单例模式：

举例

懒汉式：

class Singleton {

private static Singleton instance=null;

private Singleton(){}

public static Singleton getInstance() {

if(instance==null) {

instance=new Singleton();

}

return instance;

}

}

饿汉式：

class Singleton {

private static Singleton instance=new Singleton();

private Singleton(){}

public static Singleton getInstance() {

return instance;

}

}

优缺点

其中饿汉式预先加载，线程安全，代码简练。

懒汉式对象延迟加载，效率高。

建造者模式：

举例

public class Computer

{

//电脑中零件的集合

List<string> parts = new List<string>();

//添加零件

public void add(string part)

{parts.Add(part);}

//展示组装完成的电脑

public void Show()

{ Console.WriteLine(”\n 这是一台电脑，它包括：”) ；

foreach(string part in parts)

{Console.Write(part);}

}

}

//定义抽象的Builder

public abstract class Builder

{

//添加主板

public abstract void addMainBoard();

//添加CPU

public abstract void addCPU();

//添加内存

public abstract void addMemory();

//获取组装好的产品

public abstract Computer GetComputer();

}

//具体建造者，建造一台高性能计算机

public class HighPerformanceComputer : Builder

{

private Computer HPcomputer = new Computer();

public override void AddMainBoard()//具体装主板的方法

{HPComputer.add(“高档主板”);}

public override void AddCPU()//具体装CPU的方法

{HPComputer.add(“双核 4.0GHz”);}

public override void AddMemory() //具体装内存的方法

{HPComputer.add(“4GB内存”);}

public override void GetComputer() //具体获得产品的方法

{return HPComputer;}

}

//具体建造者，建造一台普通性能计算机

public class OrdinaryComputer : Builder

{

private Computer Ocomputer = new Computer();

public override void AddMainBoard()//具体装主板的方法

{OComputer.add(“中档主板”);}

public override void AddCPU() //具体装CPU的方法

{OComputer.add(“P4 2.0GHz”);}

public override void AddMemory() //具体装内存的方法

{OComputer.add(“512MB内存”);}

public override void GetComputer() //具体获得产品的方法

{return OComputer;}

}

//定义指挥者类

public class Director

{

public abstract void Construct(Builder builder)

{

builder.AddMainBoard();

builder.AddCPU();

builder.AddMemory();

}

}

//客户端代码

static void Main(string[] args)

{

//创建一个指挥者，用来指挥组装计算机

Director director = new Director();

//高性能计算机的Builder

Builder b1 = new HighPerformanceComputer();

//创建一台高性能计算机

director.Construct(b1);

//获取一台高性能计算机

Computer HighPerformanceComputer = b1.GetComputer();

//展示一台高性能计算机

HighPerformanceComputer.Show();

}

建造者模式的优缺点：

优点：

客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的产品对象。

每一个具体建造者都相对独立，与其他的具体建造者无关，因此可以很方便地替换具体建造者或增加新的具体建造者，扩展方便，符合开闭原则。

可以更加精细地控制产品的创建过程。

缺点：

建造者模式所创建的产品一般具有较多的共同点，其组成部分相似，如果产品之间的差异性很大，不适合使用建造者模式，因此其使用范围受到一定的限制

如果产品的内部变化复杂，可能会需要定义很多具体建造者类来实现这种变化，导致系统变得很庞大，增加了系统的理解难度和运行成本

观察者模式：

例子：

有一个大学毕业生和一个归国留学生都希望能够及时知道求职中心最新的职业需求信息：

目标：Subject

public interface Subject{

public void addObserver(Observer o);

public void deleteObserver(Observer o);

public void notifyObservers( );

}

观察者：Observer

public interface Observer {

public void hearTelephone(String heardMess);

}

具体目标：SeekJobCenter

import java.util.ArrayList;

public class SeekJobCenter implements Subject {

String mess;

boolean changed;

ArrayList<Observer> personList;

SeekJobCenter( ){

personList=new ArrayList<Observer>( );

mess="";

changed=false;

}

public void addObserver(Observer o){

if( ! ( personList.contains(o) ) )

personList.add(o);

}

public void deleteObserver(Observer o){

if(personList.contains(o))

personList.remove(o);

}

public void notifyObservers(){

if(changed){

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

Observer observer=personList.get(i);

observer.hearTelephone(mess); }

changed=false;

}

}

public void giveNewMess(String str){

if(str.equals(mess))

changed=false;

else{ mess=str;

changed=true; }

}

}

具体观察者\_1

import java.io.\*;

public class UniverStudent implements Observer{

Subject subject;

File myFile;

UniverStudent(Subject subject,String fileName){

this.subject=subject;

subject.addObserver(this); //使当前实例成为subject所引用的具体主题的观察者

myFile=new File(fileName);

}

public void hearTelephone(String heardMess){

try{ RandomAccessFile out=new RandomAccessFile(myFile,"rw");

out.seek(out.length());

byte [] b=heardMess.getBytes();

out.write(b); //更新文件中的内容

System.out.print("我是一个大学生,");

System.out.println("我向文件"+myFile.getName()+"写入如下内容:");

System.out.println(heardMess);

}

catch(IOException exp){

System.out.println(exp.toString());

}

}

}

具体观察者\_2：

import java.io.\*;

import java.util.regex.\*;

public class HaiGui implements Observer{

Subject subject;

File myFile;

HaiGui(Subject subject,String fileName){

this.subject=subject;

subject.addObserver(this); //使当前实例成为subject所引用的具体主题的观察者

myFile=new File(fileName);

}

public void hearTelephone(String heardMess){

try{ boolean boo=heardMess.contains("java程序员")||heardMess.contains("软件");

if(boo){

RandomAccessFile out=new RandomAccessFile(myFile,"rw");

out.seek(out.length());

byte [] b=heardMess.getBytes();

out.write(b);

System.out.print("我是一个海归,");

System.out.println("我向文件"+myFile.getName()+"写入如下内容:");

System.out.println(heardMess);

}

else{

System.out.println("我是海归,这次的信息中没有我需要的信息");

}

}

catch(IOException exp){

System.out.println(exp.toString());

}

}

}

应用：

public class Application{

public static void main(String args[]){

SeekJobCenter center=new SeekJobCenter();

UniverStudent zhangLin=

new UniverStudent(center,"A.txt");

HaiGui wangHao=new HaiGui(center,"B.txt");

center.giveNewMess("腾辉公司需要10个java程序员。");

center.notifyObservers();

center.giveNewMess("海景公司需要8个动画设计师。");

center.notifyObservers();

center.giveNewMess("仁海公司需要9个电工。");

center.notifyObservers();

}

观察者模式的优点：

具体目标和具体观察者是松耦合关系。由于目标接口仅仅依赖于观察者接口，因此具体目标只是知道它的观察者是实现观察者接口的某个类的实例，但不需要知道具体是哪个类。同样，由于观察者仅仅依赖于目标接口，因此具体观察者只是知道它依赖的目标是实现目标接口的某个类的实例，但不需要知道具体是哪个类。

中介者模式：

举例

抽象中介者 : MarriageAgency.java

public interface MarriageAgency {

void pair(Person person); // 为person配对

void register(Person person); // 注册会员

}

2．抽象同事类: Person.java

public abstract class Person {

String name; // 姓名

int age; // 年龄

Sex sex; // 性别

int requestAge; // 要求对象的年龄。对对象只有这一个要求

MarriageAgency agency; // 婚姻中介

public Person(String name, int age, Sex sex, int requestAge,

MarriageAgency agency) {

this.name = name;

……

agency.register(this); // 注册会员

}

// 寻找对象

public void findPartner() {

agency.pair(this);

}

}

enum Sex {MALE, FEMALE;}

3．具体中介者 : MarriageAgencyImpl.java

public class MarriageAgencyImpl implements MarriageAgency {

List<Man> men = new ArrayList<Man>(); // 男会员

List<Woman> women = new ArrayList<Woman>(); // 女会员

public void register(Person person) {

if (person.sex == Sex.MALE)

men.add((Man) person);

else if (person.sex == Sex.FEMALE)

women.add((Woman) person);

}

public void pair(Person person) {

if (person.sex == Sex.MALE) {

for (Woman w : women)

if (w.age == person.requestAge) {

System.out.println(person.name + "和" + w.name + "配对成功");

return;

}

} else if (person.sex == Sex.FEMALE) {

for (Man m : men)

if (m.age == person.requestAge) {

System.out.println(person.name + "和" + m.name + "配对成功");

return;

}

}

System.out.println("没有为" + person.name + "找到合适的对象");

}

}

4．具体同事类1 : Man.java

public class Man extends Person {

public Man(String name, int age, int requestAge,

MarriageAgency agency) {

super(name, age, Sex.MALE, requestAge, agency);

}

}

5．具体同事类2: Woman.java

public class Woman extends Person {

public Woman(String name, int age, int requestAge,

MarriageAgency agency) {

super(name, age, Sex.FAMALE, requestAge, agency);

}

}

6．测试类: Test.java

public class Test {

public static void main(String[] args) {

MarriageAgency agency = new MarriageAgencyImpl();

Person m1 = new Man("John", 20, 18, agency);

Person m2 = new Man("Mike", 27, 25, agency);

Person w1 = new Woman("Mary", 25, 27, agency);

Person w2 = new Woman("Jane", 20, 22, agency);

m1.findPartner();

m2.findPartner();

}

}

中介者模式优缺点：

优点：

简化了对象之间的交互

将各同事解耦

减少子类生成

可以简化各同事类的设计和实现

缺点：

在具体中介者类中包含了同事之间的交互细节，可能会导致具体中介者类非常复杂，使得系统难以维护。