实验十四 设计模块（三 ）

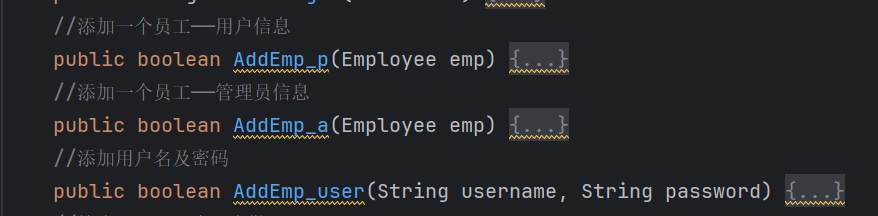
1. 阅读下面设计模式资料（或查阅其它相关资料），结合项目的进程和开发历程，分析项目采用了那些设计模式

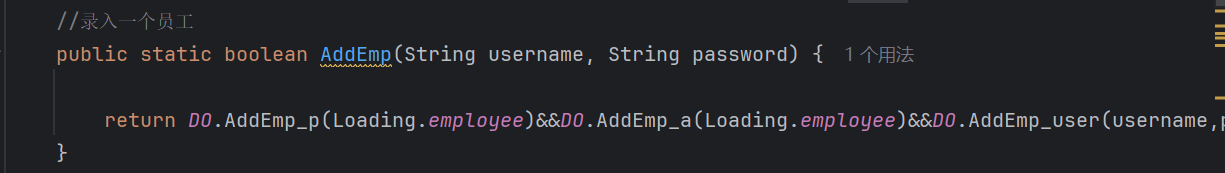
Design Patterns-Elements of Reusable Object-Oriented Software.pdf

The GoF Design Patterns Reference.pdf

[Design Patterns - Wikipedia](https://en.wikipedia.org/wiki/Design_Patterns)

（1）工厂模式：在Data\_Obtain类中，实现了所有的sql语句，而在HRservice中通过调用Data\_Obtain类中封装好的函数来实现前端的数据需求。

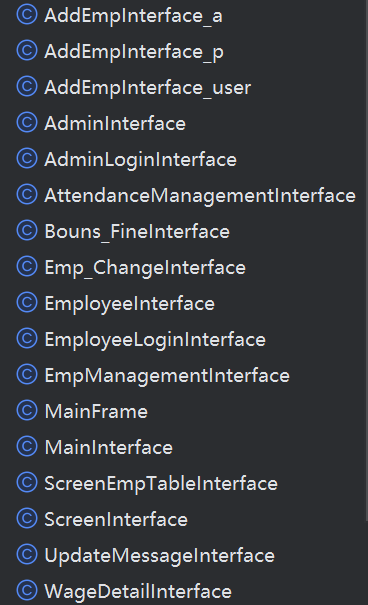




（2）享元模式：下图所示为Loading类，它是一个Flyweight，可以同时在不同上下文使用。



1. 外观模式：

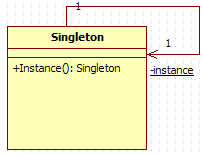


为子系统提供了一个更高层次、更简单的接口，从而降低了子系统的复杂度和依赖。这使得子系统更易于使用和管理。外观承担与子系统中类交互的责任。

（4）迭代子（Iterator）模式：在Loading类中存在大量的HashSet，HashMap需要使用Iterator迭代来遍历寻找合适的对象。

2. 给出4种设计模式的例子（语言不限，以组为单位），并总结其特点 （保存到每个小组选定的协作开发平台上）

1. 单例模式



例子：

如在Java中，Runtime类就是一个单例模式的典型应用。Runtime类用于获取与当前Java应用程序关联的运行时对象，该对象是一个单例实例。通过调用Runtime.getRuntime()方法可以获取该实例，从而使用一些与系统相关的功能，如执行命令行。

单例模式具体实现有两种：

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;

}

}

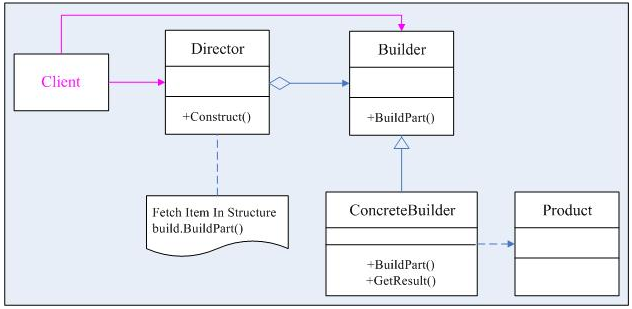
特点：

全局访问：一个类只允许有一个实例，并提供一个全局访问点。

控制资源访问：常用于需要访问某个全局资源的场景，如线程池、数据库连接池等。

线程安全：在多线程环境下，需要确保单例模式的线程安全性。

1. 建造者模式



例子：

如组装一台电脑，组装过程基本不变，可由主板、CPU、内存等按某种固定的方式组装。但主板、CPU、内存等“部件”本身会经常应需求发生变化。

//定义产品角色

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();

}

特点：

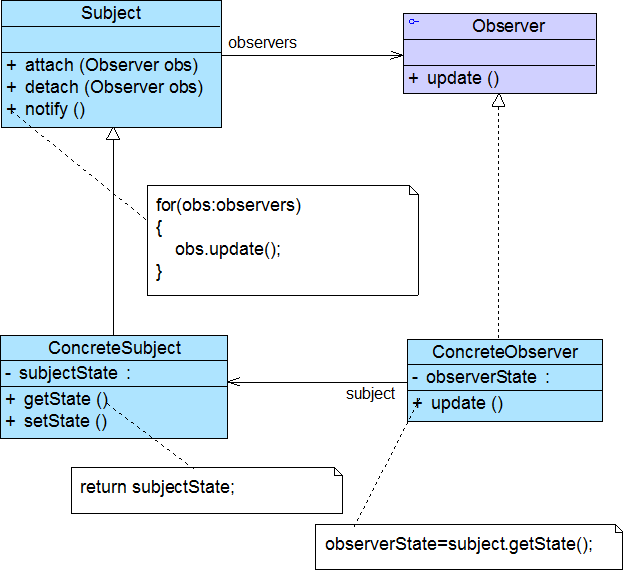
客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的产品对象。

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。

如果产品之间的差异性很大，或是产品的内部变化复杂，不适合使用建造者模式。

1. 观察者模式



例子：

在Android中，广播机制就是观察者模式的应用。通过注册广播接收器，我们可以监听系统或应用程序发出的广播事件。当广播事件发生时，注册的接收器都会收到通知并执行相应的操作。

或是如果有一个大学毕业生和一个归国留学者都希望能及时知道求职中心最新的职业需求信息。

//目标 : Subject.java

public interface Subject{

public void addObserver(Observer o);

public void deleteObserver(Observer o);

public void notifyObservers( );

}

//观察者: Obsever.java

public interface Observer {

public void hearTelephone(String heardMess);

}

//具体目标 SeekJobCenter.java

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 UniversityStudent.java

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());

}

}

}

特点：

一对多依赖：当一个对象状态发生改变时，它的所有依赖对象都会收到通知并自动更新。

解耦：观察者和被观察者之间解耦，使得它们之间的关系更加灵活。

事件驱动：基于事件驱动的设计，使得系统更加易于理解和维护。

1. 适配器模式

例子：

假设我们有一个老旧的数据库访问接口，但现在需要将其与新的系统框架集成。可以使用适配器模式来封装老旧的接口，提供一个新的、与系统框架兼容的接口给客户端使用。

又或者是假设我们有一个老式的电灯（OldLight）和一个新式的智能灯泡（SmartBulb），但我们的智能灯泡控制器（SmartBulbController）只能控制具有特定接口的电灯。

//基础定义及接口：

public interface Light {

void turnOn();

void turnOff();

}

public class OldLight implements Light {

public void turnOn() {

System.out.println("Old Light is on.");

}

public void turnOff() {

System.out.println("Old Light is off.");

}

// 老式电灯特有的方法

public void switchOn() {

this.turnOn();

}

public void switchOff() {

this.turnOff();

}

}

public class SmartBulb {

public void brighten() {

System.out.println("Smart Bulb is brightened.");

}

public void dim() {

System.out.println("Smart Bulb is dimmed.");

}

}

//将SmartBulb适配为Light接口：

public class SmartBulbAdapter implements Light {

private SmartBulb bulb;

public SmartBulbAdapter(SmartBulb bulb) {

this.bulb = bulb;

}

public void turnOn() {

// 适配：将brighten方法视为turnOn

bulb.brighten();

}

public void turnOff() {

System.out.println("Smart Bulb is considered off (but actually dimmed).");

bulb.dim();

}

}

//使用Light接口来控制电灯：

public class SmartBulbController {

private Light light;

public SmartBulbController(Light light) {

this.light = light;

}

public void operateLight() {

light.turnOn();

// 其他操作

light.turnOff();

}

}

特点：

接口转换：将一个类的接口转换成客户端希望的另一个接口，使得原本由于接口不兼容而不能一起工作的类可以一起工作。

复用性：通过适配器，可以复用已有的类或接口，减少代码量。

灵活性：适配器模式使得系统更加灵活，可以方便地集成不同来源的组件或库。

1. 中介者模式

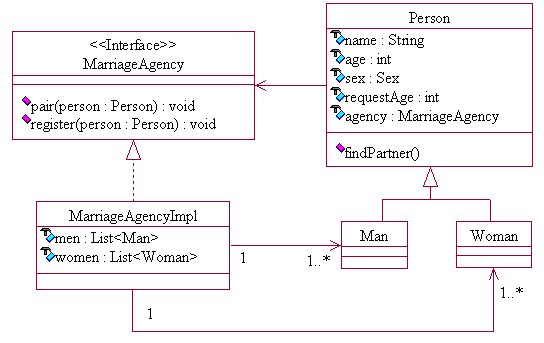


例子：

比如机场调度系统。在机场中，每架飞机都需要根据航班计划进行起降，而跑道、气象等因素都可能影响飞机的起降。机场调度系统作为中介者，负责协调和管理所有飞机、跑道、气象等因素，确保每架飞机都能安全、顺利地起降。

再比如在电商平台中，卖家和买家之间的交流也通过中介模式实现。卖家和买家通过电商平台发送消息、询价、议价、支付等，电商平台作为中介者，负责处理这些交互，确保交易的顺利进行。

以下婚姻机构相关的系统是一个具体的例子：



//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

}

//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;}

//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");

}

}

//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);

}

}

//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);

}

}

特点：

解耦与集中控制：中介模式通过引入中介者对象，将原本相互关联的对象之间的交互集中起来，降低了它们之间的耦合度。

简化交互：在中介模式下，各个对象只需要与中介者对象进行交互，而无需直接与其他对象进行交互。

提高系统灵活性：由于对象之间的交互通过中介者进行，因此可以在不修改对象本身的情况下，通过修改中介者的行为来改变对象之间的交互方式。