out.println("插入方桩");
13. }
15. }
17. **public** Decorator **implements** Work{
19. **private** Work work;
21. **private** List list=**new** ArrayList();
23. **public** Decorator(Work work){
25. **this**.work=work;
27. list.add("挖坑");
29. list.add("装木板');//打桩的额外功能
31. }
33. **public** **void** insert(){
35. work.insert();
37. Iterator it=list.iterator();
39. **while**(it.hasNext()){
41. System.out.println(((String)it.next())+"正在进行");
43. }
45. }
47. }

**八 门面模式（facade Pattern）**

          又叫外观模式 ，为子系统中的一组接口提供一个一致的界面。

门面模式的门面类将客户端与子系统的内部复杂性分隔开，使得客户端只需要与门面对象打交道，而不需要与子系统内部的很多对象打交道。 一个典型应用就是进行数据库连接。一般我们在每一次对数据库进行访问，都要进行以下操作：先得到 connect 实例，然后打开 connect 获得连接，得到一个 statement ，执行 sql 语句进行查询，得到查询结果集。



 eg:

**[java]** [view plain](http://blog.csdn.net/cdl2008sky/article/details/3335385) [copy](http://blog.csdn.net/cdl2008sky/article/details/3335385)

1. 买衣服
3. **public** **class** Clothing {
4. **public** **void** getClothing(){
5. System.out.println("买衣服");
6. }
7. }
9. 买鞋子
11. **public** **class** Shoes {
12. **public** **void** getShoes(){
13. System.out.println("买鞋子");
14. }
15. }
16. 买玩具
18. **public** **class** Toy {
19. **public** **void** getToys(){
20. System.out.println("买玩具");
21. }
22. }


26. //商店(门面)
28. **public** **class** ShoppingCenter {
30. **public** **void** clothingStore(){
31. Clothing c = **new** Clothing();
32. c.getClothing();
33. }
35. **public** **void** shoesStore(){
36. Shoes s = **new** Shoes();
37. s.getShoes();
38. }
40. **public** **void** ToyStore(){
41. Toy t = **new** Toy();
42. t.getToys();
43. }
44. }
46. //客户端public class Buys {
47. **public** **void** buy(){
48. //买衣服
49. Clothing c = **new** Clothing();
50. c.getClothing();
51. }
52. //方法(1)原始方法
53. **new** Buys().buy(); //没有门面，客户直接去货架上拿东西
54. //方法(2)增加门面商店后的方法。客户只需要在商店里拿东西了
56. ShoppingCenter sc = **new** ShoppingCenter();
57. sc.clothingStore();
58. sc.shoesStore();
59. sc.ToyStore();
60. }

**九 代理模式（proxy Pattern）**

定义：为其它对象提供一种代理以控制这个对象的访问，一般客户不想直接或者不能引用某一个对象。而代理则可以在客户和目标对象之间起到中间作用。proxy 可以控制原对象，但不能修改原对象。代理类必须和原来的接口(目标类)是一样的。

eg:把要进行安全性检查的一个方法模块化，写在类里。当我们生产了代理对象的时候，如果调用代理方法的时候，会自动默认调用invoke方法。这个模块化的类就是所谓的切面(Aspect),安全性检查就是advice。

**[java]** [view plain](http://blog.csdn.net/cdl2008sky/article/details/3335385) [copy](http://blog.csdn.net/cdl2008sky/article/details/3335385)

1. **import** java.lang.reflect.InvocationHandler;//是代理实例的调用处理程序 实现的接口。
2. **import** java.lang.reflect.Proxy;

5. **public** **class** SecurityHandler **implements** InvocationHandler{
6. **private** Object targetObject;//目标实例
7. **public** Object newProxy(Object targetObject){
8. **this**.targetObject = targetObject;
9. //返回一个目标对象的代理类
10. **return** Proxy.newProxyInstance(targetObject.getClass().getClassLoader(),//目标程序的类加载器
11. targetObject.getClass().interfaces(),//目标程序的接口
12. **this**);//代理指定调用的处理程序
13. }
15. **public** Object invoke(Object proxy,Method method,Object args) **throws** Throws{
16. //安全性检查,这是一个before Advice
17. checkSecurity();
18. //调用目标对象的正是实现,有可能有返回值，
19. Object res = **null**;
20. **try**{
21. res = method.invoke(targetObject,args);
22. }**catch**(Exception ex){
23. ex.printStackTrace();
24. }
25. }

28. //安全性检查
29. **private** **void** checkSecurity(){
30. //todo something;
31. }
32. }
34. //客户端调用
35. **public** **class** Client{
36. SecurityHandler sh = **new** SecurityHandler();
37. //得到代理类,Annotation就是使用了动态代理机制
38. UserManagerImpl userManagerImpl = (UserManagerImpl)sh.newProxy(**new**   UserManagerImpl());
39. userManagerImpl.addUser();
40. userManagerImpl.deleteUser(12);
41. }

**十 享元模式（FlyWeight Pattern）**

享元模式属于结构型模式，其意图是运用共享技术有效地支持大量细粒度的对象。

 FlyweightFactory(Object obj)：创建并管理Flyweight对象。确保合理的共享Flyweight。当用户请求一个flyweight时，FlyweightFactory对象提供一个已经创建的实例或者创建一个。比如：String类的字符串池，每次创建一个新的字符串对象都会去字符串池里去找有没有这个对象，如果没有创建一个新的，如果共享原来的对象。

**[java]** [view plain](http://blog.csdn.net/cdl2008sky/article/details/3335385) [copy](http://blog.csdn.net/cdl2008sky/article/details/3335385)

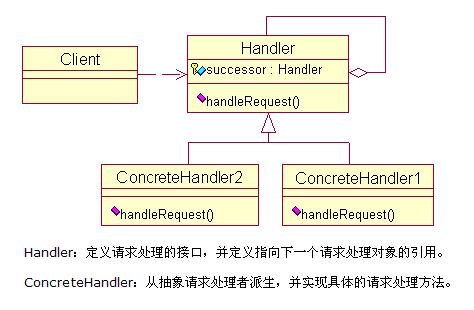
1. 享元接口
2. **public** **interface** FlyWeight {
3. **public** **void** operator();
4. }
6. 享元具体实现
7. **public** **class** ConcreteFlyWeight **implements** FlyWeight {
8. **private** String str;
10. **public** ConcreteFlyWeight(String str){
11. **this**.str = str;
12. }
13. **public** **void** operator() {
14. System.out.println("this is FlyWeight===》"+str);
15. }
17. }

20. 享元工厂类
21. **public** **class** FlyWeightFactory {
23. **private** Map map = **new** HashMap();
25. **public** FlyWeight getFlyWeight(Object obj) {
26. FlyWeight flyWeight = (FlyWeight) map.get(obj);
27. **if** (flyWeight == **null**) {
28. flyWeight = **new** ConcreteFlyWeight(obj.toString());
29. map.put(obj, flyWeight);
30. }
32. **return** flyWeight;
33. }
34. }


38. 客户端
39. **public** **class** FlyWeightClient {
40. **public** **static** **void** main(String[] args) {
41. FlyWeightFactory factory = **new** FlyWeightFactory();
42. FlyWeight f1;
43. FlyWeight f2;
44. FlyWeight f3;
45. FlyWeight f4;
46. FlyWeight f5;
47. FlyWeight f6;
48. f1 = factory.getFlyWeight("Good");
49. f2 = factory.getFlyWeight("Bad");
50. f3 = factory.getFlyWeight("Head");
51. f4 = factory.getFlyWeight("Good");
52. f5 = factory.getFlyWeight("Good");
53. f6 = factory.getFlyWeight("Good");
54. f1.operator();
55. f2.operator();
56. f3.operator();
57. f4.operator();
58. f5.operator();
59. f6.operator();
60. }
61. }
62. f4,f5,f5 不会重新创建对象，而是共享了f1

**十一 责任链模式**

       责任链模式用于弱化请求发生者和请求处理者之间的关系。当多个对象都可以对请求进行处理，但不同的对象能处理的请求类型不同时，可以通过指向另一个对象的引用把这些对象连成一条责任链。当 Client 发出一个请求时，并不知道具体由哪个对象进行处理，它看到的只是一条责任链，将请求直接交给责任链，请求会在责任链中传递，直到找到一个能够进行处理的对象或者遍历结束找不到一个能够处理的对象为止。



**[c-sharp]** [view plain](http://blog.csdn.net/cdl2008sky/article/details/3335385) [copy](http://blog.csdn.net/cdl2008sky/article/details/3335385)

1. **abstract** **public** **class**  Employee {
2. **protected** String name;
4. **protected** Employee boss; //属性 boss 指向自己的上级老板
6. **protected** **int** requestLevel;
8. **public** Employee(String name, Employee boss){
9. **this**.name = name;
10. **this**.boss = boss;
11. }
13. **public** String getName(){
14. **return** name;
15. }
17. **public** **void** handleRequest(Request request){
18. **if** (boss!=**null**){
19. boss.handleRequest(request);
20. } **else**{
21. System.**out**.println("No body can handle the request!");
22. }
23. }
24. }


28. Manager:
30. **public** **class** Manager extends Employee {
32. /\*\*
33. \* @param name
34. \* @param boss
35. \*/
36. **public** Manager(String name, Employee boss) {
37. super(name, boss);
38. requestLevel = 1;
39. }
41. **public** **void** handleRequest(Request request){
42. **if** (requestLevel >= request.getType()){
43. System.**out**.println("(Manager) say: I can handle the request");
44. } **else** {
45. System.**out**.println("(Manager) say: I can not handle the request.My boss"+boss.getName()+" can handle the request");
46. super.handleRequest(request);
47. }
48. }
49. }


53. Director:
55. **public** **class** Director extends Employee {
57. /\*\*
58. \* @param name
59. \* @param boss
60. \*/
61. **public** Director(String name, Employee boss) {
62. super(name, boss);
63. requestLevel = 2;
64. }
65. **public** **void** handleRequest(Request request){
66. **if** (requestLevel >= request.getType()){
67. System.**out**.println("(Director) say: I can handle the request");
68. } **else** {
69. System.**out**.println("(Director) say: I can not handle the request.My boss"+boss.getName()+" can handle the request");
70. super.handleRequest(request);
71. }
72. }
74. }


78. CEO:
80. **public** **class** CEO extends Employee {
82. /\*\*
83. \* @param name
84. \* @param boss
85. \*/
86. **public** CEO(String name, Employee boss) {
87. super(name, boss);
88. requestLevel = 3;
89. }
91. **public** **void** handleRequest(Request request){
92. **if** (requestLevel >= request.getType()){
93. System.**out**.println("(Manager) say: I can handle the request");
94. } **else** {
95. System.**out**.println("(Manager) say: I can not handle the request.My boss "+boss.getName()+"can handle the request");
96. super.handleRequest(request);
97. }
98. }
99. }


103. Request:
105. **public** **class** Request {
106. **public** **static** final **int** REQUEST\_LEVEL1 = 1;
108. **public** **static** final **int** REQUEST\_LEVEL2 = 2;
110. **public** **static** final **int** REQUEST\_LEVEL3 = 3;
112. **private** **int** type;
114. **public** Request(**int** type) {
115. **this**.type = type;
116. }
118. **public** **int** getType() {
119. **return** type;
120. }
121. }

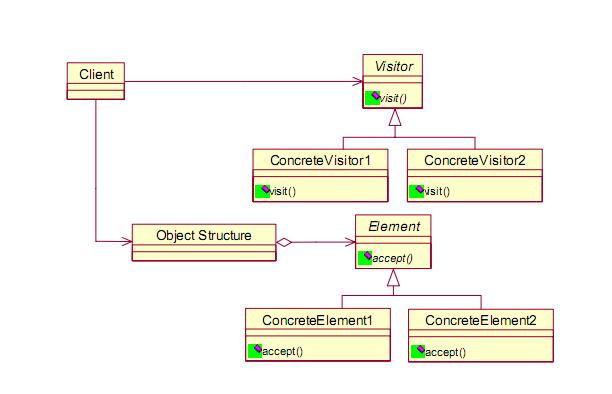

125. Client:
127. **public** **class** responsibilityChain {
129. /\*\*
130. \* @param args
131. \*/
132. **public** **static** **void** main(String[] args) {
133. Employee ceo = **new** CEO("Tom",**null**);
135. Employee director = **new** Director("chen", ceo);
137. Employee manager = **new** Manager("liling",director);
139. Request request1 = **new** Request(Request.REQUEST\_LEVEL1);
140. Request request2 = **new** Request(Request.REQUEST\_LEVEL2);
141. Request request3 = **new** Request(Request.REQUEST\_LEVEL3);
143. manager.handleRequest(request1);
145. manager.handleRequest(request2);
147. manager.handleRequest(request3);
148. }
150. }
152. 后台日志：
154. (Manager) say: I can handle the request
155. (Manager) say: I can not handle the request.My bosschen can handle the request
156. (Director) say: I can handle the request
157. (Manager) say: I can not handle the request.My bosschen can handle the request
158. (Director) say: I can not handle the request.My bossTom can handle the request
159. (Manager) say: I can handle the request

eg: 不同角色的人员能够处理的 Request 的级别不同，Manager 只能处理一般的 Request，Director 具有更高的权限，能够审批更高级别的 Request，CEO 能处理的 Request 级别还要高些。

抽象类：

**十二 访问者模式**

   在不修改已有程序的前提下，通过添加额外的"访问者"来完成对已有代码的提升。  
定义：表示作用于某一结构对象的各个元素的操作。它可以使你在不修改各个元素类的情况下定义作用于这些元素的新的操作。



**[java]** [view plain](http://blog.csdn.net/cdl2008sky/article/details/3335385) [copy](http://blog.csdn.net/cdl2008sky/article/details/3335385)

1. **(1) 访问者接口**
3. **public interface Visitor {**
5. **public void visit(Red red);**
7. **public void visit(Yellow yellow);**
9. **public void visit(Green green);**
10. **}**
12. **(2) 访问者实现类 Bee2**
14. **public class Bee2 implements Visitor {**
16. **public void visit(Red red) {**
17. **System.out.println("Bee2 and red");**
18. **}**
20. **public void visit(Yellow yellow) {**
21. **System.out.println("Bee2 and yellow");**
22. **}**
24. **public void visit(Green green) {**
25. **System.out.println("Bee2 and green");**
26. **}**
27. **}**

30. **//对象生成器 Generator(生成) Generic(普通)**
32. **public class ColorGenerator {**
33. **private static Random random = new Random();**
35. **public static Color newColor(){**
36. **switch(random.nextInt(3)){**
37. **default:**
38. **case 0: return new Red();**
39. **case 1: return new Yellow();**
40. **case 2: return new Green();**
41. **}**
42. **}**
43. **}**
45. **(3) 对象结构**
47. **public class  ColorStruct{**
49. **private ColorGenerator  colorGenerator;**
51. **private List<Color>  colors = new ArrayList()<Color>**
53. **public List<Color> getColors(){**
55. **for (int i = 0; i< 3; i++){**
56. **colors.add(ColorGenerator.newColor());**
57. **}**
59. **}**
61. **}**
63. **(4) 元素**
65. **public interface Color {**
66. **void accept(Visitor visitor);**
67. **}**


71. **public class Yellow implements Color {**
73. **public void accept(Visitor v) {**
74. **v.visit(this);**
75. **}**
77. **}**

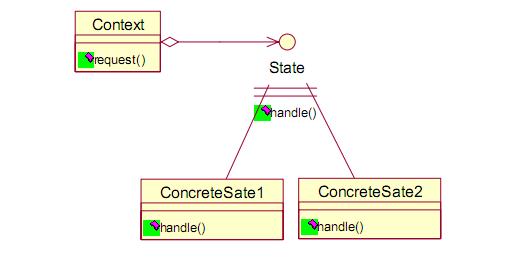

81. **public class Red implements Color {**
83. **public void accept(Visitor v) {**
84. **v.visit(this);**
85. **}**
87. **}**


91. **public class Green implements Color {**
93. **public void accept(Visitor v) {**
94. **v.visit(this);**
95. **}**
97. **}**


101. **Client:客户端 制作一个访问者Bee2**
103. **Visitor v;**
104. **v= new Bee2();**
105. **for (Iterator<Color> iterator =  colors.iterator(); iterator.hasNext();){**
106. **Color color = (Color)iterator.next();**
107. **color.accept(v);**
108. **}**

**十三 状态模式**

输入信号是事件；输出是状态结果，状态模式状态机就是一个黑盒子。  
状态模式主要突出了两个字:”改变”，对象的状态决定了状态的行为，事物的本质决定了事物的行为。  
对一个状态特别重要的对象，通过状态机来建模一个对象的状态；  
状态模式处理的核心问题是状态的迁移，因为在对象存在很多状态情况下，  
对各个business flow，各个状态之间跳转和迁移过程都是及其复杂的。  
例如一个工作流，审批一个文件，存在新建、提交、已修改。  
HR部门审批中、老板审批中、HR审批失败、老板审批失败等状态，涉及多个角色交互，涉及很多事件，  
这种情况下用状态模式(状态机)来建模更加合适；  
把各个状态和相应的实现步骤封装成一组简单的继承自一个接口或抽象类的类，  
通过另外的一个Context来操作他们之间的自动状态变换，通过event来自动实现各个状态之间的跳转  
参与者：         
  Context(环境，Person)定义客户感兴趣的类。  
  State(Moodstate)：定义一个接口以封装与Context的一个特定状态相关的行为  
  ConcreteState Subclasses(具体状态子类，如Angry)每一个子类实现一个与Context的状态相关的行为。  
他们的协作关系是：  
  Context将于状态相关的请求委托给当前的ConcreteState对象处理。  
  Context可将自身作为一个参数传递给处理该请求的状态对象，这使得状态对象在必要的时候可访问Context。  
  Context是客户使用的主要接口，客户可用状态对象来配置一个Context，一旦一个Context配置完毕，他的客户不再需要直接与状态对象打交道。  
  Context或者ConcreteState子类都可以决定哪个状态是另外那个状态的后继者，以及是在何种条件下进行状态转换。



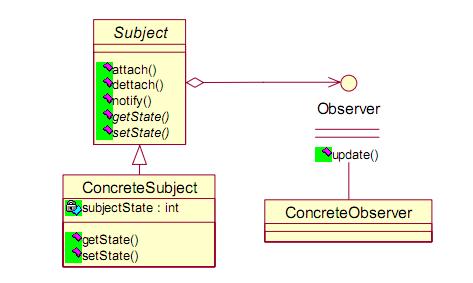
**[java]** [view plain](http://blog.csdn.net/cdl2008sky/article/details/3335385) [copy](http://blog.csdn.net/cdl2008sky/article/details/3335385)

1. **Context:**
3. **public class Persion {**
4. **private MoodState state;**
6. **public Persion(){**
7. **state = new Mad(this);//开始默认为抓狂状态**
8. **}**
10. **public void setState(MoodState state) {**
11. **this.state = state;**
12. **}**
14. **public void request(){**
15. **state.doSomething();**
16. **state.changeState();**
17. **}**
19. **}**
21. **状态接口：**
22. **public interface MoodState {**
23. **public void doSomething();**
25. **public void changeState();**
26. **}**

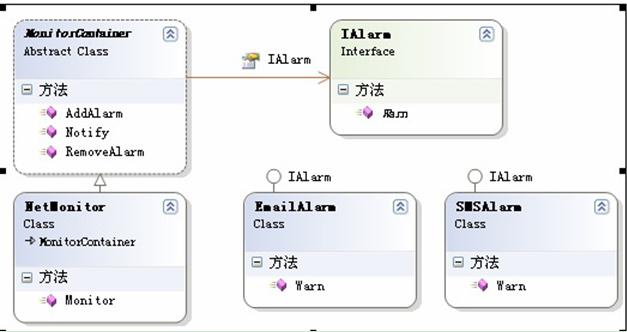

30. **抓狂状态：**
31. **public class Mad implements MoodState {**
33. **Persion persion;**
34. **public Mad(Persion persion){**
35. **this.persion = persion;**
36. **}**
38. **public void changeState() {**
39. **persion.setState(new Angry(persion));**
40. **}**
42. **public void doSomething() {**
43. **System.out.println("I'm Mad!");**
45. **}**
47. **}**
49. **生气状态：**
50. **public class Angry implements MoodState {**
52. **Persion persion;**
54. **public Angry(Persion persion){**
55. **this.persion = persion;**
56. **}**
58. **public void changeState() {**
59. **persion.setState(new Happy(persion));**
60. **}**
62. **public void doSomething() {**
63. **System.out.println("I'm Angry!");**
64. **}**
66. **}**
68. **高兴状态：**
69. **public class Happy implements MoodState {**
71. **Persion persion;**
73. **public Happy(Persion persion){**
74. **this.persion = persion;**
75. **}**
76. **public void changeState() {**
77. **persion.setState(new Mad(persion));**
78. **}**
80. **public void doSomething() {**
81. **System.out.println("I'm Happy!");**
82. **}**
84. **}**

**十四 观察者模式**

  定义对象之间的一种一对多的关系，当一个对象的状态改变时， 所有依赖它的对象都会被通知并且自动更新.  
1， 观察者（具体执行操作的对象，有多个）   
2， 被观察者（顾名思义是被观察的对象，如果该对象发生某些变化则通知观察者执行对应的操作）   
参照 java.util.Observable java.util.Observer



下面以一个例子来讲解观察者模式  
一监控系统，需要根据系统某些变化报警，报警方式有Email、短信等多种，  
IAlarm是报警接口。(观察者)当我们的系统需要添加报警方式的时候只需实现IAlarm的Warn方法即可。Email类是EmailAlarm报警的实现，SMSAlarm类是短信报警的实现。  
MonitorContainer是监视器（抽象类）相当于被观察者。只负责通知变化，当子类调用Notify方法它即会通知报警模块报警，如Email和SMS(短信)。  
NetMonitor是其中的具体的监控模块，继承于MonitorContainer。当发现系统网络有问题时会调用父类的Notify方法。



**[java]** [view plain](http://blog.csdn.net/cdl2008sky/article/details/3335385) [copy](http://blog.csdn.net/cdl2008sky/article/details/3335385)

1. **(1)定义观察者：**
3. **public class MonitorContainer {**
4. **private int state;**
6. **private List<IAlarm> obs;**
7. **//创建被观察者时就创建一个它持有的观察者列表**
8. **public MonitorContainer(){**
9. **obs = new ArrayList<IAlarm>();**
10. **}**
12. **//添加观察者到观察者列表中去**
13. **public void addAlarm(IAlarm obj){**
14. **if (obj == null) {**
15. **throw new NullPointerException();**
16. **}**
17. **if(!obs.contains(obj)){**
18. **obs.add(obj);**
19. **}**
20. **}**
21. **//删除一个观察者**
22. **public void remove(IAlarm obj){**
23. **obs.remove(obj);**
24. **}**
25. **//通知操作,即被观察者发生变化，通知对应的观察者进行事先设定的操作**
26. **public void notifyAlarm(){**
27. **//state 越小，级别越低**
28. **if (state == 2){**
29. **//邮件预警**
30. **new EmailAlarm().warm();**
31. **}**
32. **if(state == 1){**
33. **//消息预警**
34. **new SMSAlarm().warm();**
35. **}**
36. **if(state == 0){**
37. **//级别最高都需要预警**
38. **for (int i = 0; i<obs.size(); i++ ){**
39. **obs.get(i).warm();**
40. **}**
41. **}**
42. **}**
44. **public int getState() {**
45. **return state;**
46. **}**
48. **public void setState(int state) {**
49. **this.state = state;**
50. **}**
51. **}**
53. **(2) 具体观察者(网络预警)**
55. **public class NetMonitor extends MonitorContainer {**
56. **public void doBusiness(int state){**
57. **super.setState(state);**
58. **//状态改变，通知观察者**
59. **notifyAlarm();**
60. **}**
61. **}**
63. **(3)定义观察者：**
65. **public interface IAlarm {**
66. **public void warm();**
67. **}**
69. **//具体观察者**
71. **public class EmailAlarm implements IAlarm {**
73. **public void warm() {**
74. **System.out.println("EmailAlarm:电子邮件预警!");**
75. **}**
77. **}**
79. **//具体观察者**
81. **public class SMSAlarm implements IAlarm {**
83. **public void warm() {**
84. **System.out.println("SMSAlarm:短信预警!");**
85. **}**
87. **}**
89. **Client:**
90. **public class ObserverClient {**
91. **public static void main(String []args){**
92. **NetMonitor netMonitor = new NetMonitor();**
93. **//添加观察者**
94. **EmailAlarm e = new EmailAlarm();**
95. **SMSAlarm s = new SMSAlarm();**
96. **netMonitor.addAlarm(e);**
97. **netMonitor.addAlarm(s);**
98. **netMonitor.doBusiness(0);**
99. **}**
100. **}**

**十五 备忘录模式**

      其意图是在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样以后就可以将对象恢复到原先保存的状态。  
备忘录实际上有两个接口，  
“备忘录管理者角色”只能看到备忘录提供的窄接口——对于备忘录角色中存放的属性是不可见的。不能对备忘录的属性操作或检查  
“备忘发起角色”则能够看到一个宽接口——能够得到自己放入备忘录角色中属性。

**[java]** [view plain](http://blog.csdn.net/cdl2008sky/article/details/3335385) [copy](http://blog.csdn.net/cdl2008sky/article/details/3335385)

1. **(1) 备忘录的发起者**
3. **public class Employee {**
4. **private String name;**
6. **private int age;**
8. **private String sex;**

11. **// 定义一个备忘录管理者**
12. **private Caretaker c = new Caretaker();**
14. **public Employee(String name, int age, String sex){**
15. **this.name = name;**
16. **this.age = age;**
17. **this.sex = sex;**
18. **}**
20. **//创建一个备忘录角色，并将当前状态属性存入，托给"备忘录管理者角色"存放**
21. **public  void createMemento(){**
22. **c.saveMemento(new Memento(name));**
23. **}**
25. **//读取备忘录角色以恢复以前的状态**
26. **public void readMemento(){**
27. **this.name = c.getMemento().getName();**
28. **}**
30. **public void show(){**
31. **System.out.println("Employee");**
32. **System.out.println(" name:"+name);**
33. **System.out.println(" age:"+age);**
34. **System.out.println(" sex:"+sex);**
35. **}**
37. **}**


41. **(2)备忘录**
43. **public class Memento {**
44. **private String name;**
46. **public Memento(String name){**
47. **this.name = name;**
48. **}**
50. **public String getName() {**
51. **return name;**
52. **}**
54. **public void setName(String name) {**
55. **this.name = name;**
56. **}**
57. **}**
59. **(3) 备忘录管理者**
61. **public class Caretaker {**
62. **private List<Memento> meList;**
64. **//注意:备忘录，只能恢复前一步，比如redo,undo**
65. **private int current;**
67. **public Caretaker(){**
68. **current = -1;**
69. **meList = new ArrayList<Memento>();**
70. **}**
72. **public void saveMemento(Memento memento){**
73. **for (int i= current+1; i<meList.size(); i++){**
74. **meList.remove(i);**
75. **}**
76. **current ++;**
77. **meList.add(memento);**
78. **}**
80. **public Memento getMemento(){**
81. **if (current>=0){**
82. **return meList.get(current--);**
83. **}**
84. **return null;**
85. **}**
86. **}**


90. **Client:**
92. **Employee employee = new Employee("chen",10,"男");**
93. **employee.createMemento();**
94. **employee.show();**
96. **employee.name ="qing";**
97. **employee.createMemento();**
98. **employee.show();**
100. **employee.name ="li";**
101. **employee.createMemento();**
102. **employee.show();**
104. **//从备忘录恢复**
105. **employee.readMemento();**
106. **employee.show();**
108. **//从备忘录恢复**
109. **employee.readMemento();**
110. **employee.show();**
112. **//从备忘录恢复**
113. **employee.readMemento();**
114. **employee.show();**

**十六 调停者模式**

   调停者模式有着分层设计的锥形，调停者模式包装了一系列对象相互作用的方式，使得这些对象不必相互明显作用。从而使他们可以松散偶合。  
系统中有大量的对象，这些对象既会影响别的对象，又会被别的对象所影响，因此常常叫做同事（Colleague）对象。   
抽象调停者（Mediator）角色：定义出同事对象到调停者对象的接口，其中主要的方法是一个（或多个）事件方法。在某些情况下，这个抽象对象可以省略。   
具体调停者（ConcreteMediator）角色:实现了抽象超类所声明的事件方法。具体调停者知晓所有的具体同事类，它从具体同事对象接收消息，向具体同事对象发出命令。  
抽象同事类（Colleague）角色:定义出调停者到同事对象的接口。同事对象只知道调停者而不知道其余的同事对象。

**[java]** [view plain](http://blog.csdn.net/cdl2008sky/article/details/3335385) [copy](http://blog.csdn.net/cdl2008sky/article/details/3335385)

1. (1)调停者接口
3. **public** **interface**  Mediator {
4. **public**  **void** colleagueChanged();
5. }
7. (2)具体调停者
9. **public** **class** ConcreteMediator **implements** Mediator {
11. **private** Landlord landlord;
13. **private** Lodger lodger;
15. /\*创建同事对象\*/
16. **public**  ConcreteMediator(){
17. landlord = **new** Landlord(**this**);
18. lodger = **new** Lodger(**this**);
19. }
21. **public** **void** colleagueChanged() {
22. //这样中介拿到房东的房屋信息(s1)和租房者的需求信息(s2)
23. landlord.action();
24. String s1 = landlord.getHousesInfo();
25. lodger.action();
26. String s2 = lodger.getRequireInfo();
27. System.out.println("房屋信息:"+s1);
28. System.out.println("需求信息:"+s2);
29. }
31. }


35. (3) 抽象同事者
37. **public** **abstract** **class** Colleague {
38. **private** Mediator mediator;
40. **public** Colleague(Mediator mediator){
41. **this**.mediator = mediator;
42. }
44. /\* 获得调停者对象\*/
45. **public** Mediator getMediator() {
46. **return** mediator;
47. }
49. //抽象行为
50. **public** **abstract** **void** action();
52. }


56. (4) 具体同时者 ，房东
58. **public** **class** Landlord **extends** Colleague {
60. **private** String housesInfo;
62. **public** Landlord(Mediator mediator) {
63. **super**(mediator);
64. }
66. /\*\*
67. \* 房东具体的行为方法,例如,将房间资料交给中介等
68. \*/
69. @Override
70. **public** **void** action() {
71. housesInfo = ("进行将房间资料交给中介等操作");
72. }
74. **public** String getHousesInfo() {
75. **return** housesInfo;
76. }
78. }

**十七 解释器模式**

  描写简单语法的解释器，主要用在面向对象语言开发编译器中  
定义上下文context类，定义HashMap   
定义抽象类表达式Expression,定义方法interpret(Context con)  
定义变量Variable，常量Constant，加法Add，减法Subtract，乘法Multiply ，除法Division 类表达式  
计算 (a\*b)/(a-b+2)

**十八 命令模式**

      解耦了发送者和接受者之间联系。 发送者调用一个操作，接受者接受请求执行相应的动作，因为使用Command模式解耦，  
发送者无需知道接受者任何接口。  
将这些命令封装成在一个类中,将这些命令封装成在一个类中,然后用户(调用者)再对这个类进行操作,  
这就是Command模式,换句话说,本来用户(调用者)是直接调用这些命令的,如菜单上打开文档(调用者),  
就直接指向打开文档的代码,使用Command模式,就是在这两者之间增加一个中间者,将这种直接关系拗断,  
同时两者之间都隔离,基本没有关系了.  
在这个例子中，被操作的对象有电风扇(fan)和灯(light)，他们主要有开和关两种动作。把操作开或者关的请求封装成一个对象，  
于是就有了在本例中就有FanOn(Off)Command和lightOn（Off）Command四个类，每个类都实现了Command接口，具备执行和撤销操作。  
这些对象的协作关系是 Client创建一个具体的命令对象（fanOnCommand） 并指定他的Receiver（fan）对象，  
Invoker(dealer)对象存储该具体的命令对象，该Invoker通过调用Command对象的Execute对象的Execute操作来提交一个请求，  
若该命令是可撤销的，具体的命令对象就再执行Execute之前存储当前的状态以取消该命令。  
具体对象调用它的Receiver对象的一些操作以执行该请求。

(5) 具体同时者 ，房客

public class Lodger extends Colleague {  
   
 private String requireInfo;

 public Lodger(Mediator mediator) {  
  super(mediator);  
  // TODO Auto-generated constructor stub  
 }  
      
 /\*\*  
  \* 房客具体的行为方法,例如,将租房需求交给中介等  
  \*/  
   
 @Override  
 public void action() {  
  requireInfo="进行将租房需求交给中介等操作";  
 }

 public String getRequireInfo() {  
  return requireInfo;  
 }

}

Client:

public class MediatorClient {  
 public static void main(String []args){  
  ConcreteMediator c = new ConcreteMediator();  
  c.colleagueChanged();  
 }  
}