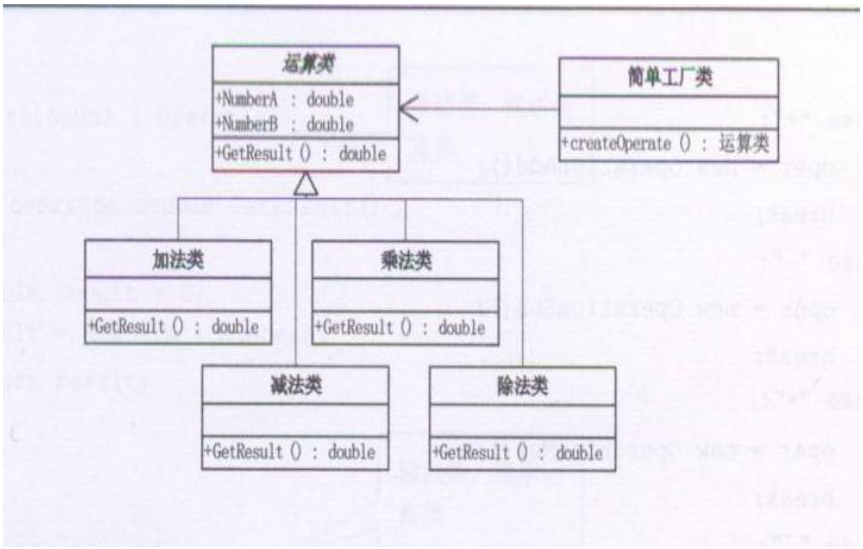
（一）简单工厂模式

主要用于创建对象。新添加类时，不会影响以前的系统代码。核心思想是用一个工厂来根据输入的条件产生不同的类，然后根据不同类的 virtual 函数得到不同的结果。

GOOD:适用于不同情况创建不同的类时

BUG：客户端必须要知道基类和工厂类，耦合性差



（工厂类与基类为关联关系）

例：

```
//基类
class COperation
{
public:
	int m_nFirst;
	int m_nSecond;
	virtual double GetResult()
	{
		double dResult=0;
		return dResult;
	}
};
//加法
class AddOperation : public COperation
{
public:
	virtual double GetResult()
	{
		return m_nFirst+m_nSecond;
	}
```

```
};
//减法
class SubOperation : public COperation
{
public:
	virtual double GetResult()
	{
		return m_nFirst-m_nSecond;
	}
};

//工厂类
class CCalculatorFactory
{
public:
	static COperation* Create(char cOperator);
};

COperation* CCalculatorFactory::Create(char cOperator)
{
	COperation *oper;
	//在 C#中可以用反射来取消判断时用的 switch，在 C++中用什么呢？RTTI？？
	switch (cOperator)
	{
	case '+':
		oper=new AddOperation();
		break;
	case '-':
		oper=new SubOperation();
		break;
	default:
		oper=new AddOperation();
		break;
	}
	return oper;
}
```

客户端

```
int main()
{
	int a,b;
	cin>>a>>b;
	COperation * op=CCalculatorFactory::Create('-');
	op->m_nFirst=a;
```

```
        op->m_nSecond=b;
        cout<<op->GetResult()<<endl;
        return 0;
}
```

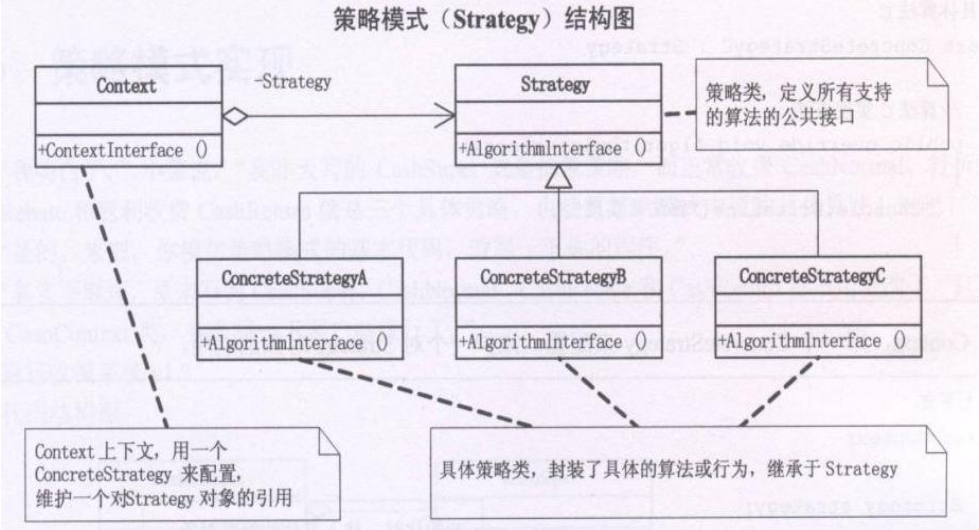
（二）策略模式

定义算法家族，分别封装起来，让它们之间可以互相替换，让算法变化，不会影响到用户

GOOD:适合类中的成员以方法为主，算法经常变动；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。

策略模式和简单工厂基本相同，但简单工厂模式只能解决对象创建问题，对于经常变动的算法应使用策略模式。

BUG:客户端要做出判断



例

```
//策略基类
class COperation
{
public:
        int m_nFirst;
        int m_nSecond;
        virtual double GetResult()
        {
                double dResult=0;
                return dResult;
        }
};

//策略具体类—加法类
class AddOperation : public COperation
{
```

```
public:
    AddOperation(int a,int b)
    {
        m_nFirst=a;
        m_nSecond=b;
    }
    virtual double GetResult()
    {
        return m_nFirst+m_nSecond;
    }
};

class Context
{
private:
    COperation* op;
public:
    Context(COperation* temp)
    {
        op=temp;
    }
    double GetResult()
    {
        return op->GetResult();
    }
};

//客户端
int main()
{
    int a,b;
    char c;
    cin>>a>>b;
    cout<<"请输入运算符：;
    cin>>c;
    switch(c)
   {
      case '+':
        Context *context=new Context(new AddOperation(a,b));
        cout<<context->GetResult()<<endl;
       break;
      default:
       break;
  }
```

```
    return 0;
}
```

策略与工厂结合

GOOD:客户端只需访问 Context 类，而不用知道其它任何类信息，实现了低耦合。

在上例基础上，修改下面内容

```
class Context
{
private:
    COperation* op;
public:
    Context(char cType)
    {
        switch (cType)
        {
        case '+':
            op=new AddOperation(3,8);
            break;
        default:
            op=new AddOperation();
            break;
        }
    }
    double GetResult()
    {
        return op->GetResult();
    }
};
//客户端
int main()
{
    int a,b;
    cin>>a>>b;
    Context *test=new Context('+');
    cout<<test->GetResult()<<endl;
    return 0;
}
```

单一职责原则

就一个类而言，应该仅有一个引起它变化的原因。

如果一个类承担的职责过多，就等于把这些职责耦合在一起，一个职责的变化可能会削弱或者抑制这个类完成其它职责能力。这种耦合会导制脆弱的设计，当变化发生时，设计会遭受到意想不到的破坏。

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。

开放－－封闭原则

软件实体可以扩展，但是不可修改。即对于扩展是开放的，对于修改是封闭的。面对需求，对程序的改动是通过增加代码来完成的，而不是改动现有的代码。

当变化发生时，我们就创建抽象来隔离以后发生同类的变化。

开放－－封闭原则是面向对象的核心所在。开发人员应该对程序中呈现出频繁变化的那部分做出抽象，拒绝对任何部分都刻意抽象及不成熟的抽象。

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，把父类替换成子类，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。

依赖倒转原则

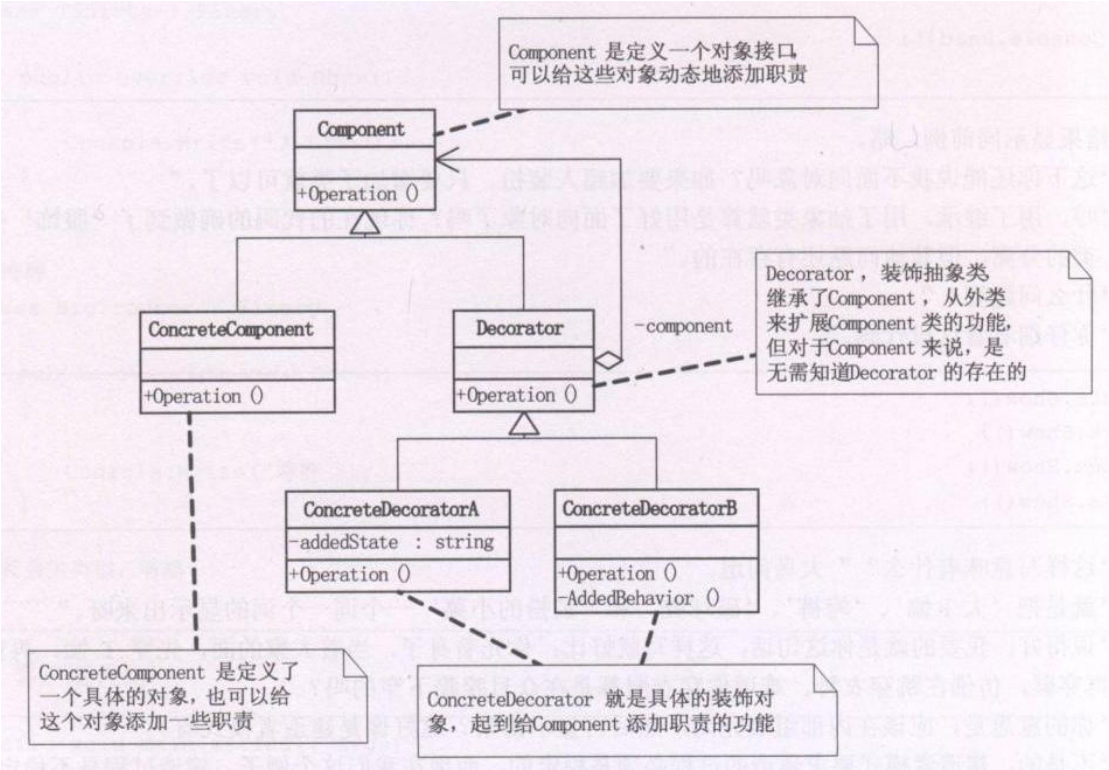
抽象不应该依赖细节，细节应该依赖抽象。即针对接口编程，不要对实现编程。

高层模块不能依赖低层模块，两者都应依赖抽象。

依赖倒转原则是面向对象的标志，用哪种语言编写程序不重要，如果编写时考虑的是如何针对抽象编程而不是针对细节编程，即程序的所有依赖关系都终止于抽象类或接口。那就是面向对象设计，反之那就是过程化设计。

（三）装饰模式

动态地给一个对象添加一些额外的职责（不重要的功能，只是偶然一次要执行），就增加功能来说，装饰模式比生成子类更为灵活。建造过程不稳定，按正确的顺序串联起来进行控制。



GOOD:当你向旧的类中添加新代码时，一般是为了添加核心职责或主要行为。而当需要加入的仅仅是一些特定情况下才会执行的特定的功能时（简单点就是不是核心应用的功能），就会增加类的复杂度。装饰模式就是把要添加的附加功能分别放在单独的类中，并让这个类包含它要装饰的对象，当需要执行时，客户端就可以有选择地、按顺序地使用装饰功

能包装对象。

例

```
#include <string>
#include <iostream>
using namespace std;
//人
class Person
{
private:
	string m_strName;
public:
	Person(string strName)
	{
		m_strName=strName;
	}
	Person(){}
	virtual void Show()
	{
		cout<<"装扮的是:"<<m_strName<<endl;
	}

};
//装饰类
class Finery :public Person
{
protected:
	Person* m_component;
public:
	void Decorate(Person* component)
	{
		m_component=component;
	}
	virtual void Show()
	{
			m_component->Show();
	}
};
//T 恤
class TShirts: public Finery
{
public:
	virtual void Show()
	{
```

```
            cout<<"T Shirts"<<endl;
            m_component->Show();
        }
};
//裤子
class BigTrouser :public    Finery
{
public:
        virtual void Show()
        {
            cout<<" Big Trouser"<<endl;
            m_component->Show();
        }
};

//客户端
int main()
{
        Person *p=new Person("小李");
        BigTrouser *bt=new BigTrouser();
        TShirts *ts=new TShirts();

        bt->Decorate(p);
        ts->Decorate(bt);
        ts->Show();
        return 0;
}
```

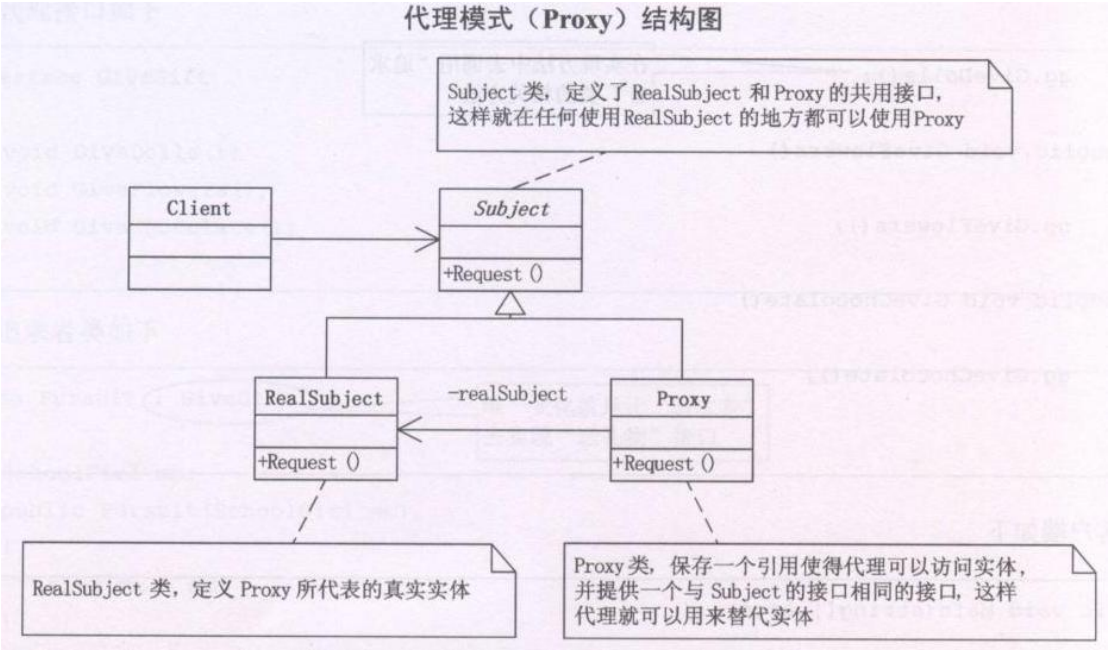
（四）代理模式

GOOD：远程代理，可以隐藏一个对象在不同地址空间的事实

虚拟代理：通过代理来存放需要很长时间实例化的对象

安全代理：用来控制真实对象的访问权限

智能引用：当调用真实对象时，代理处理另外一些事



例：

```
#include <string>
#include <iostream>
using namespace std;
//定义接口
class Interface
{
public:
	virtual void Request()=0;
};
//真实类
class RealClass : public Interface
{
public:
	virtual void Request()
	{
		cout<<"真实的请求"<<endl;
	}
};
//代理类
class ProxyClass : public Interface
{
private:
	RealClass* m_realClass;
public:
	virtual void Request()
	{
```

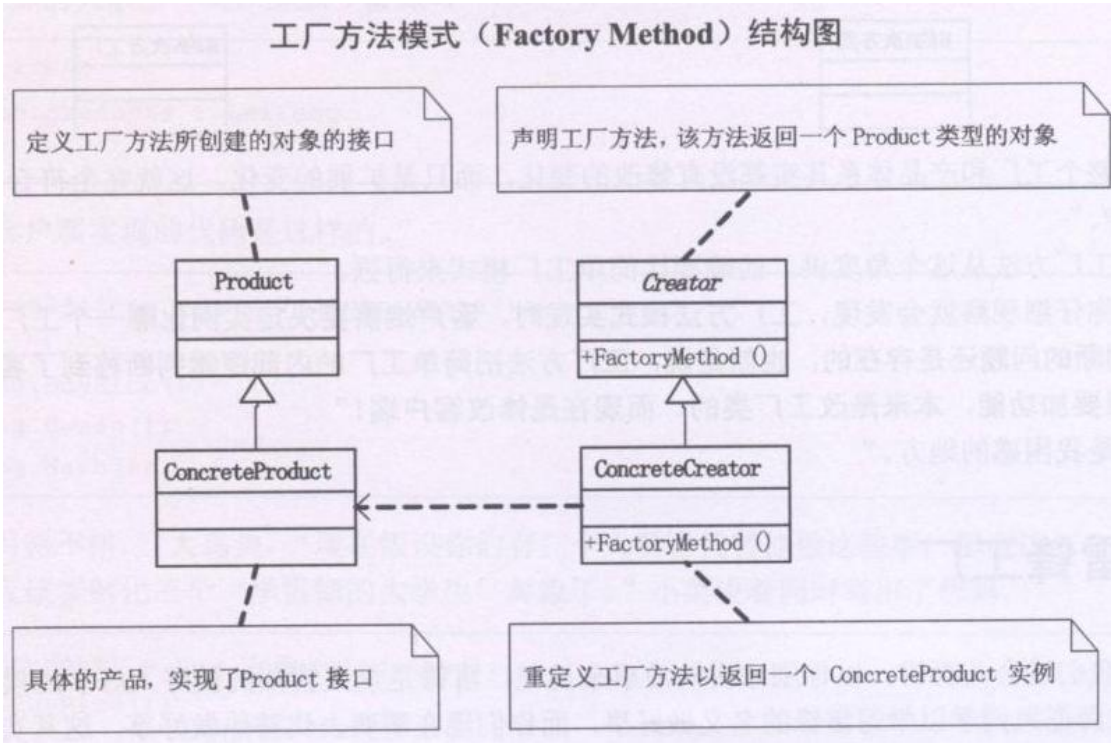
```
            m_realClass= new RealClass();
            m_realClass->Request();
            delete m_realClass;
        }
};
```

客户端：

```
int main()
{
    ProxyClass* test=new ProxyClass();
    test->Request();
    return 0;
}
```

（五）工厂方法模式

GOOD：修正了简单工厂模式中不遵守开放－封闭原则。工厂方法模式把选择判断移到了客户端去实现，如果想添加新功能就不用修改原来的类，直接修改客户端即可。



例：

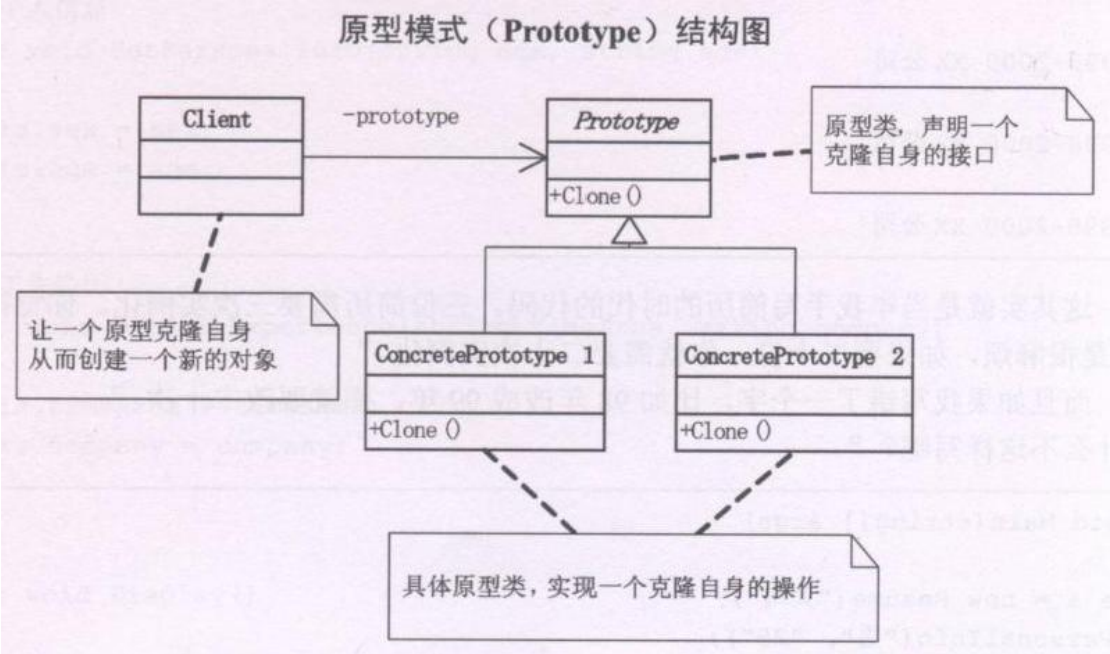
```
#include <string>
#include <iostream>
using namespace std;
//实例基类，相当于 Product(为了方便，没用抽象)
class LeiFeng
{
public:
```

```
    virtual void Sweep()
    {
        cout<<"雷锋扫地"<<endl;
    }
};

//学雷锋的大学生，相当于 ConcreteProduct
class Student: public LeiFeng
{
public:
    virtual void Sweep()
    {
        cout<<"大学生扫地"<<endl;
    }
};

//学雷锋的志愿者，相当于 ConcreteProduct
class Volenter: public LeiFeng
{
public :
    virtual void Sweep()
    {
        cout<<"志愿者"<<endl;
    }
};
//工场基类 Creator
class LeiFengFactory
{
public:
    virtual LeiFeng* CreateLeiFeng()
    {
        return new LeiFeng();
    }
};
//工场具体类
class StudentFactory : public LeiFengFactory
{
public :
    virtual LeiFeng* CreateLeiFeng()
    {
        return new Student();
    }
};
class VolenterFactory : public LeiFengFactory
```

```
{
public:
    virtual LeiFeng* CreateLeiFeng()
    {
        return new Volenter();
    }
};
//客户端
int main()
{
    LeiFengFactory *sf=new LeiFengFactory();
    LeiFeng *s=sf->CreateLeiFeng();
    s->Sweep();
    delete s;
    delete sf;
    return 0;
}
```

（六）原型模式

GOOD：从一个对象再创建另外一个可定制的对象，而无需知道任何创建的细节。并能提高创建的性能。 说白了就 COPY 技术，把一个对象完整的 COPY 出一份。



例：

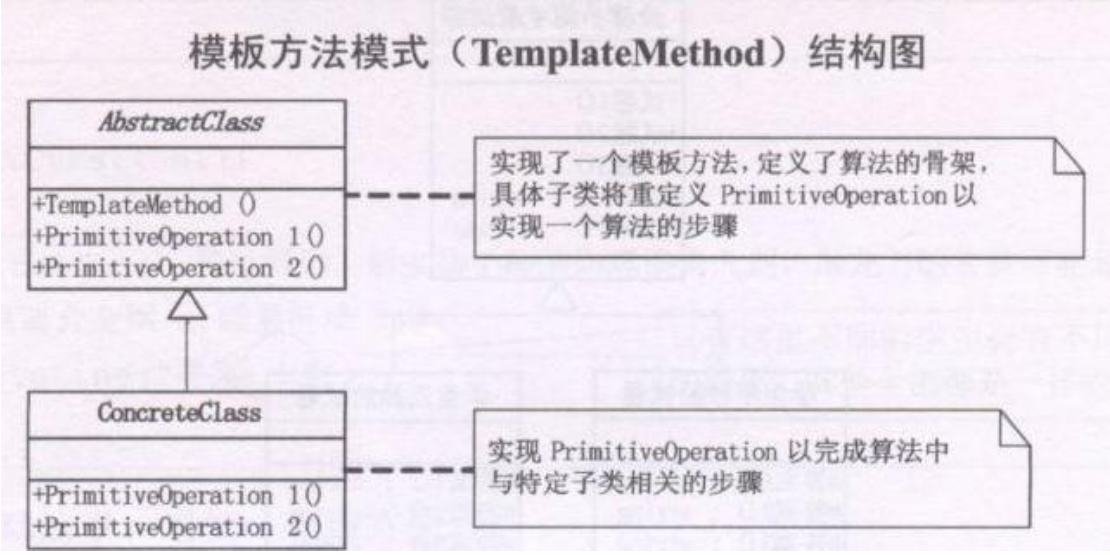
```
#include<iostream>
#include <vector>
#include <string>
using namespace std;
```

```
class Prototype //抽象基类
{
private:
	string m_strName;
public:
	Prototype(string strName){ m_strName = strName; }
	Prototype() { m_strName = " "; }
	void Show()
	{
		cout<<m_strName<<endl;
	}
	virtual Prototype* Clone() = 0 ;
} ;

// class ConcretePrototype1
class ConcretePrototype1 : public Prototype
{
public:
	ConcretePrototype1(string strName) : Prototype(strName){}
	ConcretePrototype1(){}

	virtual Prototype* Clone()
	{
		ConcretePrototype1 *p = new ConcretePrototype1() ;
		*p = *this ; //复制对象
		return p ;
	}
} ;

// class ConcretePrototype2
class ConcretePrototype2 : public Prototype
{
public:
	ConcretePrototype2(string strName) : Prototype(strName){}
	ConcretePrototype2(){}

	virtual Prototype* Clone()
	{
		ConcretePrototype2 *p = new ConcretePrototype2() ;
		*p = *this ; //复制对象
		return p ;
	}
} ;
```

```
//客户端
int main()
{
	ConcretePrototype1* test = new ConcretePrototype1("小王");
	ConcretePrototype2* test2 = (ConcretePrototype2*)test->Clone();
	test->Show();
	test2->Show();
	return 0;
}
```

（七）模板方法模式

GOOD：把不变的代码部分都转移到父类中，将可变的代码用 virtual 留到子类重写



例：

```
#include<iostream>
#include <vector>
#include <string>
using namespace std;

class AbstractClass
{
public:
	void Show()
	{
		cout<<"我是"<<GetName()<<endl;
	}
protected:
```

```
        virtual string GetName()=0;
};

class Naruto : public AbstractClass
{
protected:
        virtual string GetName()
        {
                return "火影史上最帅的六代目---一鸣惊人 naruto";
        }
};

class OnePice : public AbstractClass
{
protected:
        virtual string GetName()
        {
                return "我是无恶不做的大海贼---路飞";
        }
};

//客户端
int main()
{
        Naruto* man = new Naruto();
        man->Show();

        OnePice* man2 = new OnePice();
        man2->Show();

        return 0;
}
```

迪米特法则

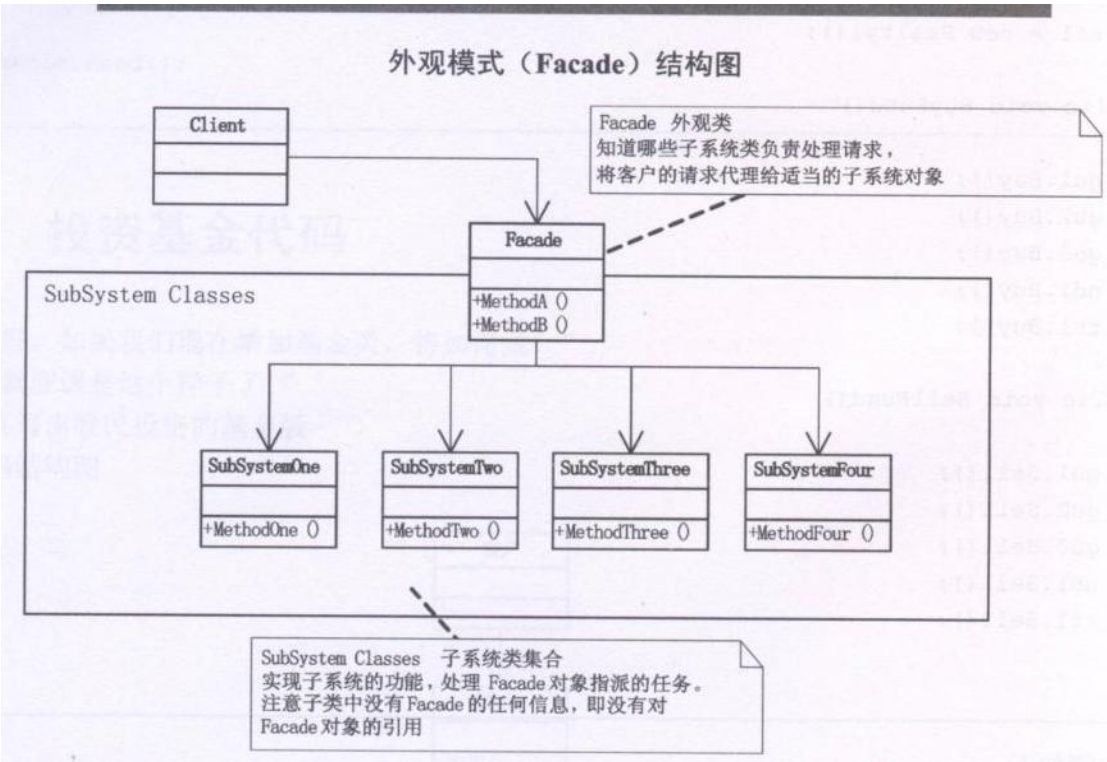
如果两个类不直接通信，那么这两个类就不应当发生直接的相互作用。如果一个类需要调用另一个类的某个方法的话，可以通过第三个类转发这个调用。

在类的结构设计上，每一个类都应该尽量降低成员的访问权限。

该法则在后面的适配器模式、解释模式等中有强烈的体现。

（八）外观模式

GOOD：为子系统的一组接口提供一个一致的界面。使用户使用起来更加方便。



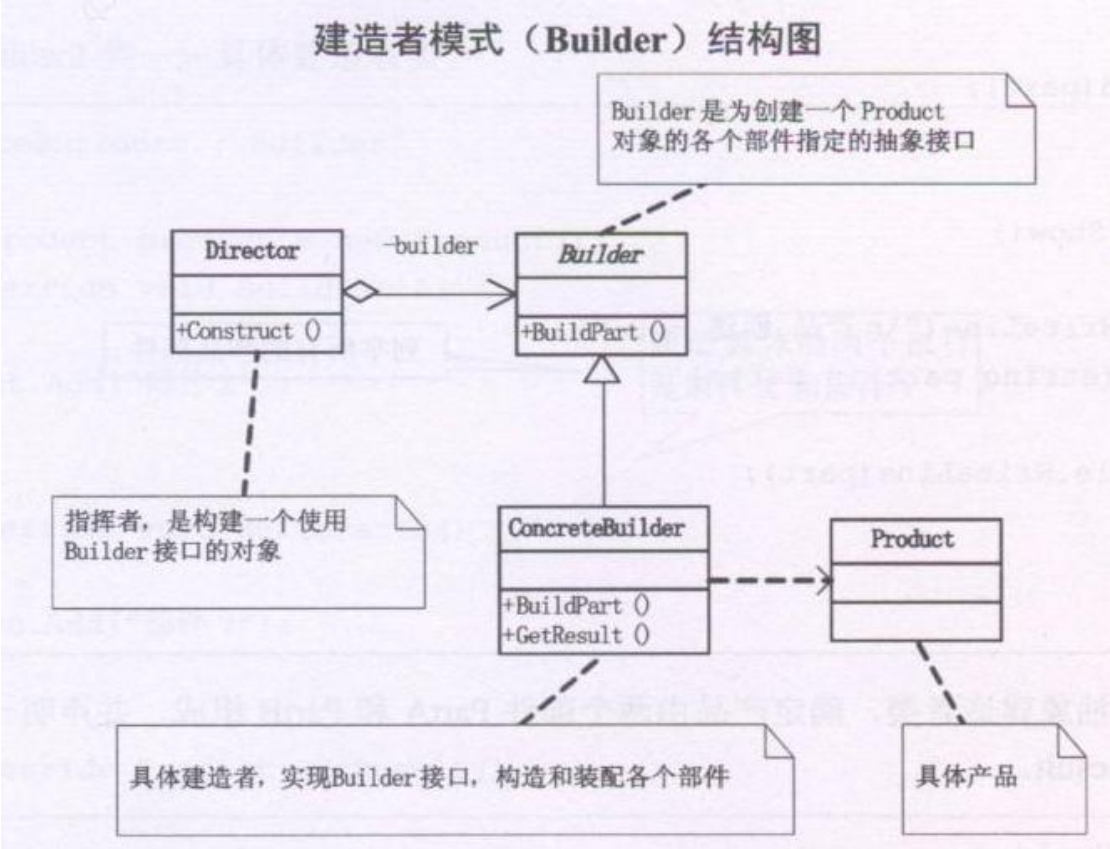
例：

```
#include<iostream>
#include <string>
using namespace std;

class SubSysOne
{
public:
	void MethodOne()
	{
		cout<<"方法一"<<endl;
	}
};

class SubSysTwo
{
public:
	void MethodTwo()
	{
		cout<<"方法二"<<endl;
	}
};

class SubSysThree
```

```
{
public:
	void MethodThree()
	{
		cout<<"方法三"<<endl;
	}
};

//外观类
class Facade
{
private:
	SubSysOne* sub1;
	SubSysTwo* sub2;
	SubSysThree* sub3;
public:
	Facade()
	{
		sub1 = new SubSysOne();
		sub2 = new SubSysTwo();
		sub3 = new SubSysThree();
	}
	~Facade()
	{
		delete sub1;
		delete sub2;
		delete sub3;
	}

	void FacadeMethod()
	{
		sub1->MethodOne();
		sub2->MethodTwo();
		sub3->MethodThree();
	}
};

//客户端
int main()
{
	Facade* test = new Facade();
	test->FacadeMethod();
	return 0;
}
```

（九）建造者模式（生成器模式）

GOOD：在当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时适用。 （P115 页）



例：

```
#include <string>
#include <iostream>
#include <vector>
using namespace std;

//最终的产品类
class Product
{
private:
    vector<string> m_product;
public:
    void Add(string strtemp)
    {
        m_product.push_back(strtemp);
    }
    void Show()
    {
        vector<string>::iterator p=m_product.begin();
```

```
        while (p!=m_product.end())
        {
            cout<<*p<<endl;
            p++;
        }
    }
};

//建造者基类
class Builder
{
public:
    virtual void BuilderA()=0;
    virtual void BuilderB()=0;
    virtual Product* GetResult()=0;
};
//第一种建造方式
class ConcreteBuilder1 : public Builder
{
private:
    Product* m_product;
public:
    ConcreteBuilder1()
    {
        m_product=new Product();
    }
    virtual void BuilderA()
    {
        m_product->Add("one");
    }
    virtual void BuilderB()
    {
        m_product->Add("two");
    }
    virtual Product* GetResult()
    {
        return m_product;
    }
};
//第二种建造方式
class ConcreteBuilder2 : public Builder
{
private:
    Product * m_product;
```

```
public:
    ConcreteBuilder2()
    {
        m_product=new Product();
    }
    virtual void BuilderA()
    {
        m_product->Add("A");
    }
    virtual void BuilderB()
    {
        m_product->Add("B");
    }
    virtual Product* GetResult()
    {
        return m_product;
    }
};

//指挥者类
class Direct
{
public:
    void Construct(Builder* temp)
    {
        temp->BuilderA();
        temp->BuilderB();
    }
};

//客户端
int main()
{
    Direct *p=new Direct();
    Builder* b1=new ConcreteBuilder1();
    Builder* b2=new ConcreteBuilder2();

    p->Construct(b1);              //调用第一种方式
    Product* pb1=b1->GetResult();
    pb1->Show();

    p->Construct(b2);              //调用第二种方式
    Product * pb2=b2->GetResult();
    pb2->Show();
```

```
    return 0;
}
```

例二（其实这个例子应该放在前面讲的）：

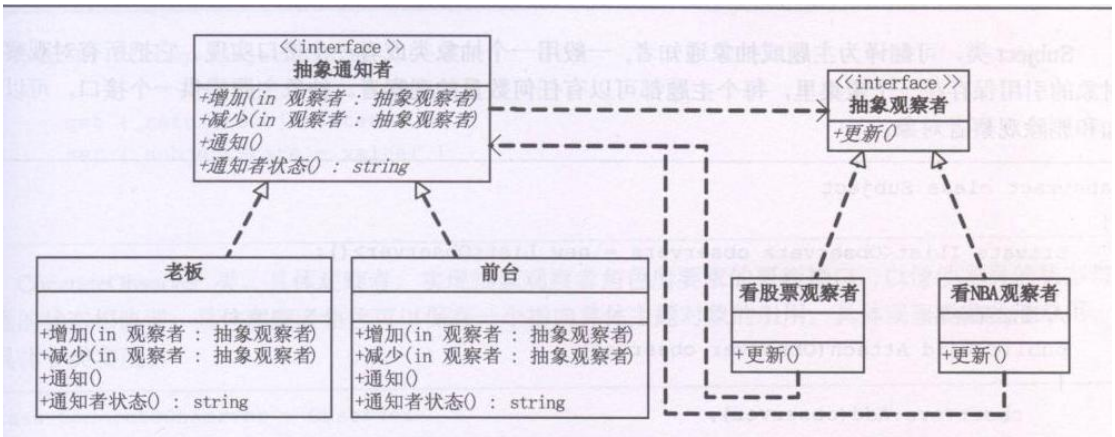
```
#include <string>
#include <iostream>
#include <vector>
using namespace std;

class Person
{
public:
    virtual void CreateHead()=0;
    virtual void CreateHand()=0;
    virtual void CreateBody()=0;
    virtual void CreateFoot()=0;
};
class ThinPerson : public Person
{
public:
    virtual void CreateHead()
    {
        cout<<"thin head"<<endl;
    }
    virtual void CreateHand()
    {
        cout<<"thin hand"<<endl;
    }
    virtual void CreateBody()
    {
        cout<<"thin body"<<endl;
    }
    virtual void CreateFoot()
    {
        cout<<"thin foot"<<endl;
    }
};

class ThickPerson : public Person
{
public:
    virtual void CreateHead()
    {
        cout<<"ThickPerson head"<<endl;
```

```
    }
    virtual void CreateHand()
    {
        cout<<"ThickPerson hand"<<endl;
    }
    virtual void CreateBody()
    {
        cout<<"ThickPerson body"<<endl;
    }
    virtual void CreateFoot()
    {
        cout<<"ThickPerson foot"<<endl;
    }
};
//指挥者类
class Direct
{
private:
    Person* p;
public:
    Direct(Person* temp) { p = temp;}
    void Create()
    {
        p->CreateHead();
        p->CreateBody();
        p->CreateHand();
        p->CreateFoot();
    }
};

//客户端代码：
int main()
{
    Person *p=new ThickPerson();
    Direct *d= new Direct(p);
    d->Create();
    delete d;
    delete p;
    return 0;
}
```

（十）观察者模式

GOOD：定义了一种一对多的关系，让多个观察对象（公司员工）同时监听一个

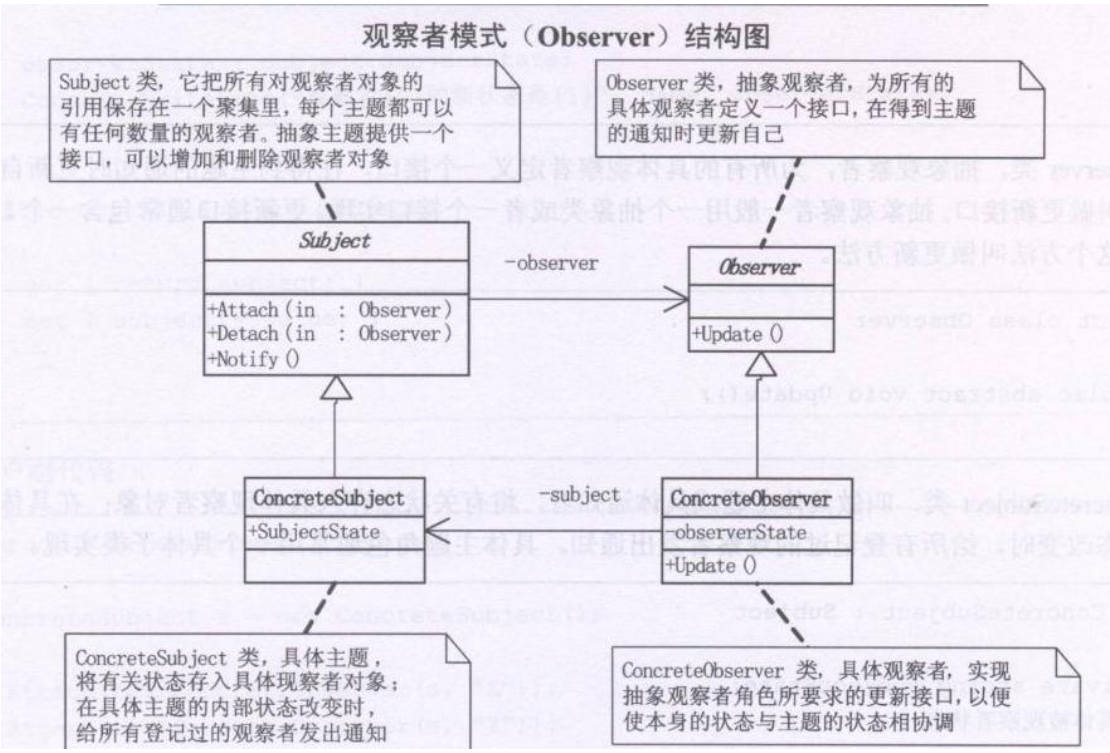
主题对象（秘书），主题对象状态发生变化时，会通知所有的观察者，使它们能够更新自己。



例：

```
#include <string>
#include <iostream>
#include <vector>
using namespace std;

class Secretary;
//看股票的同事类（观察对象，观察者）
class StockObserver
{
private:
	string name;
	Secretary* sub;
public:
	StockObserver(string strname,Secretary* strsub)
	{
		name=strname;
		sub=strsub;
	}
	void Update();
};
//秘书类（主题对象，通知者）
class Secretary
{
private:
	vector<StockObserver> observers;
public:
	string action;
	void Add(StockObserver ob)
	{
		observers.push_back(ob);
	}
```

```
    void Notify()
    {
        vector<StockObserver>::iterator p = observers.begin();
        while (p!=observers.end())
        {
            (*p).Update();
            p++;
        }
    }
};

void StockObserver::Update()
{
    cout<<name<<":"<<sub->action<<",不要玩股票了，要开始工作了"<<endl;
}

//客户端
int main()
{
    Secretary *p=new Secretary();   //创建通知者

    //观察者
    StockObserver *s1= new StockObserver("小李",p);
    StockObserver *s2 = new StockObserver("小赵",p);
    //加入通知队列
    p->Add(*s1);
    p->Add(*s2);
    //事件
    p->action="老板来了";
    //通知
    p->Notify();
    return 0;
}
```



例：

```
#include <string>
#include <iostream>
#include <vector>
using namespace std;

class SecretaryBase;
//抽象观察者
class CObserverBase
{
protected:
	string name;
	SecretaryBase* sub;
public:
	CObserverBase(string strname,SecretaryBase* strsub)
	{
		name=strname;
		sub=strsub;
	}
	virtual void Update()=0;
};
//具体的观察者，看股票的
class StockObserver : public CObserverBase
{
public:
```

```
        StockObserver(string strname,SecretaryBase* strsub) : CObserverBase(strname,strsub)
        {
        }
        virtual void Update();
    };

    //具体观察者，看 NBA 的
    class NBAObserver : public CObserverBase
    {
    public:
        NBAObserver(string strname,SecretaryBase* strsub) :
CObserverBase(strname,strsub){}
        virtual void Update();
    };


    //抽象通知者
    class SecretaryBase
    {
    public:
        string action;
        vector<CObserverBase*> observers;
    public:
        virtual void Attach(CObserverBase* observer)=0;
        virtual void Notify()=0;
    };

    //具体通知者
    class Secretary :public SecretaryBase
    {
    public:
        void Attach(CObserverBase* ob)
        {
            observers.push_back(ob);
        }
        void Notify()
        {
            vector<CObserverBase*>::iterator p = observers.begin();
            while (p!=observers.end())
            {
                (*p)->Update();
                p++;
            }
        }
```

```
};

void StockObserver::Update()
{
    cout<<name<<":"<<sub->action<<",不要玩股票了，要开始工作了"<<endl;
}
void NBAObserver::Update()
{
    cout<<name<<":"<<sub->action<<",不要看 NBA 了，老板来了"<<endl;
}
```

客户端：

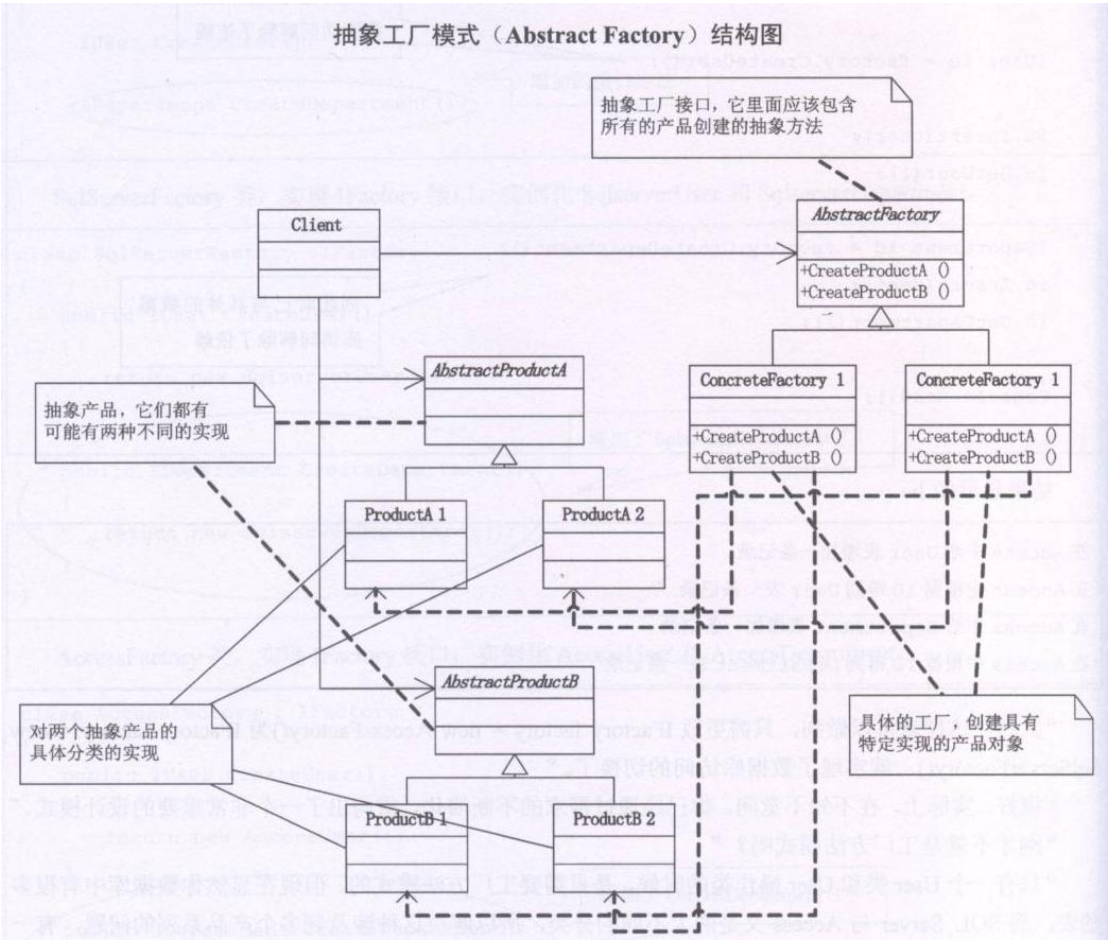
```
int main()
{
    SecretaryBase *p=new Secretary();   //创建观察者

    //被观察的对象
    CObserverBase *s1= new NBAObserver("小李",p);
    CObserverBase *s2 = new StockObserver("小赵",p);
    //加入观察队列
    p->Attach(s1);
    p->Attach(s2);
    //事件
    p->action="老板来了";
    //通知
    p->Notify();

    return 0;
}
```

（十一）抽象工厂模式

GOOD：定义了一个创建一系列相关或相互依赖的接口，而无需指定它们的具体类。

用于交换产品系列，如 ACCESS->SQL SERVER；

产品的具体类名被具体工厂的实现分离



例：

```
#include <string>
#include <iostream>
#include <vector>
using namespace std;

//用户抽象接口
class IUser
{
public :
	virtual void GetUser()=0;
	virtual void InsertUser()=0;
};

//部门抽象接口
class IDepartment
{
public:
	virtual void GetDepartment()=0;
	virtual void InsertDepartment()=0;
```

```
};

//ACCESS 用户
class CAccessUser : public IUser
{
public:
	virtual void GetUser()
	{
		cout<<"Access GetUser"<<endl;
	}
	virtual void InsertUser()
	{
		cout<<"Access InsertUser"<<endl;
	}
};

//ACCESS 部门
class CAccessDepartment : public IDepartment
{
public:
	virtual void GetDepartment()
	{
		cout<<"Access GetDepartment"<<endl;
	}
	virtual void InsertDepartment()
	{
		cout<<"Access InsertDepartment"<<endl;
	}
};

//SQL 用户
class CSqlUser : public IUser
{
public:
	virtual void GetUser()
	{
		cout<<"Sql User"<<endl;
	}
	virtual void InsertUser()
	{
		cout<<"Sql User"<<endl;
	}
};
```

```
//SQL 部门类
class CSqlDepartment: public IDepartment
{
public:
	virtual void GetDepartment()
	{
		cout<<"sql getDepartment"<<endl;
	}
	virtual void InsertDepartment()
	{
		cout<<"sql insertdepartment"<<endl;
	}
};

//抽象工厂
class IFactory
{
public:
	virtual IUser* CreateUser()=0;
	virtual IDepartment* CreateDepartment()=0;
};

//ACCESS 工厂
class AccessFactory : public IFactory
{
public:
	virtual IUser* CreateUser()
	{
		return new	CAccessUser();
	}
	virtual IDepartment* CreateDepartment()
	{
		return new CAccessDepartment();
	}
};

//SQL 工厂
class SqlFactory : public IFactory
{
public:
	virtual IUser* CreateUser()
	{
		return new	CSqlUser();
	}
```

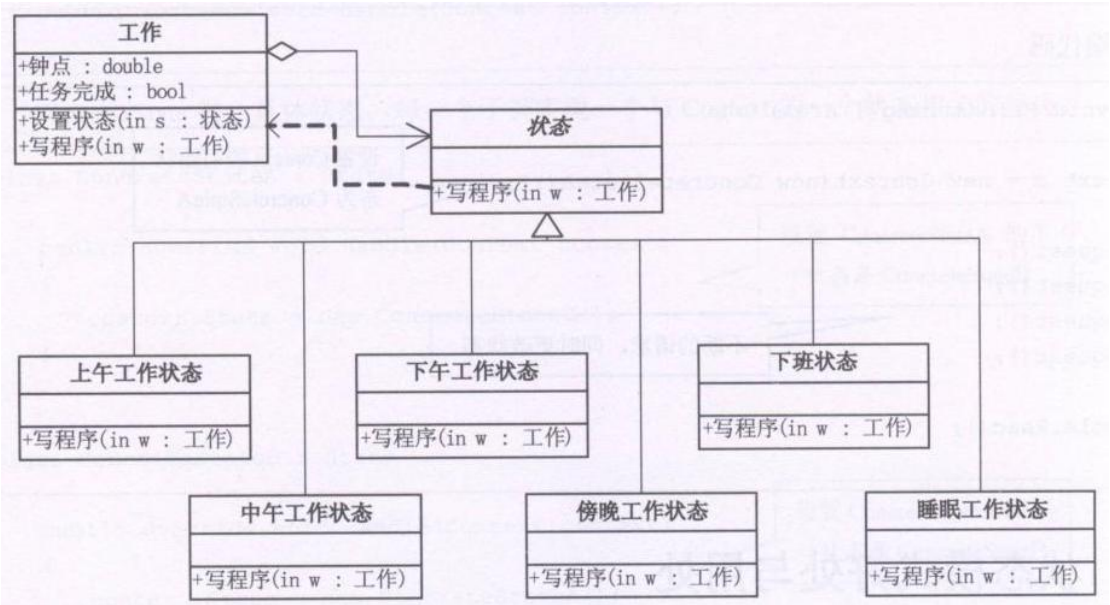
```
    virtual IDepartment* CreateDepartment()
    {
        return new CSqlDepartment();
    }
};
```

客户端：

```
int main()
{
    IFactory* factory= new SqlFactory();
    IUser* user=factory->CreateUser();
    IDepartment* depart = factory->CreateDepartment();
    user->GetUser();
    depart->GetDepartment();
    return 0;
}
```

（十二）状态模式

GOOD：当一个对象的行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为时，可考虑用到状态模式。



例

```
#include <iostream>
using namespace std;

class Work;
class ForenoonState;
class NoonState;
```

```
class State
{
public:
	virtual void WriteProgram(Work* w)=0;
};

class Work
{
private:
	State* current;
public:
	double hour;
public:
	Work();
	void SetState(State* temp)
	{
	    current =temp;
	}
	void Writeprogram()
	{
		current->WriteProgram(this);
	}

};

class NoonState :public State
{
public:
	virtual void WriteProgram(Work* w)
	{
		cout<<"execute"<<endl;
		if((w->hour)<13)
			cout<<"还不错啦"<<endl;
		else
			cout<<"不行了，还是睡觉吧"<<endl;
	}
};

class ForenoonState : public State
{
public:
	virtual void WriteProgram(Work* w)
	{
```

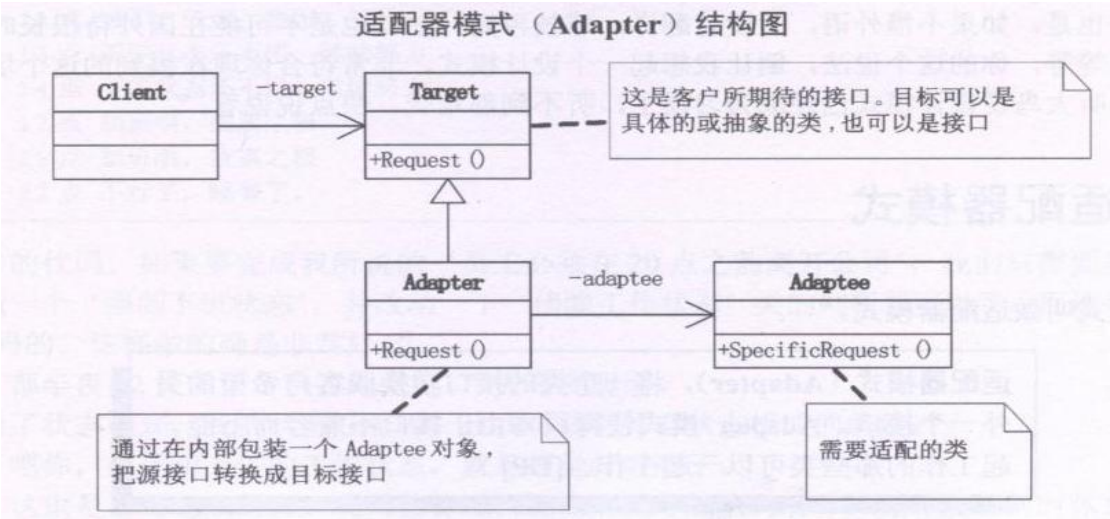
```
            if((w->hour)<12)
                cout<<"现在的精神无敌好"<<endl;
            else
            {
                w->SetState(new NoonState());
                w->Writeprogram(); //注意加上这句
            }
        }
};


Work::Work()
{
            current = new ForenoonState();
}
```

客户端：

```
int main()
{
        Work* mywork=new Work();
        mywork->hour=9;
        mywork->Writeprogram();
        mywork->hour = 14;
        mywork->Writeprogram();
        return 0;
}
```

（十三）适配器模式

GOOD：双方都不适合修改的时候，可以考虑使用适配器模式



例：

```
#include <iostream>
using namespace std;

class Target
{
public:
	virtual void Request()
	{
		cout<<"普通的请求"<<endl;
	}
};

class Adaptee
{
public:
	void SpecificalRequest()
	{
		cout<<"特殊请求"<<endl;
	}
};

class Adapter :public    Target
{
private:
	Adaptee* ada;
public:
	virtual void Request()
	{
		ada->SpecificalRequest();
		Target::Request();
	}
	Adapter()
	{
		ada=new Adaptee();
	}
	~Adapter()
	{
		delete ada;
	}
};
```

客户端：

```
int main()
{
```

```
    Adapter * ada=new Adapter();
    ada->Request();
    delete ada;
    return 0;
}
```

例二

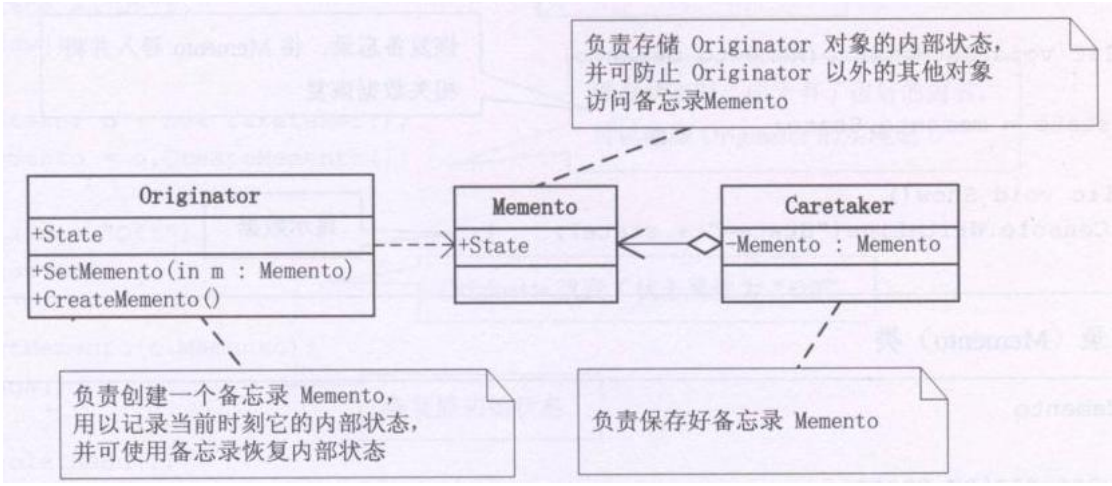
```
#include <iostream>
#include <string>
using namespace std;

class Player
{
protected:
    string name;
public:
    Player(string strName) { name = strName; }
    virtual void Attack()=0;
    virtual void Defense()=0;
};

class Forwards : public Player
{
public:
    Forwards(string strName):Player(strName){}
public:
    virtual void Attack()
    {
        cout<<name<<"前锋进攻"<<endl;
    }
    virtual void Defense()
    {
        cout<<name<<"前锋防守"<<endl;
    }
};

class Center : public Player
{
public:
    Center(string strName):Player(strName){}
public:
    virtual void Attack()
    {
        cout<<name<<"中场进攻"<<endl;
```

```
    }
    virtual void Defense()
    {
        cout<<name<<"中场防守"<<endl;
    }
};

//为中场翻译
class TransLater: public Player
{
private:
    Center *player;
public:
    TransLater(string strName):Player(strName)
    {
        player = new Center(strName);
    }
    virtual void Attack()
    {
        player->Attack();
    }
    virtual void Defense()
    {
        player->Defense();
    }
};


客户端
int main()
{
    Player *p=new TransLater("小李");
    p->Attack();
    return 0;
}
```

（十四）备忘录模式

GOOD：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，这样就可以将以后的对象状态恢复到先前保存的状态。

适用于功能比较复杂的，但需要记录或维护属性历史的类；或者需要保存的属性只是众多属性中的一小部分时 Originator 可以根据保存的 Memo 还原到前一状态。



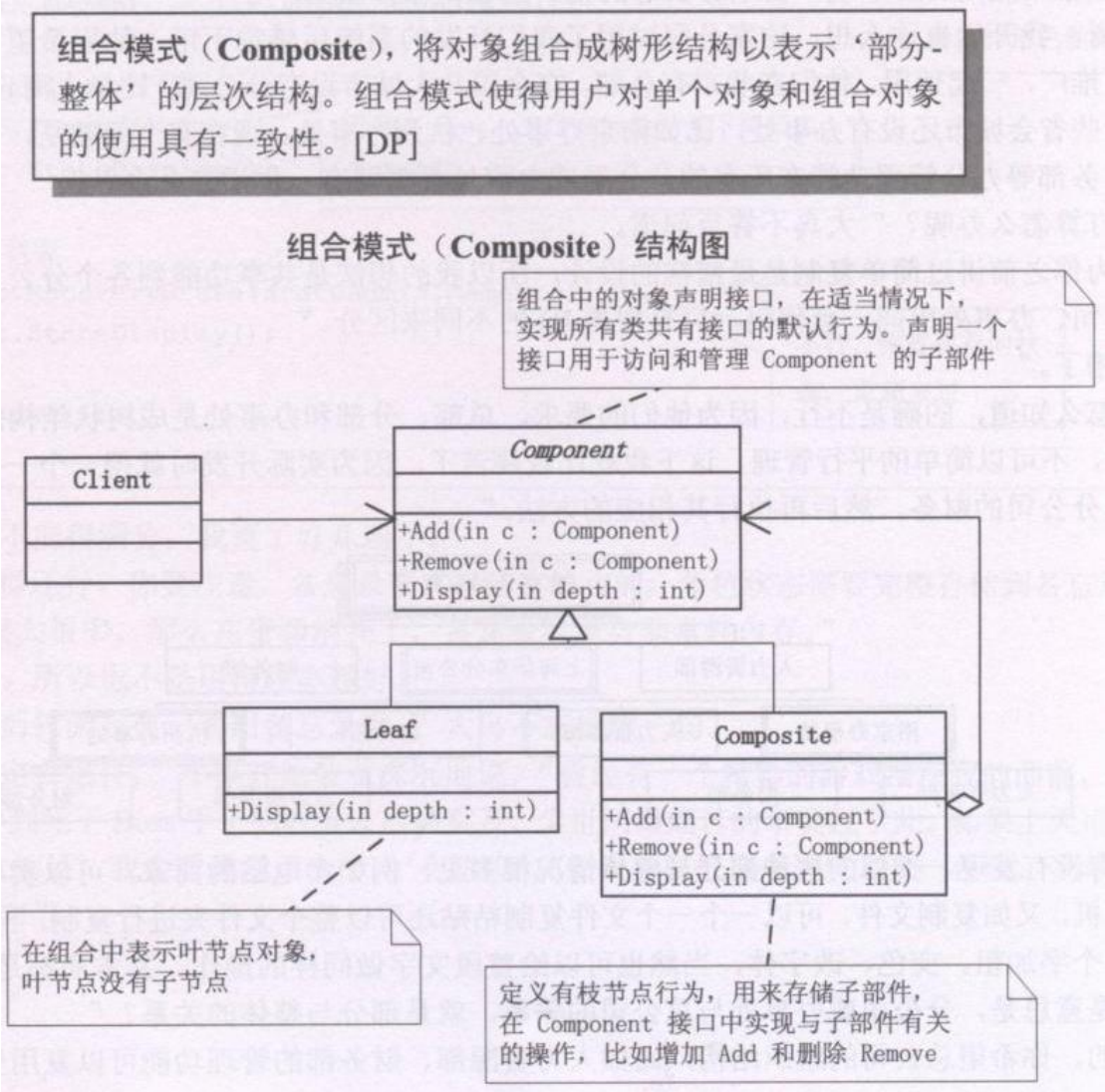
例：

```
#include <iostream>
#include <string>
using namespace std;

class Memo;

//发起人类
class Originator
{
public:
	string state;
	Memo* CreateMemo();
	void SetMemo(Memo* memo);
	void Show()
	{
		cout<<"状态："<<state<<endl;
	}
};
//备忘录类
class Memo
{
public:
	string state;
	Memo(string strState)
	{
		state= strState;
	}
};

Memo* Originator::CreateMemo()
{
```

```
        return new Memo(state);
    }

    void Originator::SetMemo(Memo* memo)
    {
        state = memo->state;
    }

    //管理者类
    class Caretaker
    {
    public:
        Memo* memo;
    };

    客户端：
    int main()
    {
        Originator* on=new Originator();
        on->state = "on";
        on->Show();

        Caretaker* c= new Caretaker();
        c->memo = on->CreateMemo();

        on->state = "off";
        on->Show();

        on->SetMemo(c->memo);
        on->Show();
        return 0;
    }
```

（十五）组合模式

GOOD：整体和部分可以被一致对待（如 WORD 中复制一个文字、一段文字、一篇文章都是一样的操作）

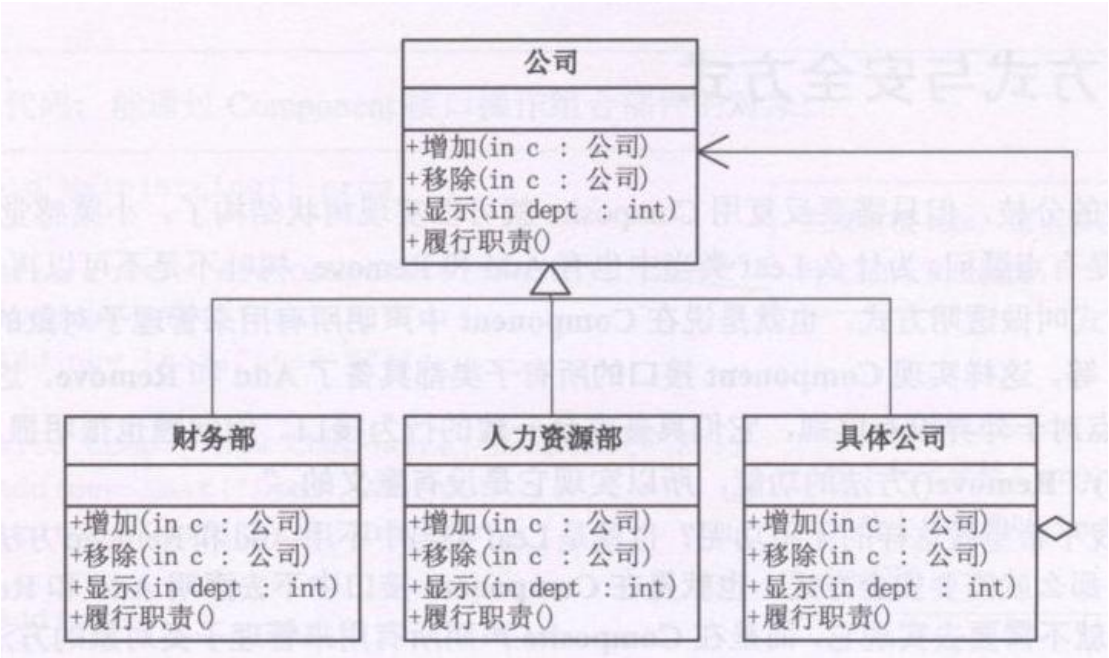


例：

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Component
{
public:
	string m_strName;
	Component(string strName)
	{
		m_strName = strName;
	}
	virtual void Add(Component* com)=0;
	virtual void Display(int nDepth)=0;
};
```

```
class Leaf : public Component
{
public:
	Leaf(string strName): Component(strName){}

	virtual void Add(Component* com)
	{
		cout<<"leaf can't add"<<endl;
	}

	virtual void Display(int nDepth)
	{
		string strtemp;
		for(int i=0; i < nDepth; i++)
		{
			strtemp+="-";
		}
		strtemp += m_strName;
		cout<<strtemp<<endl;
	}
};

class Composite : public Component
{
private:
	vector<Component*> m_component;
public:
	Composite(string strName) : Component(strName){}

	virtual void Add(Component* com)
	{
		m_component.push_back(com);
	}

	virtual void Display(int nDepth)
	{
		string strtemp;
		for(int i=0; i < nDepth; i++)
		{
			strtemp+="-";
		}
		strtemp += m_strName;
		cout<<strtemp<<endl;
```

```
        vector<Component*>::iterator p=m_component.begin();
        while (p!=m_component.end())
        {
            (*p)->Display(nDepth+2);
            p++;
        }
    }

};

//客户端
#include "Model.h"

int main()
{
    Composite* p=new Composite("小王");
    p->Add(new Leaf("小李"));
    p->Add(new Leaf("小赵"));

    Composite* p1 = new Composite("小小五");
    p1->Add(new Leaf("大三"));

    p->Add(p1);
    p->Display(1);
    return 0;
}
```



例二

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Company
{
protected:
	string m_strName;
public:
	Company(string strName)
	{
		m_strName = strName;
	}

	virtual void Add(Company* c)=0;
	virtual void Display(int nDepth)=0;
	virtual void LineOfDuty()=0;
};

class ConcreteCompany: public Company
{
private:
	vector<Company*> m_company;
public:
	ConcreteCompany(string strName):Company(strName){}

	virtual void Add(Company* c)
	{
		m_company.push_back(c);
	}
	virtual void Display(int nDepth)
	{
		string strtemp;
		for(int i=0; i < nDepth; i++)
		{
			strtemp += "-";
		}
		strtemp +=m_strName;
		cout<<strtemp<<endl;

		vector<Company*>::iterator p=m_company.begin();
		while (p!=m_company.end())
```

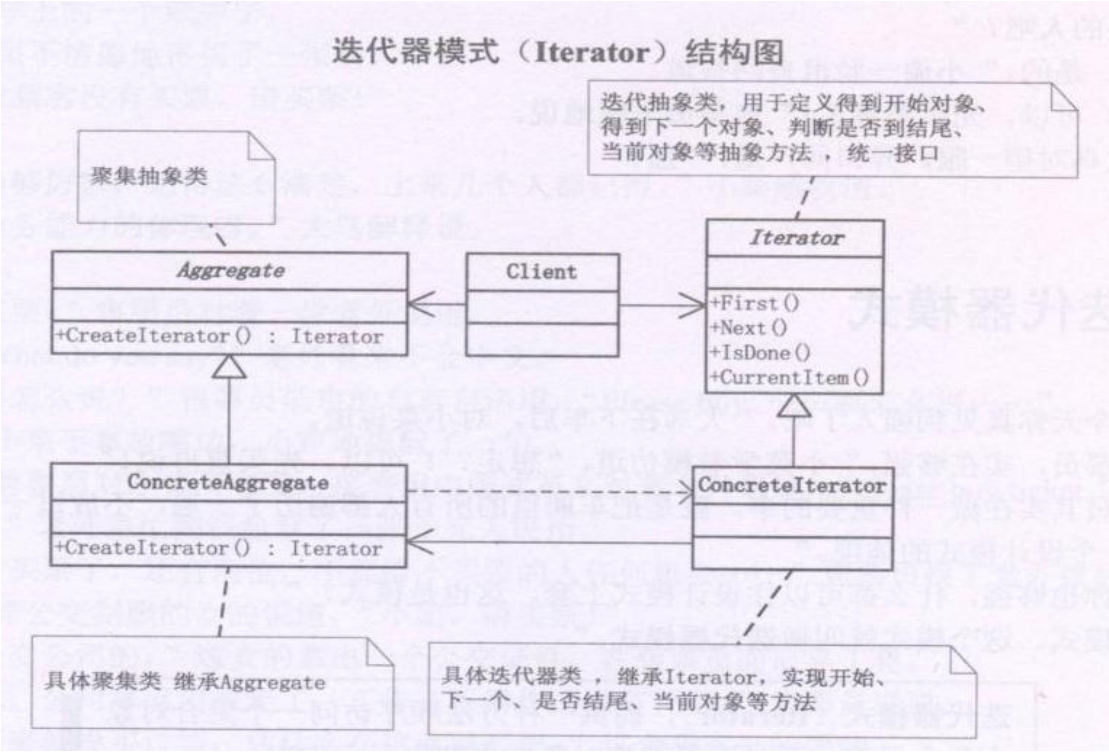
```
            {
                (*p)->Display(nDepth+2);
                p++;
            }
        }
        virtual void LineOfDuty()
        {
            vector<Company*>::iterator p=m_company.begin();
            while (p!=m_company.end())
            {
                (*p)->LineOfDuty();
                p++;
            }
        }
};

class HrDepartment : public Company
{
public:

        HrDepartment(string strname) : Company(strname){}

        virtual void Display(int nDepth)
        {
            string strtemp;
            for(int i = 0; i < nDepth; i++)
            {
                strtemp += "-";
            }

            strtemp += m_strName;
            cout<<strtemp<<endl;
        }
        virtual void Add(Company* c)
        {
            cout<<"error"<<endl;
        }

        virtual void LineOfDuty()
        {
            cout<<m_strName<<":招聘人才"<<endl;
        }
};
```

```
//客户端：
int main()
{
    ConcreteCompany *p = new ConcreteCompany("清华大学");
    p->Add(new HrDepartment("清华大学人才部"));

    ConcreteCompany *p1 = new ConcreteCompany("数学系");
    p1->Add(new HrDepartment("数学系人才部"));

    ConcreteCompany *p2 = new ConcreteCompany("物理系");
    p2->Add(new HrDepartment("物理系人才部"));

    p->Add(p1);
    p->Add(p2);

    p->Display(1);
    p->LineOfDuty();
    return 0;
}
```

（十六）迭代器模式

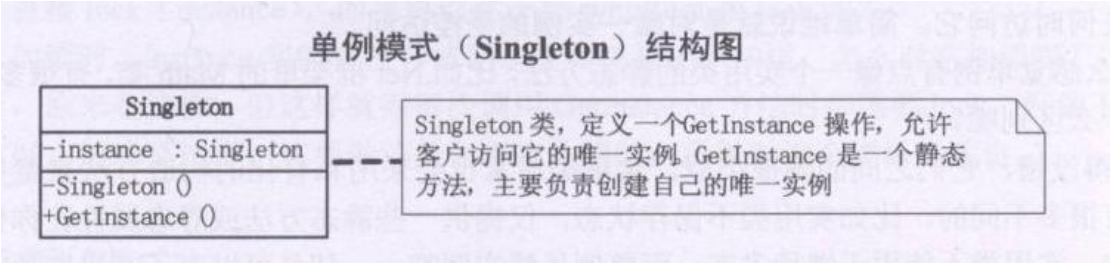
GOOD：提供一种方法顺序访问一个聚敛对象的各个元素，而又不暴露该对象的内部表示。

为遍历不同的聚集结构提供如开始，下一个，是否结束，当前一项等统一接口。



（十七）单例模式

GOOD：保证一个类仅有一个实例，并提供一个访问它的全局访问点



例：

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Singelton
{
private:
    Singelton(){}
    static Singelton* singel;

public:
    static Singelton* GetInstance()
    {
        if(singel == NULL)
        {
            singel = new Singelton();
        }
        return singel;
    }

};
Singelton* Singelton::singel = NULL;//注意静态变量类外初始化
```

客户端：

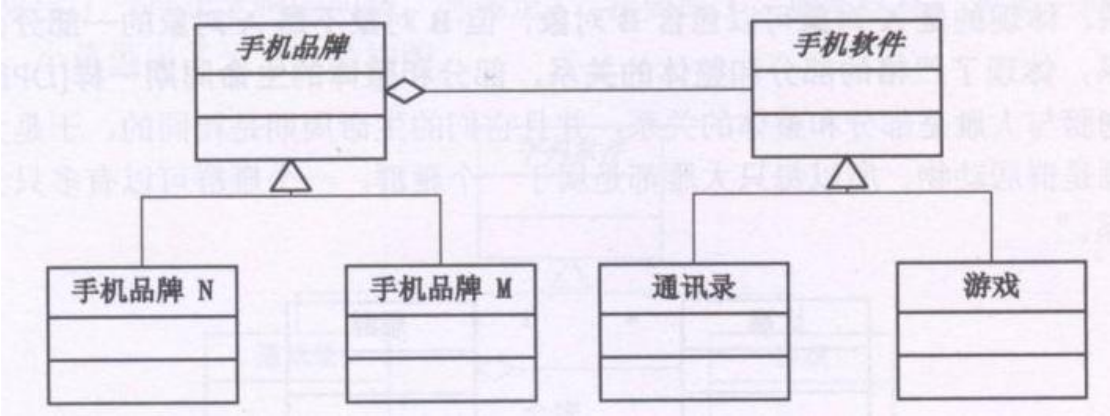
```
int main()
{
    Singelton* s1=Singelton::GetInstance();
    Singelton* s2=Singelton::GetInstance();
    if(s1 == s2)
        cout<<"ok"<<endl;
    else
        cout<<"no"<<endl;
```

```
    return 0;
}
```

（十八）桥接模式

GOOD：将抽象部分与实现部分分离，使它们可以独立变化。

这里说的意思不是让抽象基类与具体类分离，而是现实系统可能有多角度分类，每一种分类都有可能变化，那么把这种多角度分离出来让它们独立变化，减少它们之间的耦合性，即如果继承不能实现“开放－封闭原则”的话，就应该考虑用桥接模式。如下例：让“手机”既可以按品牌分类也可以



例：

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

//手机软件
class HandsetSoft
{
public:
    virtual void Run()=0;
};

//游戏软件
class HandsetGame : public HandsetSoft
{
public:
    virtual void Run()
    {
        cout<<"运行手机游戏"<<endl;
    }
};

//通讯录软件
```

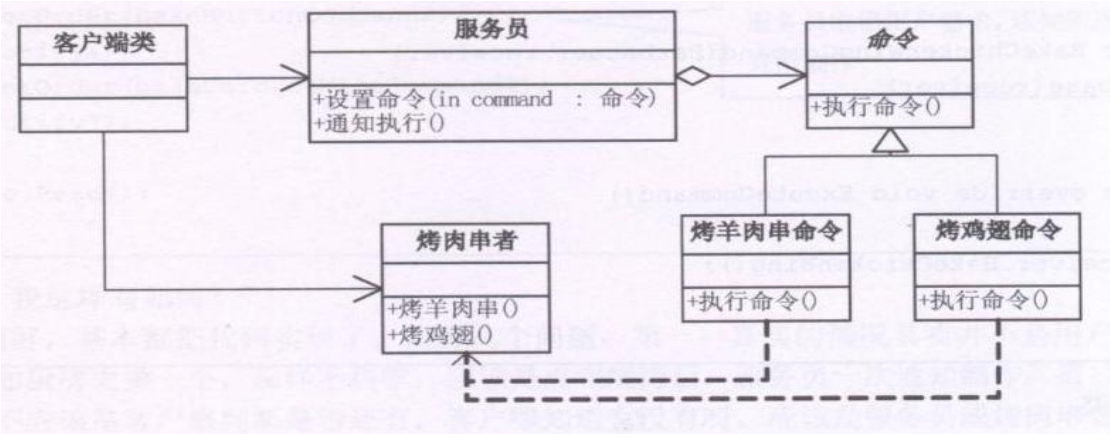
```
class HandSetAddressList : public HandsetSoft
{
public:
	virtual void Run()
	{
		cout<<"手机通讯录"<<endl;
	}
};

//手机品牌
class HandsetBrand
{
protected:
	HandsetSoft* m_soft;
public:
	void SetHandsetSoft(HandsetSoft* temp)
	{
		m_soft = temp;
	}
	virtual void Run()=0;
};

//M 品牌
class HandsetBrandM : public HandsetBrand
{
public:
	virtual void Run()
	{
		m_soft->Run();
	}
};

//N 品牌
class HandsetBrandN : public HandsetBrand
{
public:
	virtual void Run()
	{
		m_soft->Run();
	}
};

//客户端
int main()
```

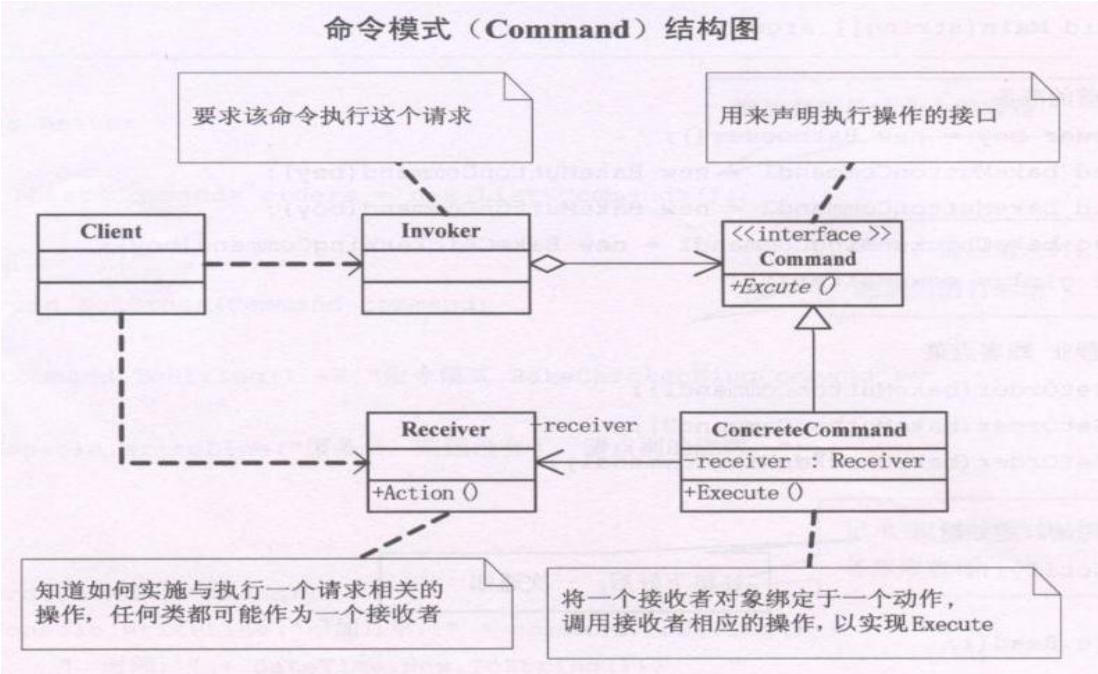
```
{
    HandsetBrand *brand;
    brand = new HandsetBrandM();
    brand->SetHandsetSoft(new HandsetGame());
    brand->Run();
    brand->SetHandsetSoft(new HandSetAddressList());
    brand->Run();

    return 0;
}
```

（十九）命令模式

GOOD：一、建立命令队列；二、可以将命令记入日志；三、接收请求的一方可以拒绝；四、添加一个新命令类不影响其它类；

命令模式把请求一个操作的对象与知道怎么操行一个操作的对象分开





例：

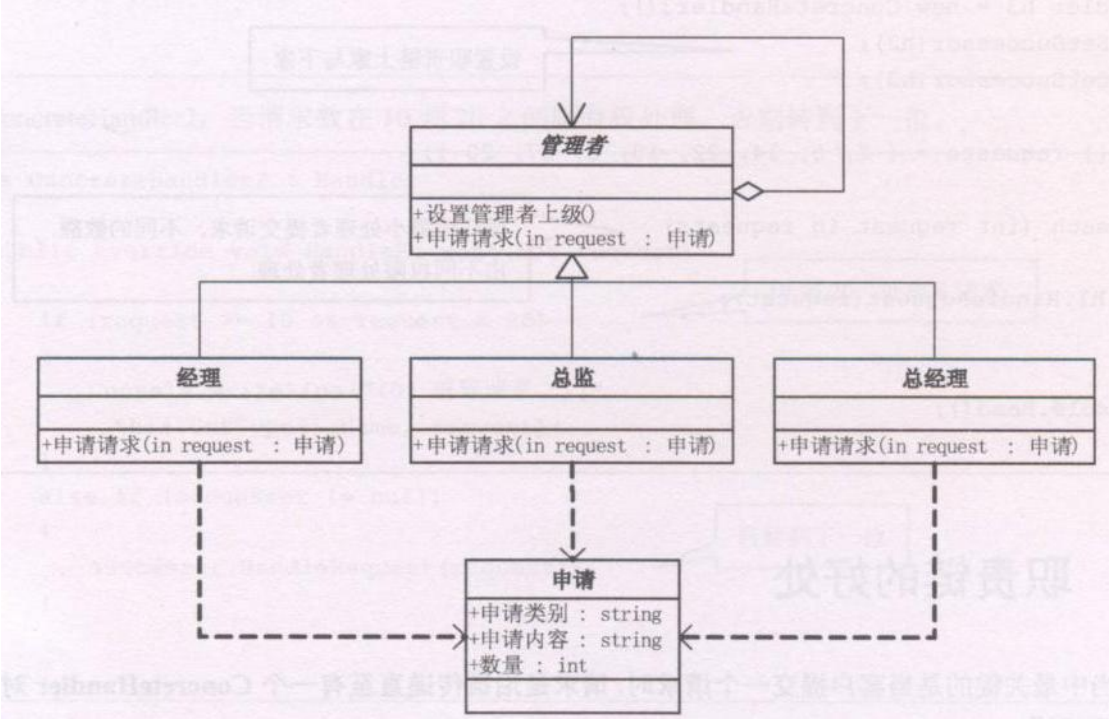
```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

//烤肉师傅
class Barbucer
{
public:
	void MakeMutton()
	{
		cout<<"烤羊肉"<<endl;
	}
	void MakeChickenWing()
	{
		cout<<"烤鸡翅膀"<<endl;
	}
};

//抽象命令类
class Command
{
protected:
	Barbucer* receiver;
public:
	Command(Barbucer* temp)
	{
		receiver = temp;
	}
	virtual void ExecuteCmd()=0;
};

//烤羊肉命令
class BakeMuttonCmd : public Command
{
public:
	BakeMuttonCmd(Barbucer* temp) : Command(temp){}
	virtual void ExecuteCmd()
	{
		receiver->MakeMutton();
	}
};
```

```
//烤鸡翅
class ChickenWingCmd : public Command
{
public:
	ChickenWingCmd(Barbucer* temp) : Command(temp){}

	virtual void ExecuteCmd()
	{
		receiver->MakeChickenWing();
	}
};

//服务员类
class Waiter
{
protected:
	vector<Command*> m_commandList;
public:
	void SetCmd(Command* temp)
	{
		m_commandList.push_back(temp);
		cout<<"增加定单"<<endl;
	}

	//通知执行
	void Notify()
	{
		vector<Command*>::iterator p=m_commandList.begin();
		while(p!=m_commandList.end())
		{
			(*p)->ExecuteCmd();
			p++;
		}
	}
};

//客户端
int main()
{
	//店里添加烤肉师傅、菜单、服务员等顾客
	Barbucer* barbucer=new Barbucer();
	Command* cmd= new BakeMuttonCmd(barbucer);
	Command* cmd2=new ChickenWingCmd(barbucer);
	Waiter* girl = new Waiter();
```

```
    //点菜
    girl->SetCmd(cmd);
    girl->SetCmd(cmd2);
    //服务员通知
    girl->Notify();
    return 0;
}
```

（二十）责任链模式

GOOD：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这个对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理为止。



例：

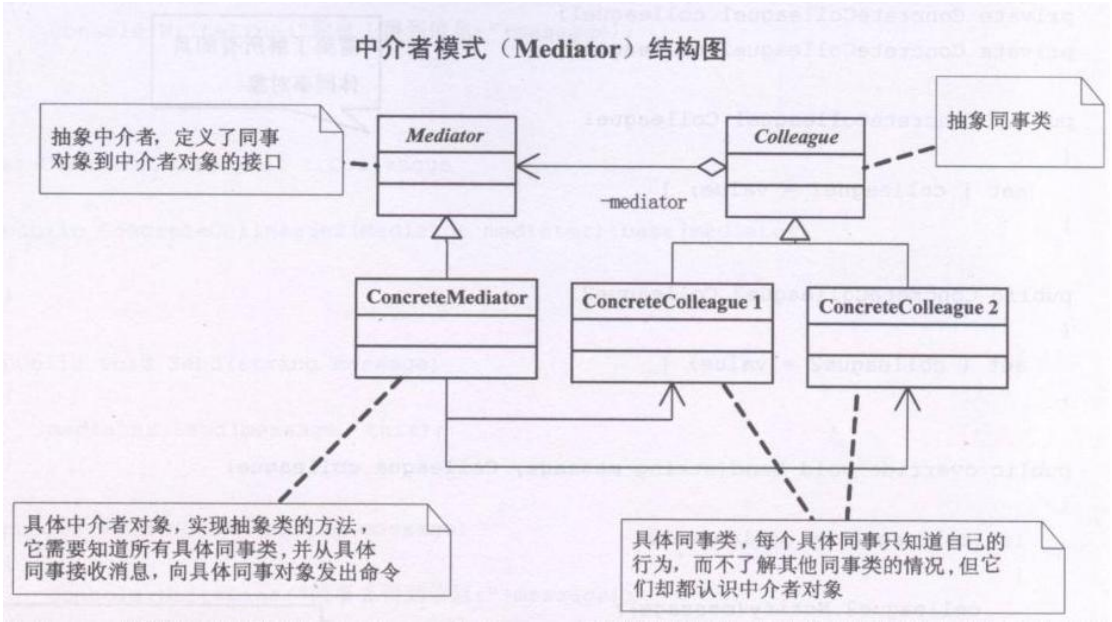
```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
//请求
class Request
{
public:
    string m_strContent;
    int m_nNumber;
};
//管理者
```

```
class Manager
{
protected:
	Manager* manager;
	string name;
public:
	Manager(string temp)
	{
		name = temp;
	}
	void SetSuccessor(Manager* temp)
	{
		manager = temp;
	}
	virtual void GetRequest(Request* request) = 0;
};
//经理
class CommonManager : public Manager
{
public:
	CommonManager(string strTemp) : Manager(strTemp){}
	virtual void GetRequest(Request* request)
	{
		if ( request->m_nNumber>=0 && request->m_nNumber<10 )
		{
			cout<<name<<"处理了"<<request->m_nNumber<<"个请求"<<endl;
		}
		else
		{
			manager->GetRequest(request);
		}
	}
};
//总监
class MajorDomo : public Manager
{
public:
	MajorDomo(string name) : Manager(name){}

	virtual void GetRequest(Request* request)
	{
		if(request->m_nNumber>=10)
		{
			cout<<name<<"处理了"<<request->m_nNumber<<"个请求"<<endl;
```

```
            }
        }
};

//客户端
int main()
{
    Manager * common = new CommonManager("张经理");
    Manager * major = new MajorDomo("李总监");

    common->SetSuccessor(major);

    Request* req = new Request();
    req->m_nNumber = 33;
    common->GetRequest(req);

    req->m_nNumber = 3;
    common->GetRequest(req);
    return 0;
}
```

（二十一）中介者模式

GOOD：用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显示的相互引用，从而降低耦合；而且可以独立地改变它们之间的交互。



例：

```
#include <iostream>
#include <string>
#include <vector>
```

```
using namespace std;

class Colleague;
//中介者类
class Mediator
{
public:
	virtual void Send(string message,Colleague* col) = 0;
};
//抽象同事类
class Colleague
{
protected:
	Mediator* mediator;
public:
	Colleague(Mediator* temp)
	{
		mediator = temp;
	}
};
//同事一
class Colleague1 : public Colleague
{
public:
	Colleague1(Mediator* media) : Colleague(media){}

	void Send(string strMessage)
	{
		mediator->Send(strMessage,this);
	}

	void Notify(string strMessage)
	{
		cout<<"同事一获得了消息"<<strMessage<<endl;
	}
};

//同事二
class Colleague2 : public Colleague
{
public:
	Colleague2(Mediator* media) : Colleague(media){}

	void Send(string strMessage)
```

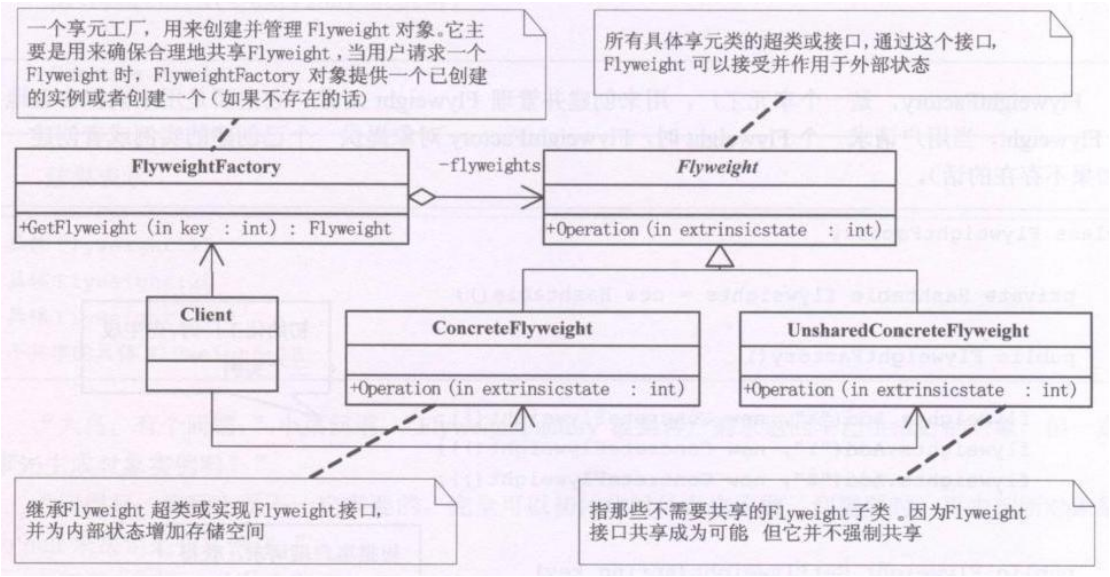
```
    {
        mediator->Send(strMessage,this);
    }

    void Notify(string strMessage)
    {
        cout<<"同事二获得了消息"<<strMessage<<endl;
    }
};

//具体中介者类
class ConcreteMediator : public Mediator
{
public:
    Colleague1 * col1;
    Colleague2 * col2;
    virtual void Send(string message,Colleague* col)
    {
        if(col == col1)
            col2->Notify(message);
        else
            col1->Notify(message);
    }
};

//客户端：
int main()
{
    ConcreteMediator * m = new ConcreteMediator();

    //让同事认识中介
    Colleague1* col1 = new Colleague1(m);
    Colleague2* col2 = new Colleague2(m);

    //让中介认识具体的同事类
    m->col1 = col1;
    m->col2 = col2;

    col1->Send("吃饭了吗？");
    col2->Send("还没吃，你请吗？");
    return 0;
}
```

（二十二）享元模式

GOOD：运用共享技术有效地支持大量细粒度的对象（对于C++来说就是共用一个内存块啦，对象指针指向同一个地方）。

如果一个应用程序使用了大量的对象，而这些对象造成了很大的存储开销就应该考虑使用。

还有就是对象的大多数状态可以外部状态，如果删除对象的外部状态，那么可以用较少的共享对象取代多组对象，此时可以考虑使用享元。



例：

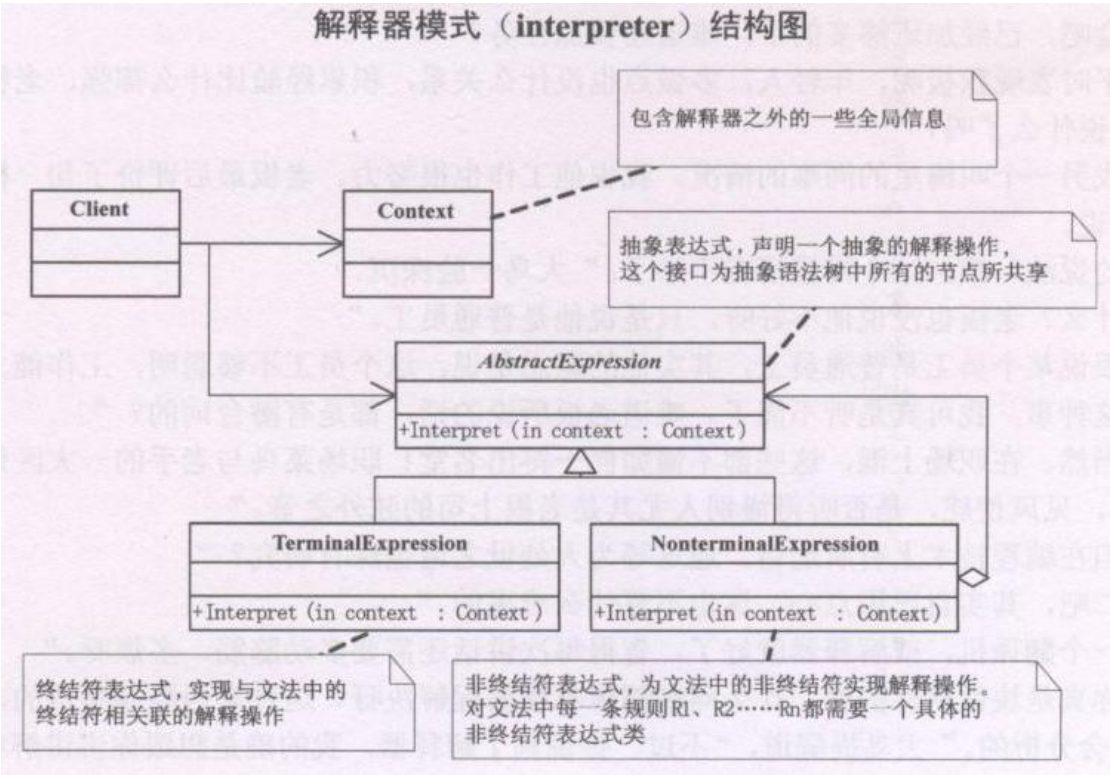
```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

//抽象的网站
class WebSite
{
public:
	virtual void Use()=0;
};

//具体的共享网站
class ConcreteWebSite : public WebSite
{
private:
	string name;
public:
	ConcreteWebSite(string strName)
	{
		name = strName;
	}
	virtual void Use()
```

```
    {
        cout<<"网站分类："<<name<<endl;
    }
};

//不共享的网站
class UnShareWebSite : public WebSite
{
private:
    string name;
public:
    UnShareWebSite(string strName)
    {
        name = strName;
    }
    virtual void Use()
    {
        cout<<"不共享的网站："<<name<<endl;
    }
};

//网站工厂类，用于存放共享的 WebSite 对象
class WebFactory
{
private:
    vector<WebSite*> websites;
public:
    WebSite* GetWeb()
    {
        vector<WebSite*>::iterator p = websites.begin();
        return *p;
    }
    WebFactory()
    {
        websites.push_back(new ConcreteWebSite("测试"));
    }
};

//客户端
int main()
{
    WebFactory* f= new WebFactory();
    WebSite* ws= f->GetWeb();
    ws->Use();
```

```
    WebSite* ws2 = f->GetWeb();
    ws2->Use();

    //不共享的类
    WebSite* ws3 = new UnShareWebSite("测试");
    ws3->Use();
    return 0;
}
```

（二十三）解释器模式

GOOD：通常当一个语言需要解释执行，并且你可以将该语言中的句子表示成为一个抽象的语法树时，可以使用解释器模式。



例：

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Context;
class AbstractExpression
{
public:
```

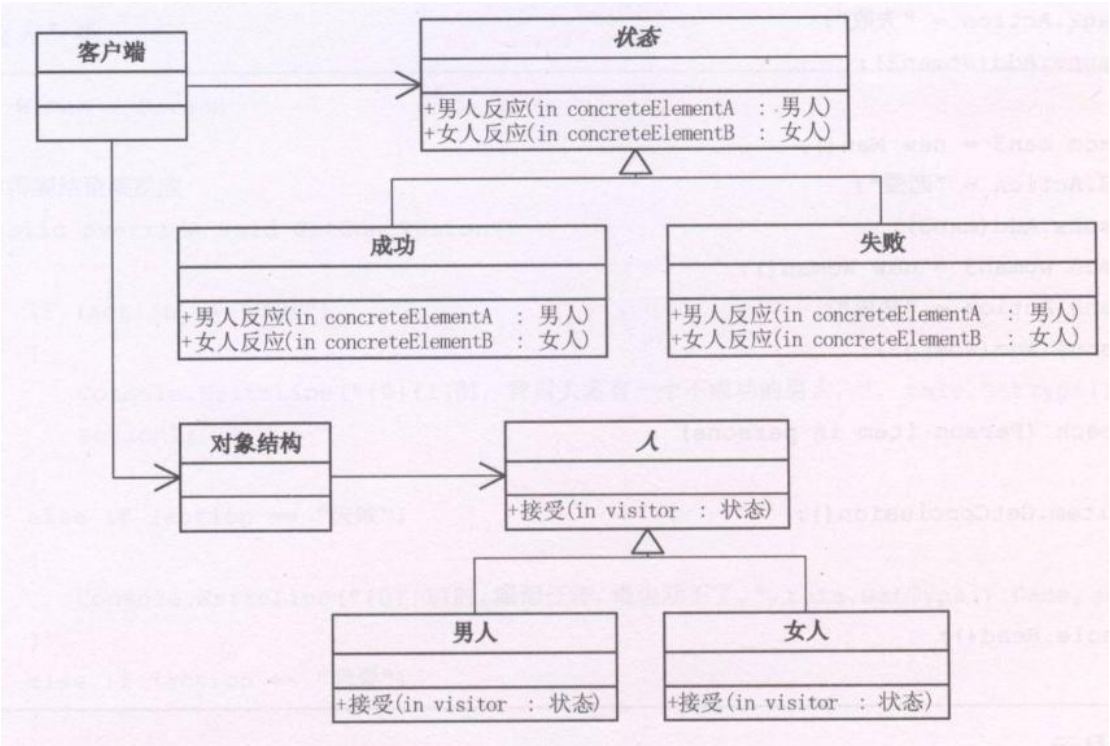
```
    virtual void Interpret(Context* context)=0;
};

class Expression : public AbstractExpression
{
public:
    virtual void Interpret(Context* context)
    {
        cout<<"终端解释器"<<endl;
    };
};

class NonterminalExpression : public AbstractExpression
{
public:
    virtual void Interpret(Context* context)
    {
        cout<<"非终端解释器"<<endl;
    }
};

class Context
{
public:
    string input;
    string output;
};

//客户端
int main()
{
    Context* context = new Context();
    vector<AbstractExpression*> express;
    express.push_back(new Expression());
    express.push_back(new NonterminalExpression());
    express.push_back(new NonterminalExpression());

    vector<AbstractExpression*>::iterator p = express.begin();
    while (p!= express.end())
    {
        (*p)->Interpret(context);
        p++;
    }
```

```
    return 0;
}
```

（二十四）访问者模式

GOOD：适用于数据结构稳定的系统。它把数据结构和作用于数据结构上的操作分离开，使得操作集合

优点：新增加操作很容易，因为增加新操作就相当于增加一个访问者，访问者模式将有关的行为集中到一个访问者对象中



例：

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Man;
class Woman;
//行为
class Action
{
public:
    virtual void GetManConclusion(Man* concreteElementA)=0;
    virtual void GetWomanConclusion(Woman* concreteElementB)=0;
};
//成功
class Success : public Action
```

```
{
public:
    virtual void GetManConclusion(Man* concreteElementA)
    {
        cout<<"男人成功时，背后有个伟大的女人"<<endl;
    }
    virtual void GetWomanConclusion(Woman* concreteElementB)
    {
        cout<<"女人成功时，背后有个没用的男人"<<endl;
    }
};

//失败
class Failure : public Action
{
public:
    virtual void GetManConclusion(Man* concreteElementA)
    {
        cout<<"男人失败时，背后有个伟大的女人"<<endl;
    }
    virtual void GetWomanConclusion(Woman* concreteElementB)
    {
        cout<<"女人失败时，背后有个没用的男人"<<endl;
    }
};

//抽象人类
class Person
{
public:
    virtual void Accept(Action* visitor)=0;
};

//男人
class Man : public Person
{
public:
    virtual void Accept(Action* visitor)
    {
        visitor->GetManConclusion(this);
    }
};

//女人
```

```
class Woman : public Person
{
public:
	virtual void Accept(Action* visitor)
	{
		visitor->GetWomanConclusion(this);
	}
};

//对象结构类
class ObjectStructure
{
private:
	vector<Person*> m_personList;

public:
	void Add(Person* p)
	{
		m_personList.push_back(p);
	}
	void Display(Action* a)
	{
		vector<Person*>::iterator p = m_personList.begin();
		while (p!=m_personList.end())
		{
			(*p)->Accept(a);
			p++;
		}
	}
};

//客户端
int main()
{
	ObjectStructure * os= new ObjectStructure();
	os->Add(new Man());
	os->Add(new Woman());

	Success* success = new Success();
	os->Display(success);

	Failure* fl = new Failure();
	os->Display(fl);
```

```
    return 0;
}
```