#### 简单工厂模式（静态工厂方法模式）

以计算器程序为例：只需输入运算符号，更经常就实例化出合适的对象。通过多态，返回父类的方式实现了计算器的结果。

1. 静态工厂方法统一管理对象的创建。

静态工厂方法通过传入的参数判断决定创建哪一个产品的实例，封装了对象的创建，客户端只管消费，实现了对责任（模块）的分割。

1. 静态工厂方法推迟了产品的实例化。

通过XML配置文件就能改变具体要创建的产品实例，修改为其他的产品实例，代码不须重新编译。

代码：

#include<iostream>

using namespace std;

class Operation{

private:

double numberA;

double numberB;

public:

Operation();

double GetNumberA();

void SetNumberA(double);

double GetNumberB();

void SetNumberB(double);

virtual double GetResult()

{

double result = 0;

return result;

}

};

//函数实现

Operation::Operation()

{

numberA = 0;

numberB = 0;

}

double Operation::GetNumberA()

{

return numberA;

}

void Operation::SetNumberA(double a)

{

numberA = a;

}

double Operation::GetNumberB()

{

return numberB;

}

void Operation::SetNumberB(double b)

{

numberB = b;

}

class OperationAdd : public Operation

{

public:

virtual double GetResult()

{

double result = 0;

result = GetNumberA() + GetNumberB();

return result;

}

};

class OperationSub : public Operation

{

public:

virtual double GetResult()

{

double result = 0;

result = GetNumberA() - GetNumberB();

return result;

}

};

class OperationMul : public Operation

{

public:

virtual double GetResult()

{

double result = 0;

result = GetNumberA() \* GetNumberB();

return result;

}

};

class OperationDiv : public Operation

{

public:

virtual double GetResult()

{

double result = 0;

if(GetNumberB() == 0)

return -1;

result = GetNumberA() / GetNumberB();

return result;

}

};

//工厂类

class OperationFactory

{

public:

//静态工厂方法

static Operation\* createOperate(char c)

{

switch(c)

{

case '+':

return new OperationAdd();

break;

case '-':

return new OperationSub();

break;

case '\*':

return new OperationMul();

break;

case '/':

return new OperationDiv();

break;

}

return NULL;

}

};

void main()

{

OperationFactory operfactory;

Operation \*oper = operfactory.createOperate('+');

oper->SetNumberA(2);

oper->SetNumberB(4);

double result = oper->GetResult();

cout << result <<endl;

}

1. **策略模式**

它定义了一系列的算法，并将每个算法封装起来，而且使他们还可以相互替换。策略模式让算法的变化不会影响到使用算法的客户。

优点：

1. 简化了单元测试，因为每个算法都有自己的类，可以通过自己的接口单独测试。
2. 避免程序中使用多重条件转移语句，使系统更灵活，并易于扩展。
3. 遵守大部分GRASP原则和常用设计原则，高内聚、低耦合。

缺点：

1. 因为每个具体策略类都会产生一个新类，所以会增加系统需要维护的类的数量。
2. 在基本的策略模式中，选择所用具体实现的职责有客户端对象承担，并转给策略模式的Context对象。

进一步优化可以将策略模式与简单工厂模型相结合，选择所用具体实现的职责也可以由Context对象承担，这就最大化的减轻了客户端的职责。

结合策略模型与简单工厂模型实现的简单收银系统代码：

Strategy.h：

#include<iostream>

#include<string>

#include<memory>

using namespace std;

//strategy抽象类，用作接口

class Strategy

{

public:

virtual double GetResult(double money) = 0;

virtual ~Strategy()

{

cout<<" in the destructor of Strategy"<<endl;

}

};

//正常结算方式

class CashNormal : public Strategy

{

double GetResult(double money)

{

return money;

}

~CashNormal()

{

cout<<" in the destructor of CashNormal"<<endl;

}

};

//打折

class CashRebate : public Strategy

{

private:

double moneyRebate; //折扣率

public:

//构造函数

CashRebate(double rebate)

{

moneyRebate = rebate;

}

double GetRebate()

{

return moneyRebate;

}

void SetRebate(double rebate)

{

moneyRebate = rebate;

}

double GetResult(double money)

{

return money \* moneyRebate;

}

};

//返现

class CashReturn : public Strategy

{

//返现的条件与方式

private:

double moneyCondition;

double moneyReturn;

public:

//构造函数

CashReturn(double condition, double re)

{

moneyCondition = condition;

moneyReturn = re;

}

double GetCondition()

{

return moneyCondition;

}

void SetCondition(double condition)

{

moneyCondition = condition;

}

double GetReturn()

{

return moneyReturn;

}

void SetReturn(double re)

{

moneyReturn = re;

}

double GetResult(double money)

{

if(money >= moneyCondition)

{

money = money - (int)(money / moneyCondition) \* moneyReturn;

}

return money;

}

};

//现金收费工厂类

class CashFactory

{

public:

static Strategy\* createGetResult(int type)

{

Strategy\* cs;

switch(type)

{

case 0:

cs = new CashNormal();

break;

case 1:

cs = new CashRebate(0.8);

break;

case 2:

cs = new CashReturn(300,100);

break;

default:

break;

}

return cs;

}

};

//CashContext类 策略模式与简单工厂模式相结合

class CashContext

{

private:

Strategy\* cs;

public:

CashContext(int type)

{

bool loop = true;

while(loop){

switch (type)

{

case 0:

cs =new CashNormal();

loop = false;

break;

case 1:

cs = new CashRebate(0.8);

loop = false;

break;

case 2:

cs = new CashReturn(300,100);

loop = false;

break;

default:

cout<<"输入有误! 请重新输入!"<<endl;

cin>>type;

break;

}

}

}

double GetResult(double money)

{

return cs->GetResult(money);

}

};

Strategy.cpp:

#include "Strategy.h"

void main(int argc, char \*argv)

{

int type = 0;

double total = 0;

cout<<"选择收费方式:"<<endl

<<"0:正常方式"<<endl

<<"1:打折方式"<<endl

<<"2:返现方式"<<endl;

cin >> type;

/\*

CashFactory cfactory;

Strategy \*pay = cfactory.createGetResult(type);

cout<<"输入总的消费金额:";

cin >> total;

cout<<"应收金额为:"<<pay->GetResult(total)<<endl;

\*/

CashContext cs(type);

cout<<"输入总的消费金额:";

cin >> total;

cout<<"应收金额为:"<<cs.GetResult(total)<<endl;

}

1. **单一职责原则(SRP)**

SRP:就一个类而言，应该仅有一个引起它变化的原因。

如果你能够想到多于一个的动机去改变一个类，那么这个类就具有多于一个的职责。

1. **开放-封闭原则**

开放-封闭原则：软件实体（类、模块、函数等等）应该可以扩展，但是不可修改。

面对需求，对程序的改动是通过增加新代码进行的，而不是更改现有的代码。

1. **依赖倒转原则**
2. 高层模块不应该依赖底层模块。两个都应该依赖抽象。
3. 抽象不应该依赖细节。细节应该依赖抽象。

里氏代换原则：一个软件实体如果使用的是一个父类的话，那么一定适用于其子类，而且它察觉不出父类对象和子类对象的区别。也就是说，在软件里面，把父类都替换成它的子类，程序的行为没有变化。（子类型必须能够替换掉它们的父类型）

1. **装饰模式**

装饰模式：动态地给一个对象添加一些额外的职责，就增加功能来说，装饰模式比生成子类更为灵活。



实例（小菜装扮）代码：

decorator.h:

#ifndef \_DECORATOR\_H\_

#define \_DECORATOR\_H\_

#include<iostream>

using namespace std;

//Person类 (ConcreteComponent) 公共抽象类

class Person

{

public:

Person()

{}

Person(char\* name)

{

personname = name;

}

virtual void Show()

{

cout << "装扮的" << personname << endl;

}

private:

char\* personname;

};

//Finery类 (Decorator) 人的具体的属性类 （还可以为兴趣爱好） 为公共抽象类的一种实例

class Finery : public Person

{

protected:

Person\* component;

public:

void Decorate(Person\* comp)

{

component = comp;

}

void Show()

{

component->Show();

}

};

//具体装饰类

class TShirts : public Finery

{

public:

void Show()

{

cout << "大T恤" <<" ";

Finery::Show(); //调用父类Show()函数

}

};

class BigTrouser : public Finery

{

public:

void Show()

{

cout << "垮裤" << " ";

Finery::Show();

}

};

class WearSneakers : public Finery

{

public:

void Show()

{

cout << "破球鞋" << " ";

Finery::Show();

}

};

class WearSuit : public Finery

{

public:

void Show()

{

cout << "西装" << " ";

Finery::Show();

}

};

class WearTie : public Finery

{

public:

void Show()

{

cout << "领带" << " ";

Finery::Show();

}

};

class WearLeatherShoes : public Finery

{

public:

void Show()

{

cout << "皮鞋" << " ";

Finery::Show();

}

};

#endif

decorator.cpp:

#include "decorator.h"

void main(int argc, char \*aegv)

{

char \*name = "小菜";

Person \*xc = new Person(name);

cout << "\n第一种装扮: " << endl;

WearSneakers \*pqx = new WearSneakers();

BigTrouser \*kk = new BigTrouser();

TShirts \*dtx = new TShirts();

pqx->Decorate(xc);

kk->Decorate(pqx);

dtx->Decorate(kk);

dtx->Show();

cout << "\n第二种装扮:" << endl;

WearLeatherShoes \*px = new WearLeatherShoes();

WearTie \*ld = new WearTie();

WearSuit \*xz = new WearSuit();

px->Decorate(xc);

ld->Decorate(px);

xz->Decorate(ld);

dtx->Decorate(xz);

dtx->Show();

delete pqx;

delete kk;

delete dtx;

delete px;

delete ld;

delete xz;

}

1. **代理模式**

代理模式：为其他对象提供一种代理以控制对这个对象的访问。

代理模式的场合：

* 远程代理：为一个对象在不同的地址空间提供局部代表。这样可以隐藏一个对象存在于不同地址空间的事实。
* 虚拟代理：根据需要创建开销很大的对象通过它来存放实例化需要很长时间的真实对象。
* 安全代理：用来控制真实对象访问时的权限。
* 智能指引：当调用真实的对象时，代理处理另外一些事。

送礼物实例代码：

proxy.h:

#ifndef \_PROXY\_H\_

#define \_PROXY\_H\_

#include<iostream>

using namespace std;

class SchoolGirl

{

private:

char \* name;

public:

char \* GetName()

{

return name;

}

void SetName(char \* n)

{

name = n;

}

};

//代理接口

class IGiveGift

{

public:

virtual void GiveDolls() = 0;

//virtual void GiveFlowers() = 0;

//virtual void GiveChocolate() = 0;

};

class Persuit : public IGiveGift

{

private:

SchoolGirl \* mm;

public:

Persuit(SchoolGirl\* m)

{

mm = m;

}

void GiveDolls()

{

cout << mm->GetName() << "送你洋娃娃" << endl;

}

};

class Proxy : public IGiveGift

{

private:

Persuit \* gg;

public:

Proxy(SchoolGirl\* mm)

{

gg = new Persuit(mm);

}

void GiveDolls()

{

gg->GiveDolls();

}

};

#endif

Proxy.cpp:

#include "proxy.h"

void main(int argc, char\* argv)

{

SchoolGirl \*jiaojiao = new SchoolGirl();

jiaojiao->SetName("李娇娇");

Proxy \*daili = new Proxy(jiaojiao);

daili->GiveDolls();

}

1. **工厂方法模式**

工厂方法模式：定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法使一个类的实例化延迟到其子类。

简单工厂模式在添加新的功能时需要修改工厂类 ，这违反了开放-封闭原则。

学雷锋实例代码：

factorymethod.h:

#ifndef \_FACTORYMETHOD\_H\_

#define \_FACTORYMETHOD\_H\_

#include<iostream>

using namespace std;

class LeiFeng

{

public:

void Sweep()

{

cout << "扫地" << endl;

}

void Wash()

{

cout << "洗衣" << endl;

}

void BuyRice()

{

cout << "买米" << endl;

}

};

class Undergraduate : public LeiFeng

{

};

class Volunteer : public LeiFeng

{

};

//雷锋工厂 抽象类

class IFactory

{

public:

virtual LeiFeng \* CreateLeiFeng() = 0;

};

//学雷锋的大学生工厂

class UndergraduateFactory : public IFactory

{

public:

LeiFeng \* CreateLeiFeng()

{

return new Undergraduate();

}

};

//社区志愿者工厂

class VolunteerFactory : IFactory

{

public:

LeiFeng \* CreateLeiFeng()

{

return new Volunteer();

}

};

#endif

factorymethod.cpp:

#include "factorymethod.h"

void main(int argc, char\* argv)

{

IFactory \* factory = new UndergraduateFactory();

LeiFeng \* studentA = factory->CreateLeiFeng();

studentA->BuyRice();

delete studentA;

}

1. **原型模式**

原型模式：用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

拷贝构造函数：复制一个对象分为浅拷贝和深拷贝。

浅拷贝： 给一个对象中的每个成员变量进行复制，就是把A1类中的变量直接赋给A2类中的变量，属于值传递，但是涉及到有new之类内存分配的地方，两个对象是共享内存的。即，两个对象同时指向同一内存空间。

深拷贝：不仅使用值传递，而且每个变量都有自己一份独立的内存空间，互不干扰。

常用场景：当你需要从A的实例得到一份与A内容相同但是又互不干扰的实例的话，就需要使用原型模式。

复制建立实例的源代码：

Prototype.h:

#ifndef \_PROTOTYPE\_H\_

#define \_PROTOTYPE\_H\_

#include<iostream>

#include<cstring>

using namespace std;

//抽象基类

class resume

{

protected:

char\* name;

public:

resume()

{

}

virtual ~resume()

{

}

virtual void set(const char \*str)

{

}

virtual void show()

{

}

virtual resume\* clone()

{

return 0;

}

};

class ResumeA : public resume

{

public:

ResumeA(const char \*str); //构造函数

ResumeA(const ResumeA &r); //拷贝构造函数

~ResumeA(); //析构函数

resume \* clone(); //克隆，关键所在

void show(); //显示内容

};

class ResumeB : public resume

{

public:

ResumeB(const char \*str); //构造函数

ResumeB(const ResumeB &r); //拷贝构造函数

~ResumeB(); //析构函数

resume\* clone(); //克隆，关键所在

void show(); //显示内容

};

//函数实现

ResumeA::ResumeA(const char \*str)

{

if(str == NULL)

{

name = new char[1];

name[0] = '\0';

}else

{

name = new char[strlen(str) + 1];

strcpy(name, str);

}

}

ResumeA::~ResumeA()

{

delete [] name;

}

ResumeA::ResumeA(const ResumeA &r)

{

name = new char[strlen(r.name) + 1];

strcpy(name, r.name);

}

resume\* ResumeA::clone()

{

return new ResumeA(\*this);

}

void ResumeA::show()

{

cout << "ResumeA name ：" << name << endl;

}

//ResumeB

ResumeB::ResumeB(const char \*str)

{

if(str == NULL) {

name = new char[1];

name[0] = '\0';

}

else {

name = new char[strlen(str)+1];

strcpy(name, str);

}

}

ResumeB::~ResumeB() { delete [] name;}

ResumeB::ResumeB(const ResumeB &r) {

name = new char[strlen(r.name)+1];

strcpy(name, r.name);

}

resume\* ResumeB::clone() {

return new ResumeB(\*this);

}

void ResumeB::show() {

cout<<"ResumeB name : "<<name<<endl;

}

#endif

prototype.cpp:

#include "prototype.h"

void main()

{

resume \*r1 = new ResumeA("A");

resume \*r2 = new ResumeB("B");

resume \*r3 = r1->clone();

resume \*r4 = r2->clone();

r1->show();

r2->show();

//删除r1,r2

delete r1;

delete r2;

r1 = r2 =NULL; //指针释放后清零，可以防止“野指针”

//深拷贝所以对r3,r4没有影响

r3->show();

r4->show();

delete r3;

delete r4;

r3 = r4 = NULL;

}

1. **模板方法模式**

模板方法模式：定一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

使用情景：当不变的和可变的行为在方法的子类实现中混合在一起的时候，不变的行为就会在子类中重复出现。通过模板方法模式把这些行为搬移到单一的地方，这样就帮助子类摆脱重复的不变行为的纠缠。

实例程序源代码：

Templetemethod.h:

#ifndef \_TEMPLETEMETHOD\_H\_

#define \_TEMPLETEMETHOD\_H\_

#include<iostream>

using namespace std;

class AbstractClass

{

public:

virtual void PrimitiveOperation1() = 0;

virtual void PrimitiveOperation2() = 0;

void TemplateMethod()

{

PrimitiveOperation1();

PrimitiveOperation2();

cout << "其他操作" << endl;

}

};

class ConcreteClassA : public AbstractClass

{

public:

void PrimitiveOperation1()

{

cout << "具体类A方法1实现" << endl;

}

void PrimitiveOperation2()

{

cout << "具体类A方法2实现" << endl;

}

};

class ConcreteClassB : public AbstractClass

{

public:

void PrimitiveOperation1()

{

cout << "具体类B方法1实现" << endl;

}

void PrimitiveOperation2()

{

cout << "具体类B方法2实现" << endl;

}

};

#endif

templetemethod.cpp:

#include "templetemethod.h"

void main()

{

AbstractClass \*c;

c = new ConcreteClassA();

c->TemplateMethod();

c = new ConcreteClassB();

c->TemplateMethod();

}

1. **迪米特法则**

迪米特法则：如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用。如果其中一个类需要调用另一个类的某一个方法的话，可以通过第三者转发这个调用。

1. **外观模式**

外观模式：为子系统中的一组接口提供一个一致的界面，此模式定义了一个高层接口，这个接口使得这一个子系统更加容易使用。

代码结构示例：

facede.h:

#ifndef \_FACEDE\_H\_

#define \_FACEDE\_H\_

#include<iostream>

using namespace std;

class SubSystemOne

{

public:

void MethodOne()

{

cout << "子系统方法一" << endl;

}

};

class SubSystemTwo

{

public:

void MethodTwo()

{

cout << "子系统方法二" << endl;

}

};

class SubSystemThree

{

public:

void MethodThree()

{

cout << "子系统方法三" << endl;

}

};

class SubSystemFour

{

public:

void MethodFour()

{

cout << "子系统方法四" << endl;

}

};

//外观类

class Facede

{

private:

SubSystemOne \*one;

SubSystemTwo \*two;

SubSystemThree \*three;

SubSystemFour \*four;

public:

Facede()

{

one = new SubSystemOne();

two = new SubSystemTwo();

three = new SubSystemThree();

four = new SubSystemFour();

}

void MethodA()

{

cout << "\n方法组A()----" << endl;

one->MethodOne();

two->MethodTwo();

four->MethodFour();

}

void MethodB()

{

cout << "\n方法组B()----" << endl;

two->MethodTwo();

three->MethodThree();

}

};

#endif

facede.cpp:

#include "facede.h"

void main()

{

Facede \*facade = new Facede();

facade->MethodA();

facade->MethodB();

}

1. **建造者模式**

建造者模式：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

建造者模式实例源代码:

builder.h:

#ifndef \_BUILDER\_H\_

#define \_BUILDER\_H\_

#include<iostream>

#include<cstring>

using namespace std;

class Product

{

public:

int p;

public:

Product()

{

p = 0;

}

void Add(int n)

{

p = p + n;

}

void show()

{

cout << "\n产品 创建----" << endl;

cout << p << endl;

}

};

class Builder

{

public:

virtual void BuilderPartA() = 0;

virtual void BuilderPartB() = 0;

virtual Product \* Getresult() = 0;

};

class ConcreteBuilder1 : public Builder

{

private:

Product \* product;

public:

ConcreteBuilder1()

{

product = new Product();

}

void BuilderPartA()

{

product->Add(1);

}

void BuilderPartB()

{

product->Add(2);

}

Product \* Getresult()

{

return product;

}

};

class ConcreteBuilder2 : public Builder

{

private:

Product \* product;

public:

ConcreteBuilder2()

{

product = new Product();

}

void BuilderPartA()

{

product->Add(3);

}

void BuilderPartB()

{

product->Add(4);

}

Product \* Getresult()

{

return product;

}

};

class Director

{

public:

void Construct(Builder\* builder)

{

builder->BuilderPartA();

builder->BuilderPartB();

}

};

#endif

builder.cpp:

#include "builder.h"

void main()

{

Director \* dir = new Director();

Builder \* b1 = new ConcreteBuilder1();

Builder \* b2 = new ConcreteBuilder2();

dir->Construct(b1);

Product \* p1 = b1->Getresult();

p1->show();

dir->Construct(b2);

Product \* p2 = b2->Getresult();

p2->show();

}

1. **观察者模式**

观察者模式：定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态发生变化时，会通知所有观察者对象，使他们能够自动更新自己。又叫做发布-订阅模式。

Subject类:它把所有对观察者对象的引用保存在一个聚类里，每个主题都可以有任何数量的观察者。抽象主题提供一个接口，可以增加和删除观察者对象。

Observer类：抽象观察者，为所有的具体观察者定义一个接口，在得到主题的通知时更新自己。

ConcreteSubject类：具体主题，将有相关状态存入具体观察者对象；在具体主题的内部状态改变时，给所有登记过的观察者发出通知。

ConcreteObserser类：具体观察者类，实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题的状态相协调。

源代码：

Observer.h:

#pragma once

#include<iostream>

#include<string>

#include<list>

using namespace std;

class Subject;

class Observer

{

public:

~Observer();

virtual void Update(Subject\*) = 0;

protected:

Observer();

private:

};

class ConcreteObserverA : public Observer

{

public:

ConcreteObserverA();

~ConcreteObserverA();

virtual void Update(Subject\*);

private:

string m\_state;

};

class ConcreteObserverB :public Observer

{

public:

ConcreteObserverB();

~ConcreteObserverB();

virtual void Update(Subject\*);

private:

string m\_state;

};

class Subject

{

public:

~Subject();

virtual void Notify();

virtual void Attach(Observer\*);

virtual void Detach(Observer\*);

virtual string GetState();

virtual void SetState(string state);

protected:

Subject();

private:

string m\_state;

list<Observer\*> m\_lst;

};

class ConcreteSubjectA : public Subject

{

public:

ConcreteSubjectA();

~ConcreteSubjectA();

protected:

private:

};

class ConcreteSubjectB : public Subject

{

public:

ConcreteSubjectB();

~ConcreteSubjectB();

protected:

private:

};

//实现

Observer::Observer()

{

}

Observer::~Observer()

{

}

ConcreteObserverA::ConcreteObserverA()

{

}

ConcreteObserverA::~ConcreteObserverA()

{

}

void ConcreteObserverA::Update(Subject\* pSubject)

{

this->m\_state = pSubject->GetState();

cout << "This ConcreteObserverA is" << m\_state << std::endl;

}

ConcreteObserverB::ConcreteObserverB()

{

}

ConcreteObserverB::~ConcreteObserverB()

{

}

void ConcreteObserverB::Update(Subject\* pSubject)

{

this->m\_state = pSubject->GetState();

cout << "This ConcreteObserverB is" << m\_state << std::endl;

}

Subject::Subject()

{

}

Subject::~Subject()

{

}

void Subject::Attach(Observer\* pObserver)

{

this->m\_lst.push\_back(pObserver);

cout << "Attach an Observer\n";

}

void Subject::Detach(Observer\* pObserver)

{

list<Observer\*>::iterator iter;

iter = find(m\_lst.begin(), m\_lst.end(), pObserver);

if (iter != m\_lst.end())

{

m\_lst.erase(iter);

}

cout << "Detach an Observer\n";

}

void Subject::Notify()

{

list<Observer\*>::iterator iter = this->m\_lst.begin();

for (; iter != m\_lst.end(); iter++)

{

(\*iter)->Update(this);

}

}

string Subject::GetState()

{

return this->m\_state;

}

void Subject::SetState(string state)

{

this->m\_state = state;

}

ConcreteSubjectA::ConcreteSubjectA()

{

}

ConcreteSubjectA::~ConcreteSubjectA()

{

}

ConcreteSubjectB::ConcreteSubjectB()

{

}

ConcreteSubjectB::~ConcreteSubjectB()

{

}

Observer.cpp:

#include "observer.h"

int main()

{

Observer\* p1 = new ConcreteObserverA();

Observer\* p2 = new ConcreteObserverB();

Observer\* p3 = new ConcreteObserverA();

Subject\* pSubject = new ConcreteSubjectA();

pSubject->Attach(p1);

pSubject->Attach(p2);

pSubject->Attach(p3);

pSubject->SetState(" old");

pSubject->Notify();

cout << "-----------------------------------------------------------" << endl;

pSubject->SetState(" new");

pSubject->Detach(p3);

pSubject->Notify();

return 0;

}

使用观察者模式的动机：将一个系统分割成一系列相互协作的类有一个很不好的副作用，那就是需要维护相关对象见的一致性。我们不希望为了维持一致性而使各类紧密耦合，这样会给维护、扩展和重用都带来不便。

1. **抽象工厂模式**

抽象工厂模式：提供一个创建一系列相关或相互依赖的对象的接口，而无需指定它们具体的类。

源代码：

abstractfactory.h:

#pragma once

/\*

抽象工厂模式简介:一个工厂接口，多个产品接口

这里我们以数据库的维护为例，假设现在有一个工程项目不一定会用到

Access和SqlServer这两种数据库的哪一种，但是此工程中数据库的相关表已经被定下

因为对Access和Sqlserver的操作并不相同，此时为了维护方便我们可以使用抽象工厂模式

\*/

#include<iostream>

using namespace std;

/\*用户\*/

class User

{

};

/\*部门\*/

class Department

{

};

//定义User表的抽象接口

class IUser

{

public:

//向Iuser表中插入一个user

virtual void Insert(User\* user) = 0;

virtual User\* GetUser(int id) = 0; //根据id获取user

};

//Access中的User表（具体）

class AccessUser : public IUser

{

public:

void Insert(User \*user)

{

cout << "在Access数据库中的IUser表中插入一个User（记录）" << endl;

}

User\* GetUser(int id)

{

cout << "从Access数据库的IUser表中根据id获取一条记录" << endl;

return NULL;

}

};

class SqlserverUser : public IUser

{

public:

void Insert(User \*user)

{

cout << "在Sqlserver数据库中的IUser表中插入一个User(记录)" << endl;

}

User\* GetUser(int id)

{

cout << "从Sqlserver数据库的IUser表中根据id获取一条记录" << endl;

return NULL;

}

};

//定义Department表的抽象接口

class IDepartment

{

public:

virtual void Insert(Department\* department) = 0;

virtual Department \*GetDepartment(int id) = 0;

};

//Access数据库中的Department表

class AccessDepartment : public IDepartment

{

public:

void Insert(Department\* department)

{

cout << "在Access数据库的IDepartment表中插入一条记录" << endl;

}

Department\* GetDepartment(int id)

{

cout << "从Access数据库的IDpartment表根据id获取一条记录" << endl;

return NULL;

}

};

//Sqlserver数据库中的Department表

class SqlserverDepartment : public IDepartment

{

public:

void Insert(Department\* department)

{

cout << "在Sqlserver数据库的IDepartment表中插入一条记录" << endl;

}

Department\* GetDepartment(int id)

{

cout << "从Sqlserver数据库的IDepartment表根据id获取一条记录" << endl;

return NULL;

}

};

//工厂接口

class IFactory

{

public:

virtual IUser \* CreateUser() = 0;

virtual IDepartment \* CreateDepartment() = 0;

};

//Access工厂类

class AccessFactory : public IFactory

{

public:

IUser \* CreateUser()

{

return new AccessUser();

}

IDepartment \* CreateDepartment()

{

return new AccessDepartment();

}

};

//sqlserver工厂类

class SqlserverFactory : public IFactory

{

public:

IUser \* CreateUser()

{

return new SqlserverUser();

}

IDepartment \* CreateDepartment()

{

return new SqlserverDepartment();

}

};

abstractfactory.cpp:

#include"abstractfactory.h"

void main()

{

//建立一个Access工厂

IFactory \* sqlserverFactory = new SqlserverFactory();

IUser \* user = sqlserverFactory->CreateUser();

user->Insert(new User());

user->GetUser(0);

/\*使用这种设计模式的好处是，

1.当你需要更换数据库或时只需要更改一个地方

2.当你需要对其他表进行操作时也只需要改一个地方

可维护性大大提高

\*/

}

1. **状态模式**

状态模式：