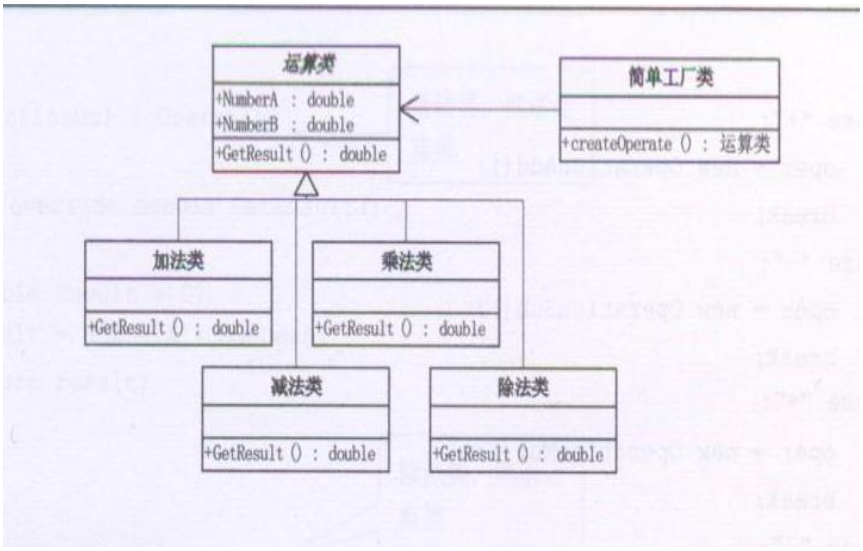
（一）简单工厂模式 .....錯誤! 尚未定義書籤。
（二）策略模式 .....錯誤! 尚未定義書籤。
策略与工厂结合.....錯誤! 尚未定義書籤。
单一职责原则.....錯誤! 尚未定義書籤。
开放－－封闭原则.....錯誤! 尚未定義書籤。
里氏代换原则.....錯誤! 尚未定義書籤。
依赖倒转原则.....錯誤! 尚未定義書籤。
（三）装饰模式 .....錯誤! 尚未定義書籤。
（四）代理模式 .....錯誤! 尚未定義書籤。
（五）工厂方法模式 .....錯誤! 尚未定義書籤。
（六）原型模式 .....錯誤! 尚未定義書籤。
（七）模板方法模式 .....錯誤! 尚未定義書籤。
迪米特法则.....錯誤! 尚未定義書籤。
（八）外观模式 .....錯誤! 尚未定義書籤。
（九）建造者模式（生成器模式）.....錯誤! 尚未定義書籤。
（十）观察者模式 .....錯誤! 尚未定義書籤。
（十一）抽象工厂模式.....錯誤! 尚未定義書籤。
（十二）状态模式 .....錯誤! 尚未定義書籤。
（十三）适配器模式 .....錯誤! 尚未定義書籤。
（十四）备忘录模式 .....錯誤! 尚未定義書籤。
（十五）组合模式 .....錯誤! 尚未定義書籤。
（十六）迭代器模式 .....錯誤! 尚未定義書籤。
（十七）单例模式 .....錯誤! 尚未定義書籤。
（十八）桥接模式 .....錯誤! 尚未定義書籤。
（十九）命令模式 .....錯誤! 尚未定義書籤。
（二十）责任链模式 .....錯誤! 尚未定義書籤。
（二十一）中介者模式.....錯誤! 尚未定義書籤。
（二十二）享元模式 .....錯誤! 尚未定義書籤。
（二十三）解释器模式.....錯誤! 尚未定義書籤。
（二十四）访问者模式.....錯誤! 尚未定義書籤。

（一）簡單工廠模式

主要用於創建對象。新添加類時，不會影響以前的系統代碼。核心思想是用一個工廠來根據輸入的條件產生不同的類，然後根據不同類的 virtual 函數得到不同的結果。

GOOD:適用於不同情況創建不同的類時

BUG：用戶端必須要知道基類和工廠類，耦合性差



（工廠類與基類為關聯關係）

例：

```
//基類
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;
	}
```

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

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

COperation* CCalculatorFactory::Create(char cOperator)
{
	COperation *oper;
	//在 C#中可以用反射來取消判斷時用的 switch，在 C++中用什麼呢？RTTI？？
	switch (cOperator)
	{
	case '+':
		oper=new AddOperation();
		break;
	case '-':
		oper=new SubOperation();
		break;
	default:
		oper=new AddOperation();
		break;
	}
	return oper;
}

用戶端
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;
}
```

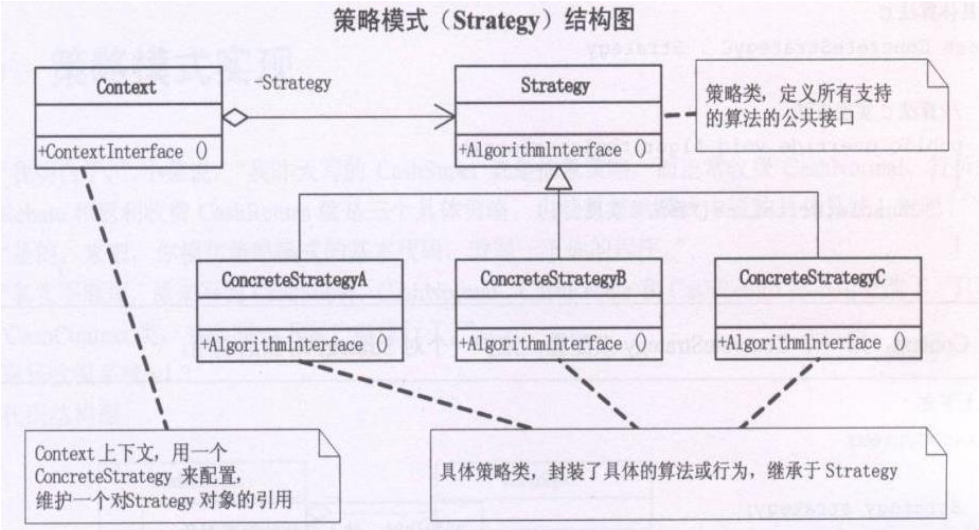
（二）策略模式

定義演算法家族，分別封裝起來，讓它們之間可以互相替換，讓演算法變化，不會影響到用戶

GOOD:適合類中的成員以方法為主，演算法經常變動；簡化了單元測試（因為每個演算法都有自己的類，可以通過自己的介面單獨測試。

策略模式和簡單工廠基本相同，但簡單工廠模式只能解決物件創建問題，對於經常變動的演算法應使用策略模式。

BUG:用戶端要做出判斷



例

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

//策略具體類—加法類
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();
    }
};

//用戶端
int main()
{
    int a,b;
    char c;
    cin>>a>>b;
    cout<<"請輸入運算子：;
    cin>>c;
    switch(c)
   {
      case '+':
        Context *context=new Context(new AddOperation(a,b));
        cout<<context->GetResult()<<endl;
       break;
      default:
       break;
  }
```

```
    return 0;
}
```

策略與工廠結合

GOOD:用戶端只需訪問 Context 類，而不用知道其它任何類資訊，實現了低耦合。
在上例基礎上，修改下面內容

```
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();
    }
};
//用戶端
int main()
{
    int a,b;
    cin>>a>>b;
    Context *test=new Context('+');
    cout<<test->GetResult()<<endl;
    return 0;
}
```

單一職責原則

就一個類而言，應該僅有一個引起它變化的原因。

如果一個類承擔的職責過多，就等於把這些職責耦合在一起，一個職責的變化可能會削弱或者抑制這個類完成其它職責能力。這種耦合會導制脆弱的設計，當變化發生時，設計會遭受到意想不到的破壞。

如果你能夠想到多於一個的動機去改變一個類，那麼這個類就具有多於一個的職

責。

開放——封閉原則

軟體實體可以擴展，但是不可修改。即對於擴展是開放的，對於修改是封閉的。面對需求，對程式的改動是通過增加代碼來完成的，而不是改動現有的代碼。

當變化發生時，我們就創建抽象來隔離以後發生同類的變化。

開放——封閉原則是物件導向的核心所在。開發人員應該對程式中呈現出頻繁變化的那部分做出抽象，拒絕對任何部分都刻意抽象及不成熟的抽象。

裡氏代換原則

一個軟體實體如果使用的是一個父類的話，那麼一定適用其子類。而且它察覺不出父類物件和子類物件的區別。也就是說：在軟體裡面，把父類替換成子類，程式的行為沒有變化。

子類型必須能夠替換掉它們的父類型。

依賴倒轉原則

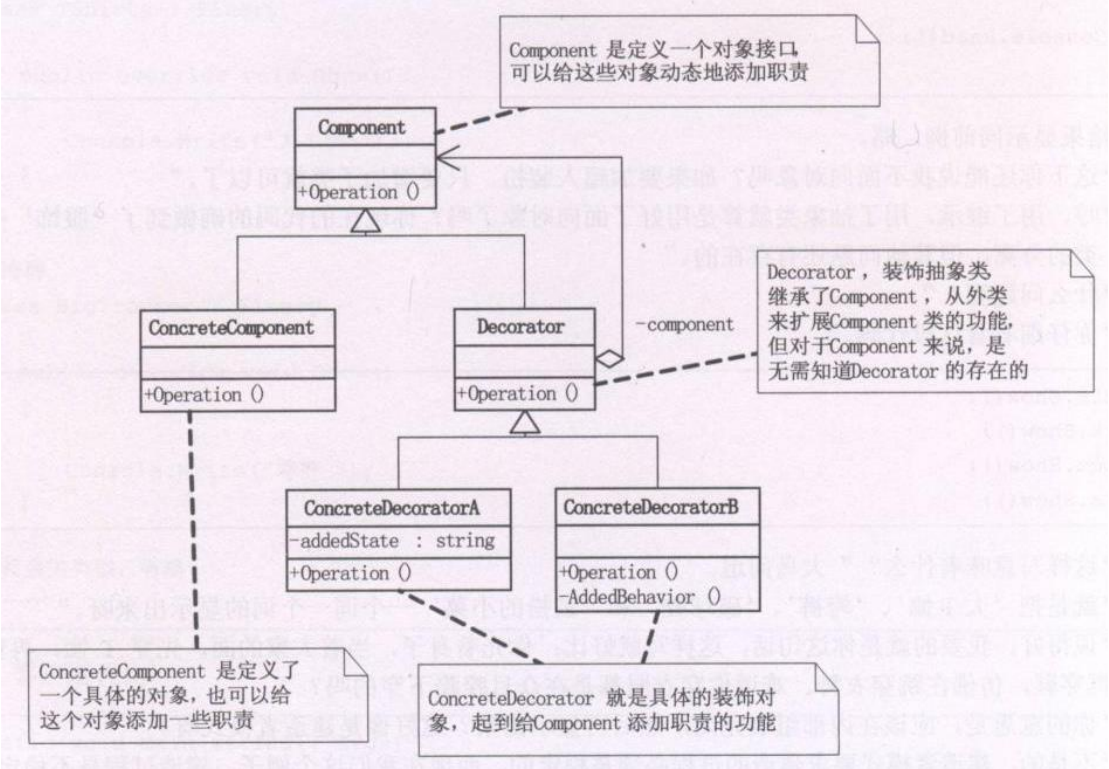
抽象不應該依賴細節，細節應該依賴抽象。即針對介面程式設計，不要對實現程式設計。

高層模組不能依賴低層模組，兩者都應依賴抽象。

依賴倒轉原則是物件導向的標誌，用哪種語言編寫程式不重要，如果編寫時考慮的是如何針對抽象程式設計而不是針對細節程式設計，即程式的所有依賴關係都終止於抽象類別或介面。那就是物件導向設計，反之那就是過程化設計。

（三）裝飾模式

動態地給一個物件添加一些額外的職責（不重要的功能，只是偶然一次要執行），就增加功能來說，裝飾模式比生成子類更為靈活。建造過程不穩定，按正確的順序串聯起來進行控制。



GOOD:當你向舊的類中添加新代碼時，一般是為了添加核心職責或主要行為。而當需要加入的僅僅是一些特定情況下才會執行的特定的功能時（簡單點就是不是核心應用的功能），就會增加類的複雜度。裝飾模式就是把要添加的附加功能分別放在單獨的類中，並讓這個類包含它要裝飾的物件，當需要執行時，用戶端就可以有選擇地、按順序地使用裝飾功能包裝物件。

例

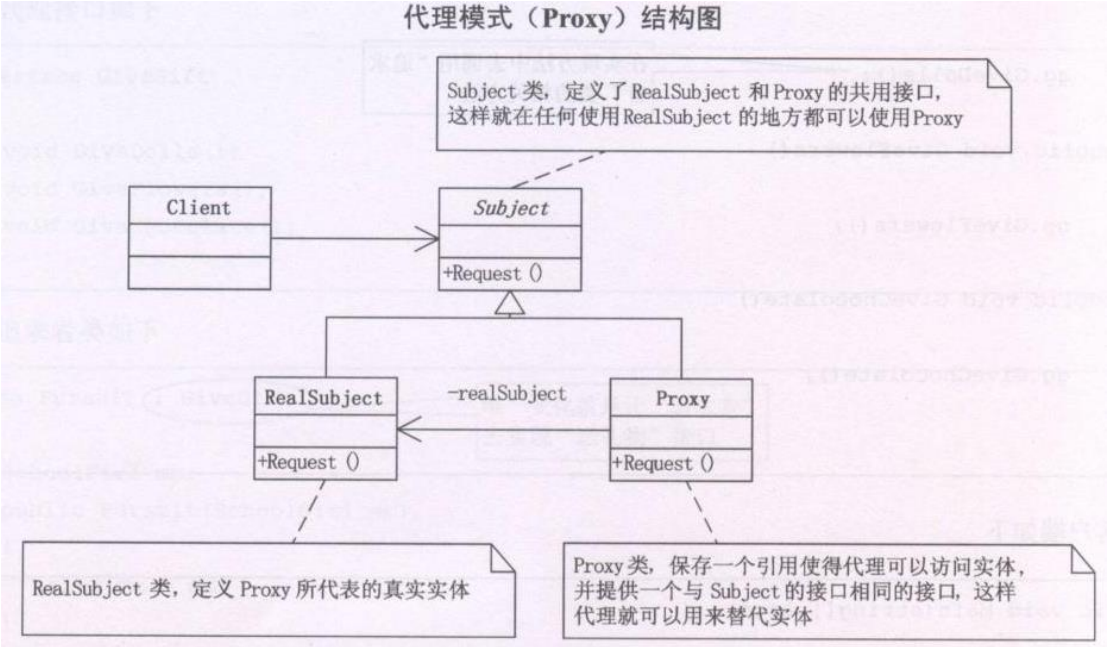
```
#include <string>
#include <iostream>
using namespace std;
//人
class Person
{
private:
    string m_strName;
public:
    Person(string strName)
    {
        m_strName=strName;
    }
    Person(){}
    virtual void Show()
    {
        cout<<"装扮的是:"<<m_strName<<endl;
    }
```

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

//用戶端
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：遠端代理，可以隱藏一個物件在不同位址空間的事實
虛擬代理：通過代理來存放需要很長時間產生實體的物件
安全代理：用來控制真實物件的存取權限
智能引用：當調用真實物件時，代理處理另外一些事



例：

```
#include <string>
#include <iostream>
using namespace std;
//定義介面
class Interface
{
public:
    virtual void Request()=0;
};
//真實類
class RealClass : public Interface
{
public:
```

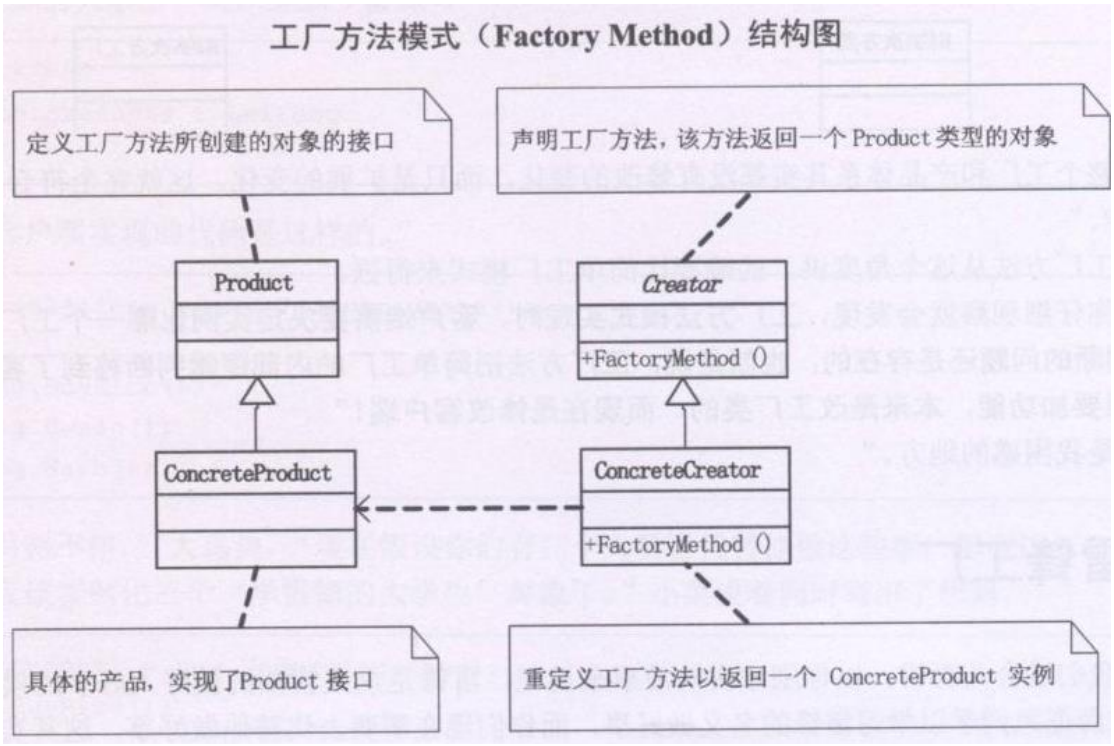
```
    virtual void Request()
    {
        cout<<"真實的請求"<<endl;
    }
};
//代理類
class ProxyClass : public Interface
{
private:
    RealClass* m_realClass;
public:
    virtual void Request()
    {
        m_realClass= new RealClass();
        m_realClass->Request();
        delete m_realClass;
    }
};
```

用戶端：

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

（五）工廠方法模式

GOOD：修正了簡單工廠模式中不遵守開放－封閉原則。工廠方法模式把選擇判斷移到了用戶端去實現，如果想添加新功能就不用修改原來的類，直接修改用戶端即可。



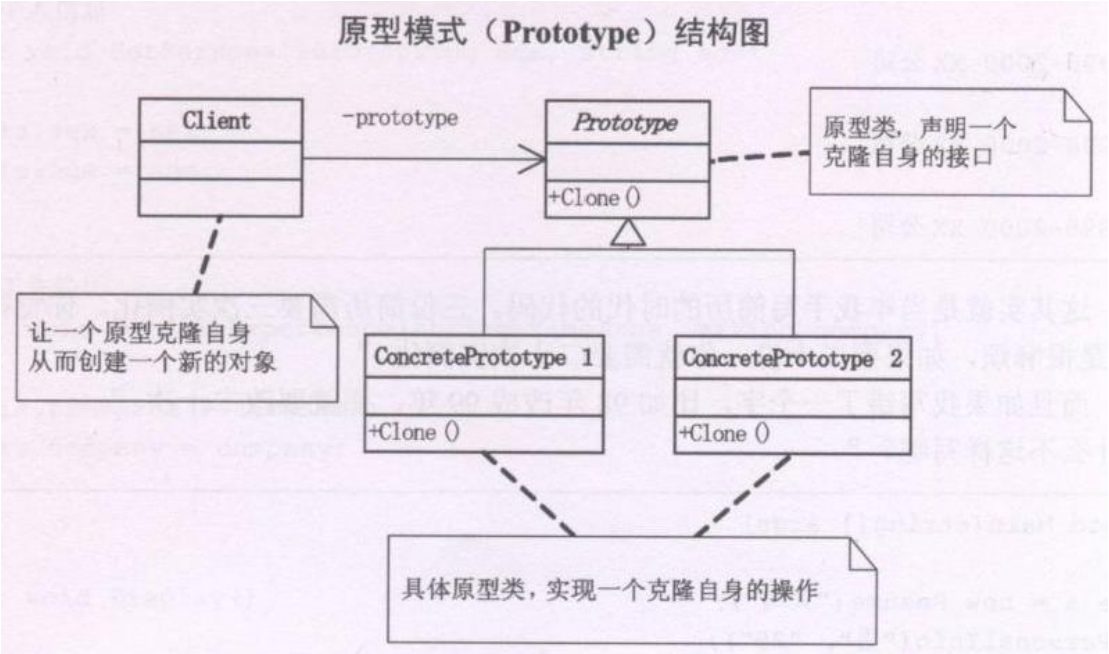
例：

```
#include <string>
#include <iostream>
using namespace std;
//實例基類，相當於 Product(為了方便，沒用抽象)
class LeiFeng
{
public:
	virtual void Sweep()
	{
		cout<<"雷鋒掃地"<<endl;
	}
};

//學雷鋒的大學生，相當於 ConcreteProduct
class Student: public LeiFeng
{
public:
	virtual void Sweep()
	{
		cout<<"大學生掃地"<<endl;
	}
};

//學雷鋒的志願者，相當於 ConcreteProduct
class Volenter: public LeiFeng
```

```
{
public :
	virtual void Sweep()
	{
		cout<<"志願者"<<endl;
	}
};
//工廠基類 Creator
class LeiFengFactory
{
public:
	virtual LeiFeng* CreateLeiFeng()
	{
		return new LeiFeng();
	}
};
//工廠具體類
class StudentFactory : public LeiFengFactory
{
public :
	virtual LeiFeng* CreateLeiFeng()
	{
		return new Student();
	}
};
class VolenterFactory : public LeiFengFactory
{
public:
	virtual LeiFeng* CreateLeiFeng()
	{
		return new Volenter();
	}
};
//用戶端
int main()
{
	LeiFengFactory *sf=new LeiFengFactory();
	LeiFeng *s=sf->CreateLeiFeng();
	s->Sweep();
	delete s;
	delete sf;
	return 0;
}
```

（六）原型模式

GOOD：從一個物件再創建另外一個可定制的物件，而無需知道任何創建的細節。並能提高創建的性能。 說白了就 COPY 技術，把一個物件完整的 COPY 出一份。



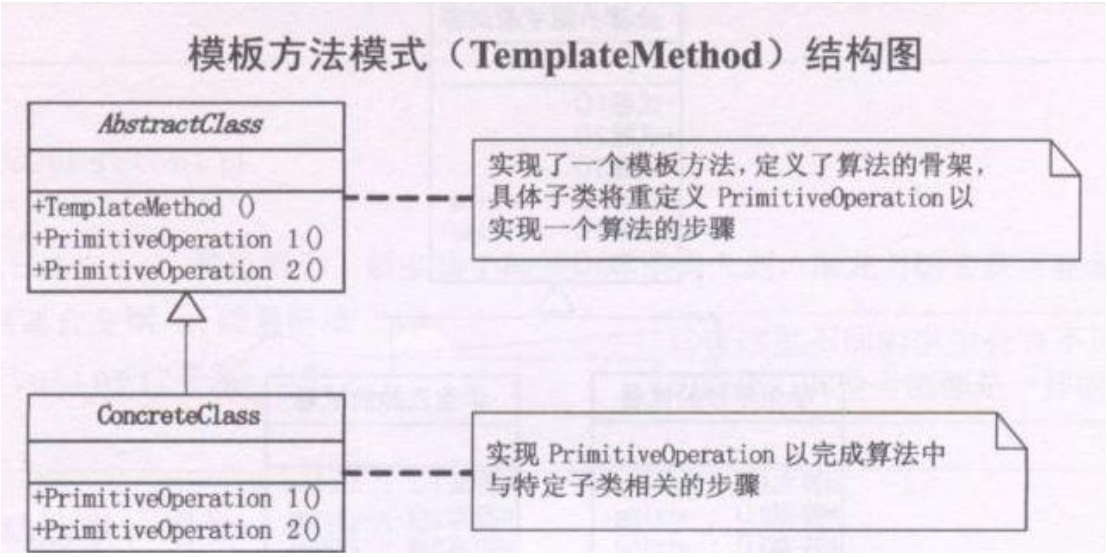
例：

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

class Prototype //抽象基類
{
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 ; //複製對象
        return p ;
    }
} ;

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

    virtual Prototype* Clone()
    {
        ConcretePrototype2 *p = new ConcretePrototype2() ;
        *p = *this ; //複製對象
        return p ;
    }
} ;

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

（七）範本方法模式

GOOD：把不變的代碼部分都轉移到父類中，將可變的代碼用 virtual 留到子類重寫



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 "火影史上最帥的六代目---一鳴驚人 naruto";
	}
};

class OnePice : public AbstractClass
{
protected:
```

```
        virtual string GetName()
        {
            return "我是無惡不做的大海賊---路飛";
        }
};

//用戶端
int main()
{
        Naruto* man = new Naruto();
        man->Show();

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

        return 0;
}
```

迪米特法則

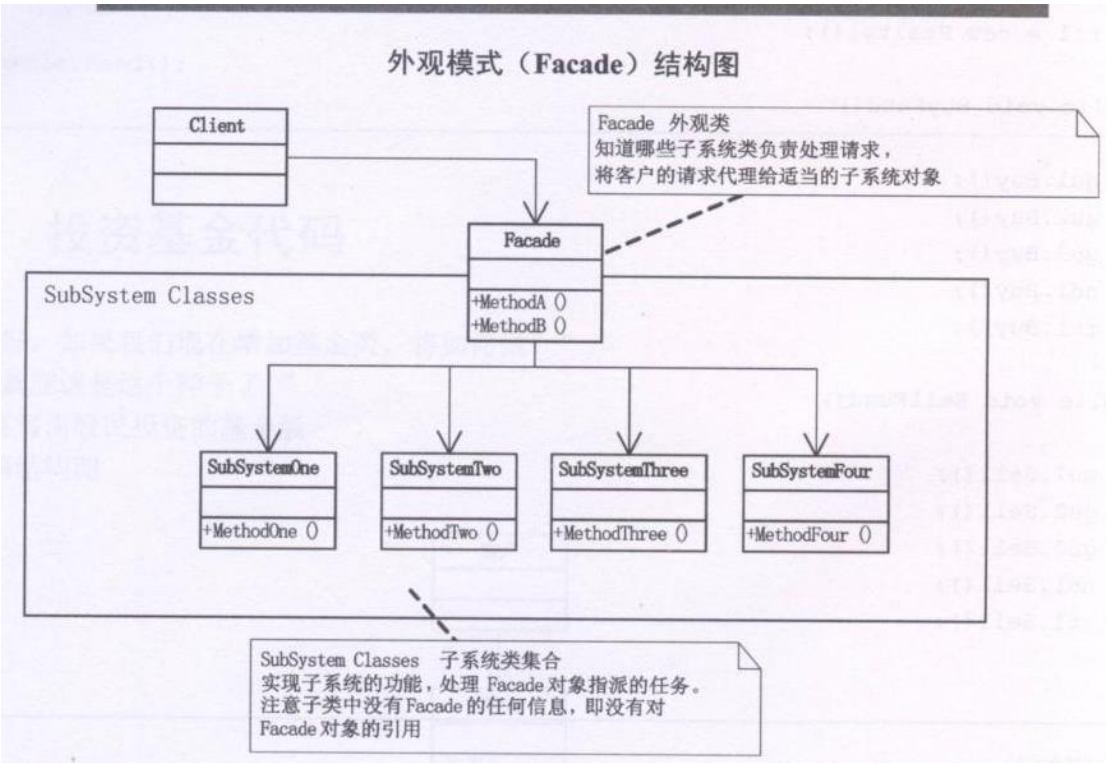
如果兩個類不直接通信，那麼這兩個類就不應當發生直接的相互作用。如果一個類需要調用另一個類的某個方法的話，可以通過第三個類轉發這個調用。

在類的結構設計上，每一個類都應該儘量降低成員的存取權限。

該法則在後面的適配器模式、解釋模式等中有強烈的體現。

（八）面板模式

GOOD：為子系統的一組介面提供一個一致的介面。使用戶使用起來更加方便。



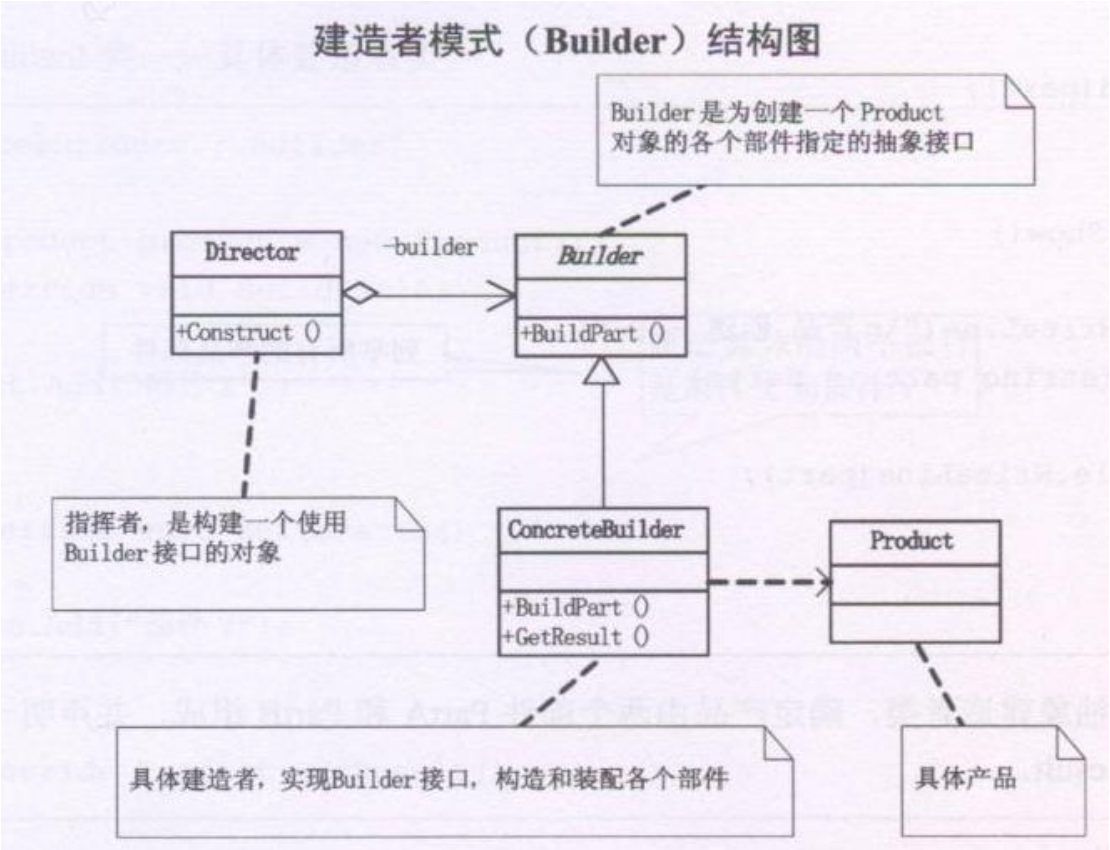
例：

```
#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;
	}
};

//外觀類
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();
	}
};

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

（九）建造者模式（生成器模式）

GOOD：在當創建複雜物件的演算法應該獨立於該物件的組成部分以及它們的裝配方式時適用。（P115 頁）



例：

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

//最終的產品類
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++;
            }
        }
};

//建造者基類
class Builder
{
public:
        virtual void BuilderA()=0;
        virtual void BuilderB()=0;
        virtual Product* GetResult()=0;
};
//第一種建造方式
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;
        }
};
//第二種建造方式
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;
    }
};

//指揮者類
class Direct
{
public:
    void Construct(Builder* temp)
    {
        temp->BuilderA();
        temp->BuilderB();
    }
};

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

    p->Construct(b1);               //調用第一種方式
    Product* pb1=b1->GetResult();
    pb1->Show();

    p->Construct(b2);               //調用第二種方式
    Product * pb2=b2->GetResult();
    pb2->Show();
```

```
    return 0;
}
```

例二（其實這個例子應該放在前面講的）：

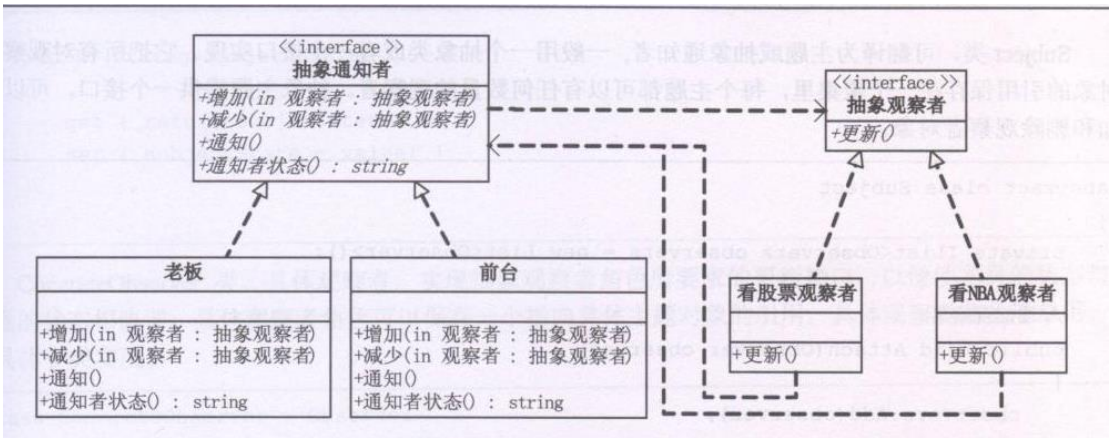
```
#include <string>
#include <iostream>
#include <vector>
using namespace std;

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;
    }
};
//指揮者類
class Direct
{
private:
    Person* p;
public:
    Direct(Person* temp) { p = temp;}
    void Create()
    {
        p->CreateHead();
        p->CreateBody();
        p->CreateHand();
        p->CreateFoot();
    }
};

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

（十）觀察者模式

GOOD：定義了一種一對多的關係，讓多個觀察物件（公司員工）同時監聽一個

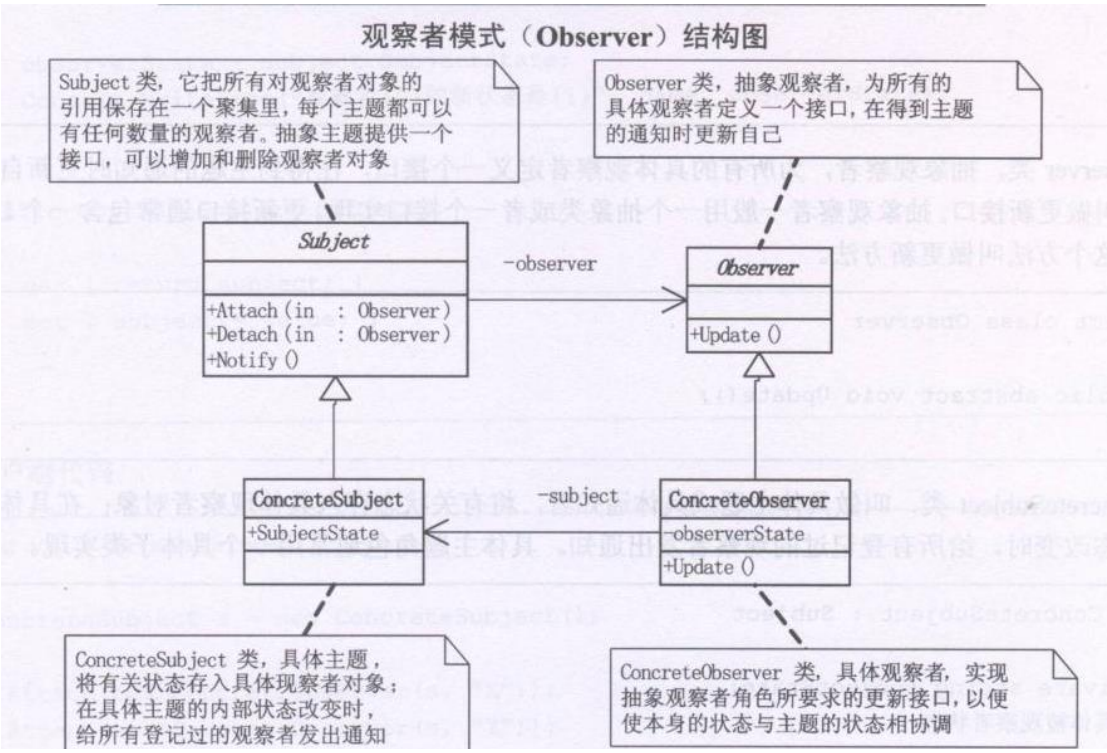
主題物件（秘書），主題物件狀態發生變化時，會通知所有的觀察者，使它們能夠更新自己。



例：

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

class Secretary;
//看股票的同事類（觀察對象，觀察者）
class StockObserver
{
private:
	string name;
	Secretary* sub;
public:
	StockObserver(string strname,Secretary* strsub)
	{
		name=strname;
		sub=strsub;
	}
	void Update();
};
//秘書類（主題物件，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，要開始工作了"<<endl;
}

//用戶端
int main()
{
    Secretary *p=new Secretary();   //創建通知者

    //觀察者
    StockObserver *s1= new StockObserver("小李",p);
    StockObserver *s2 = new StockObserver("小趙",p);
    //加入通知佇列
    p->Add(*s1);
    p->Add(*s2);
    //事件
    p->action="老闆來了";
    //通知
    p->Notify();
    return 0;
}
```



例：

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

class SecretaryBase;
//抽象觀察者
class CObserverBase
{
protected:
	string name;
	SecretaryBase* sub;
public:
	CObserverBase(string strname,SecretaryBase* strsub)
	{
		name=strname;
		sub=strsub;
	}
	virtual void Update()=0;
};
//具體的觀察者，看股票的
class StockObserver : public CObserverBase
{
public:
```

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

    //具體觀察者，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;
    };

    //具體通知者
    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++;
            }
        }
```

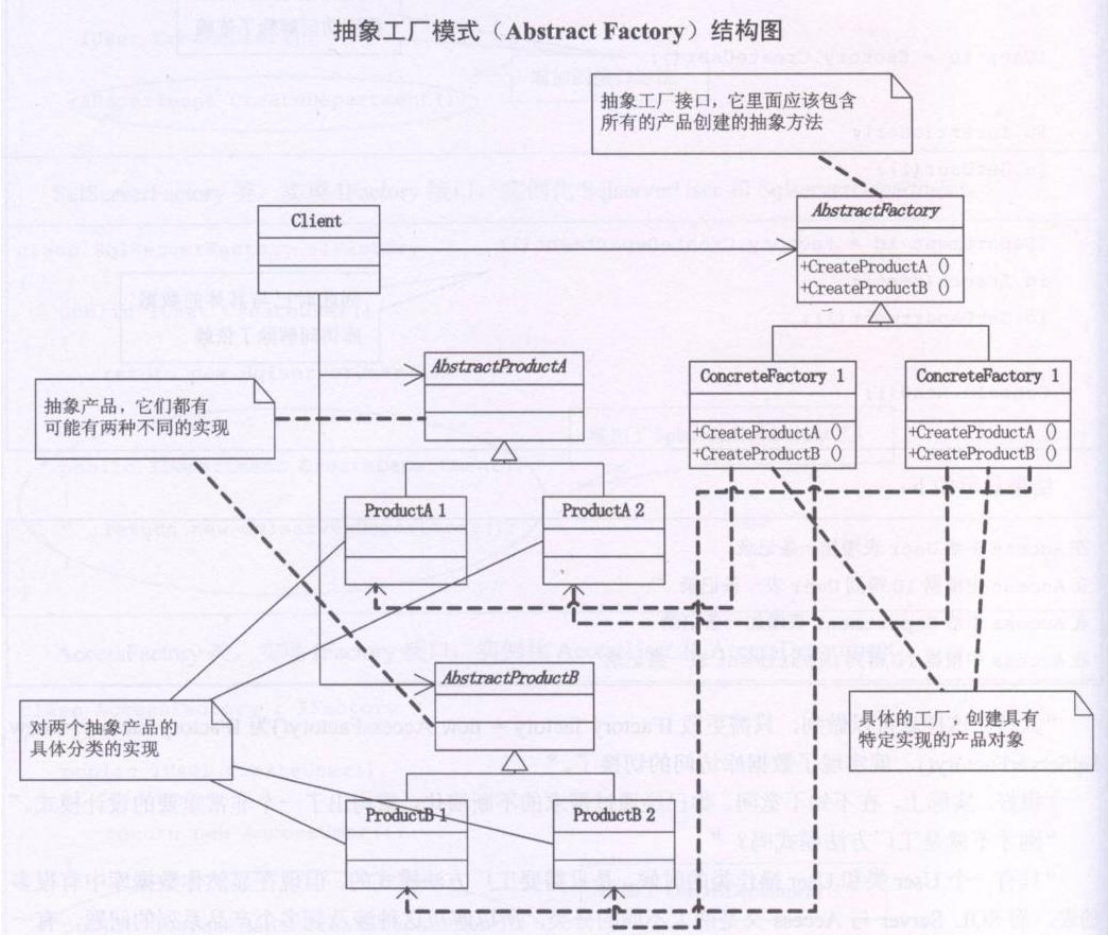
```
};

void StockObserver::Update()
{
	cout<<name<<":"<<sub->action<<",不要玩股票了，要開始工作了"<<endl;
}
void NBAObserver::Update()
{
	cout<<name<<":"<<sub->action<<",不要看 NBA 了，老闆來了"<<endl;
}


用戶端：
int main()
{
	SecretaryBase *p=new Secretary();  //創建觀察者

	//被觀察的對象
	CObserverBase *s1= new NBAObserver("小李",p);
	CObserverBase *s2 = new StockObserver("小趙",p);
	//加入觀察佇列
	p->Attach(s1);
	p->Attach(s2);
	//事件
	p->action="老闆來了";
	//通知
	p->Notify();

	return 0;
}
```

（十一）抽象工廠模式

GOOD：定義了一個創建一系列相關或相互依賴的介面，而無需指定它們的具體類。

用於交換產品系列，如 ACCESS->SQL SERVER；

產品的具體類名被具體工廠的實現分離



例：

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

//使用者抽象介面
class IUser
{
public :
	virtual void GetUser()=0;
	virtual void InsertUser()=0;
};

//部門抽象介面
class IDepartment
{
public:
	virtual void GetDepartment()=0;
	virtual void InsertDepartment()=0;
```

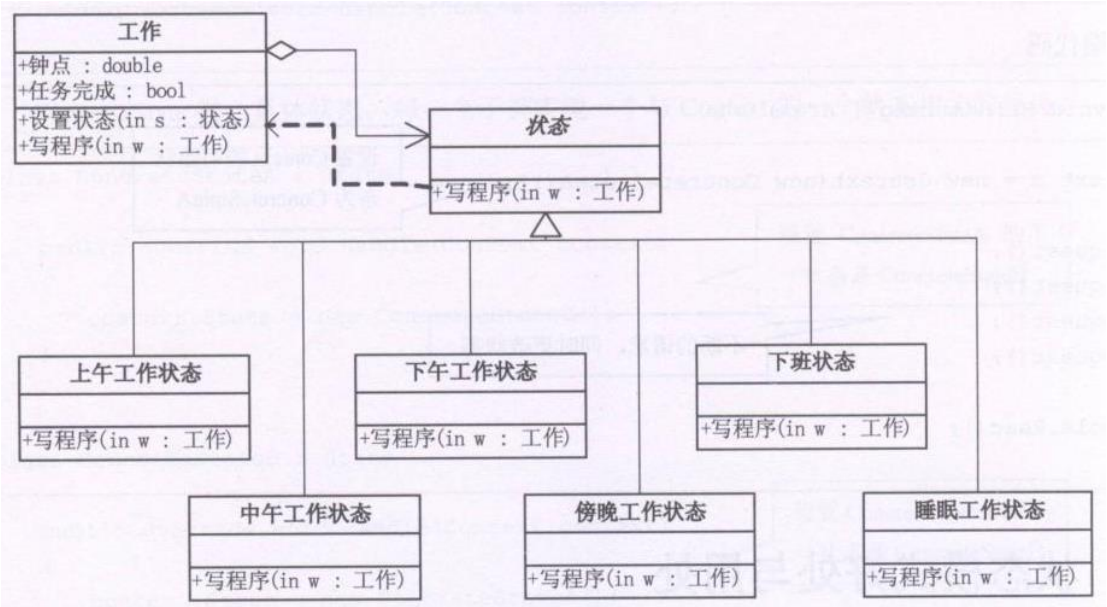
```
};

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

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

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

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

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

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

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

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

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

（十二）狀態模式

GOOD：當一個物件的行為取決於它的狀態，並且它必須在運行時刻根據狀態改變它的行為時，可考慮用到狀態模式。



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<<"還不錯啦"<<endl;
		else
			cout<<"不行了，還是睡覺吧"<<endl;
	}
};

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

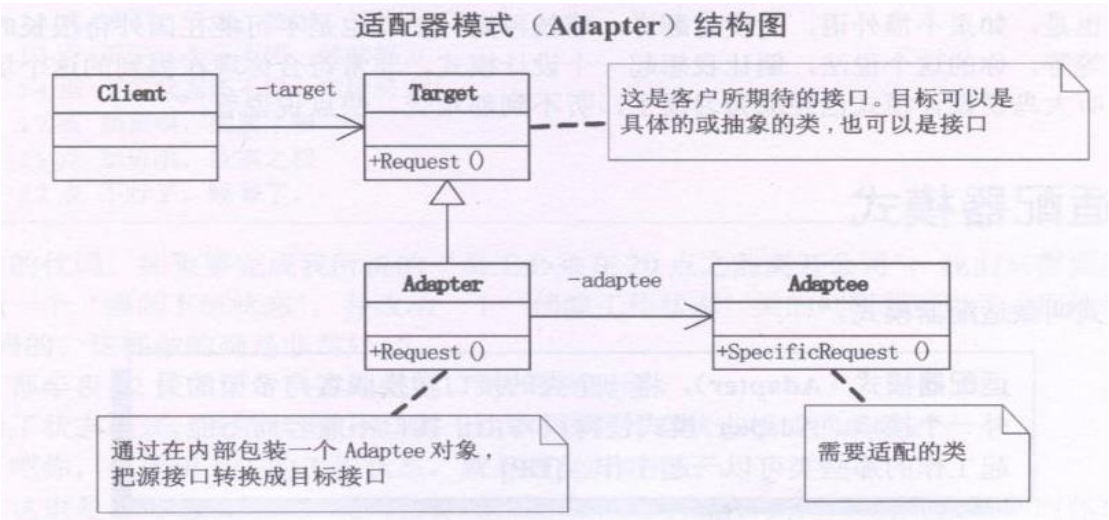
```
            if((w->hour)<12)
                cout<<"現在的精神無敵好"<<endl;
            else
            {
                w->SetState(new NoonState());
                w->Writeprogram(); //注意加上這句
            }
        }
};


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

用戶端：

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

（十三）適配器模式

GOOD：雙方都不適合修改的時候，可以考慮使用適配器模式



例：

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

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

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

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

用戶端：

```
int main()
{
```

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

例二

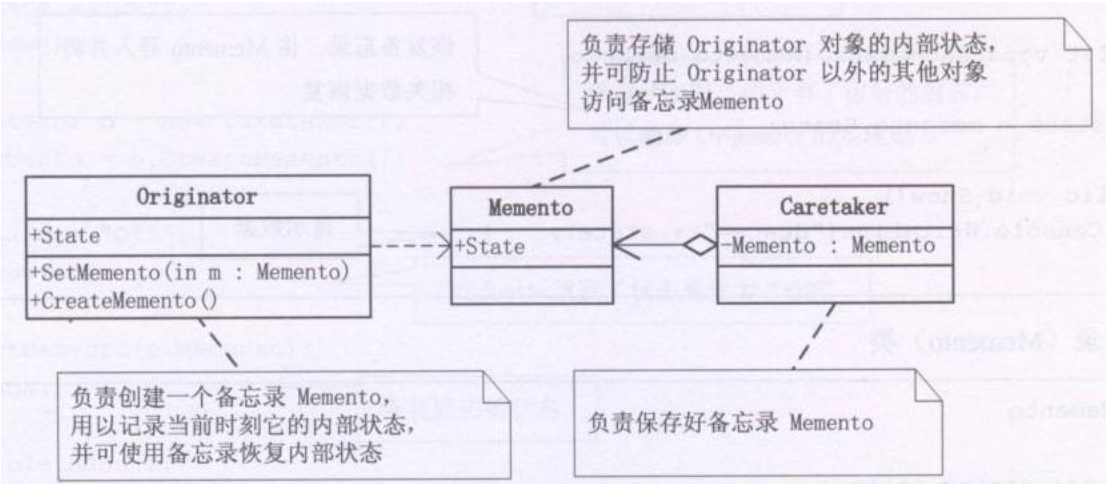
```
#include <iostream>
#include <string>
using namespace std;

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<<"前鋒進攻"<<endl;
    }
    virtual void Defense()
    {
        cout<<name<<"前鋒防守"<<endl;
    }
};

class Center : public Player
{
public:
    Center(string strName):Player(strName){}
public:
    virtual void Attack()
    {
        cout<<name<<"中場進攻"<<endl;
```

```
        }
        virtual void Defense()
        {
            cout<<name<<"中場防守"<<endl;
        }
};

//為中場翻譯
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();
        }
};


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

（十四）備忘錄模式

GOOD：在不破壞封裝性的前提下，捕獲一個物件的內部狀態，並在該物件之外保存這個狀態，這樣就可以將以後的物件狀態恢復到先前保存的狀態。

適用於功能比較複雜的，但需要記錄或維護屬性歷史的類；或者需要保存的屬性只是眾多屬性中的一小部分時 Originator 可以根據保存的 Memo 還原到前一狀態。



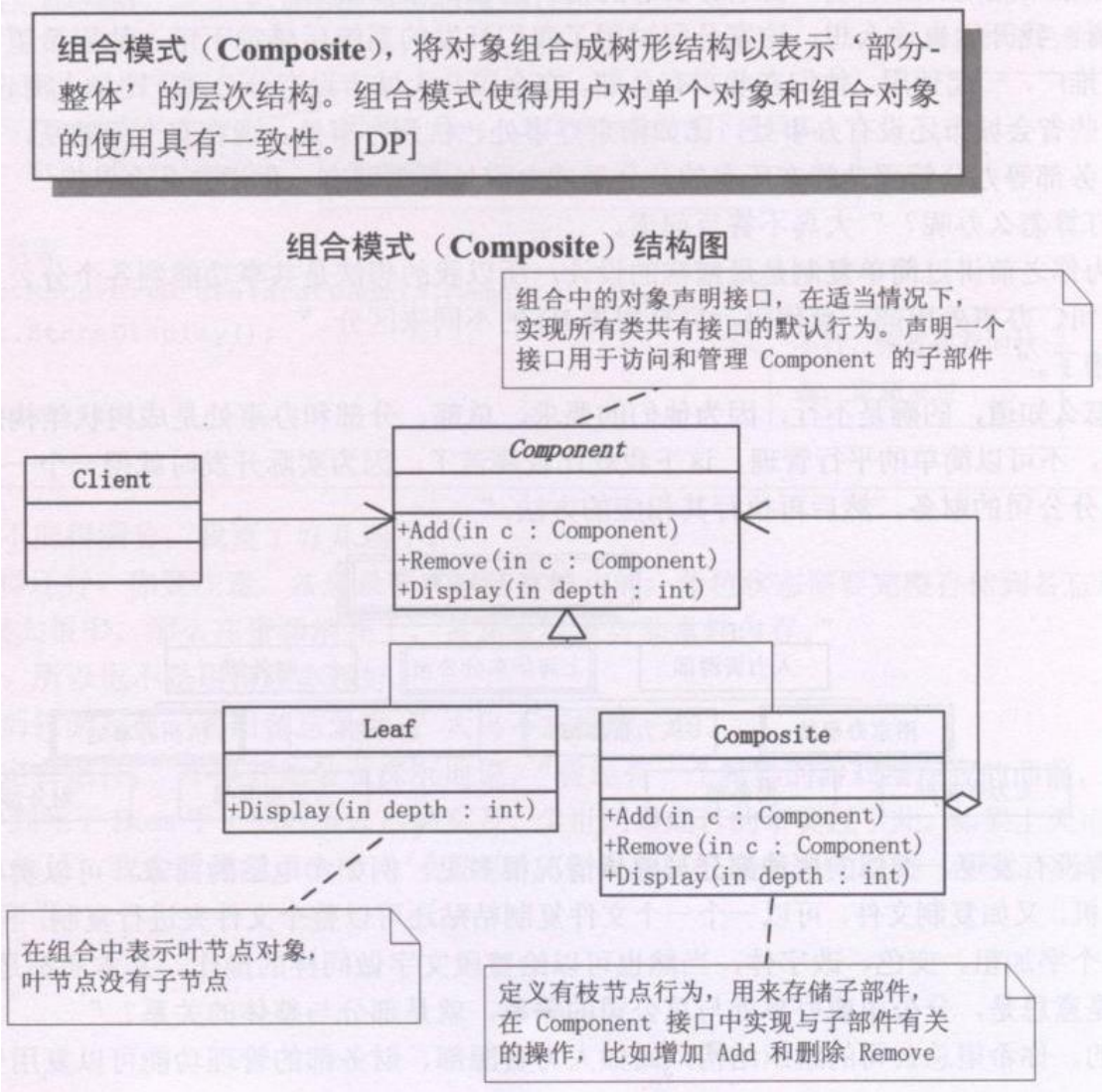
例：

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

class Memo;

//發起人類
class Originator
{
public:
	string state;
	Memo* CreateMemo();
	void SetMemo(Memo* memo);
	void Show()
	{
		cout<<"狀態："<<state<<endl;
	}
};
//備忘錄類
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;
}

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

用戶端：
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）組合模式

GOOD：整體和部分可以被一致對待（如 WORD 中複製一個文字、一段文字、一篇文章都是一樣的操作）

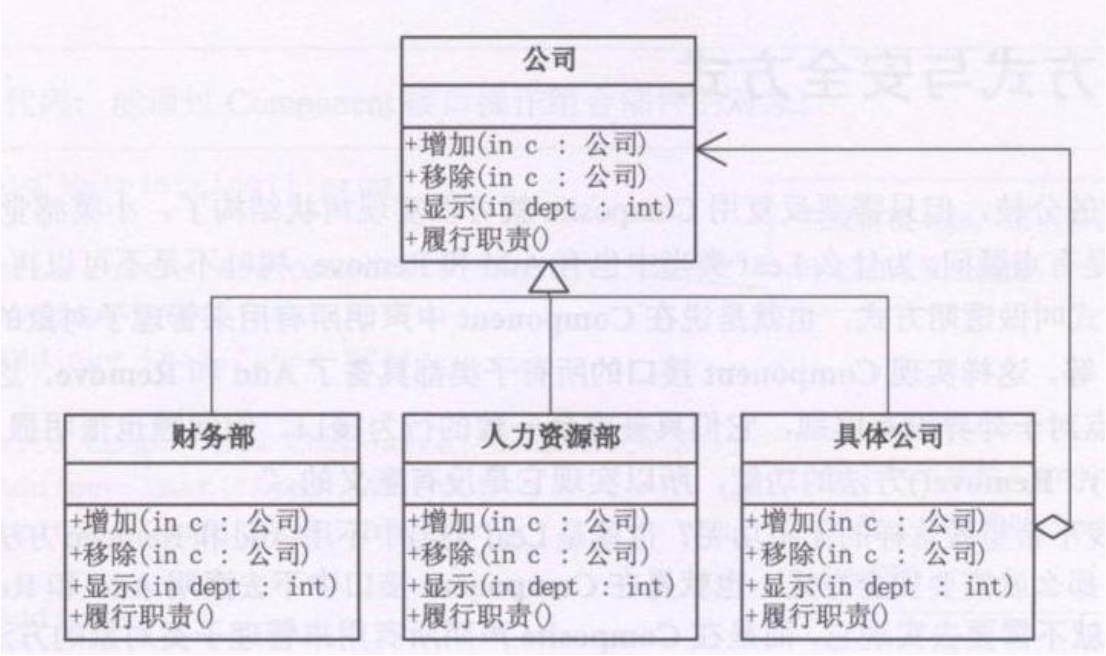


例：

```
#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++;
        }
    }

};

//用戶端
#include "Model.h"

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

    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())
```

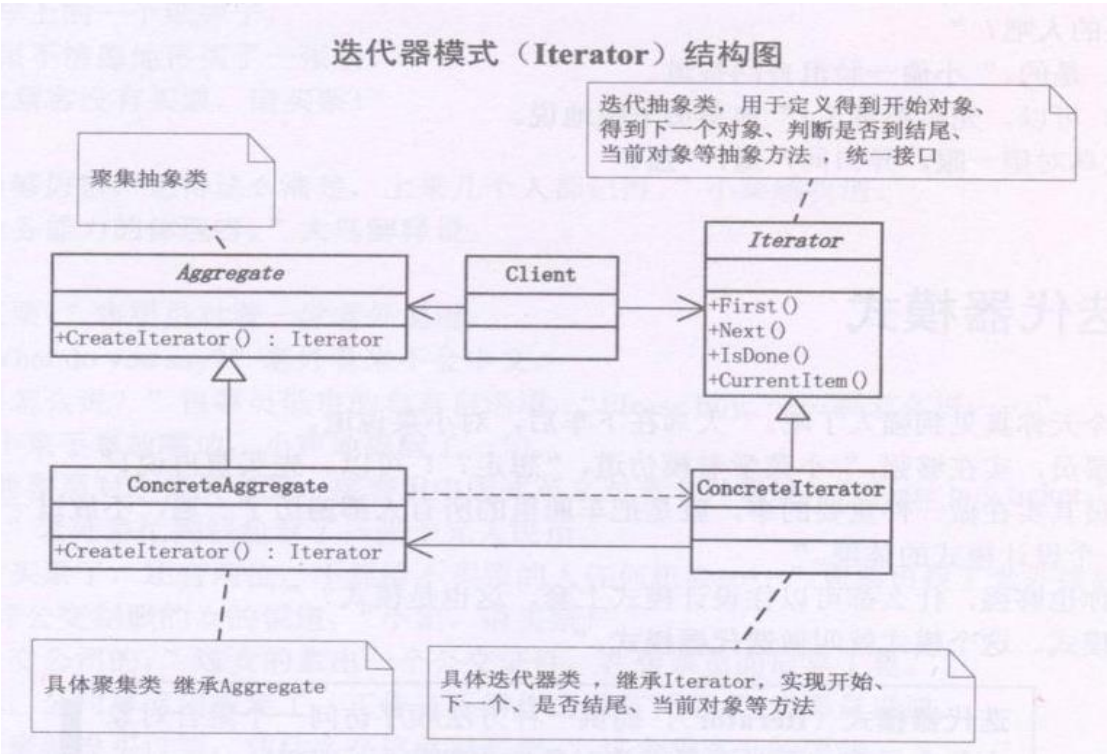
```
            {
                (*p)->Display(nDepth+2);
                p++;
            }
        }
        virtual void LineOfDuty()
        {
            vector<Company*>::iterator p=m_company.begin();
            while (p!=m_company.end())
            {
                (*p)->LineOfDuty();
   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

```
//用戶端：
int main()
{
    ConcreteCompany *p = new ConcreteCompany("清華大學");
    p->Add(new HrDepartment("清華大學人才部"));

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

    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）反覆運算器模式

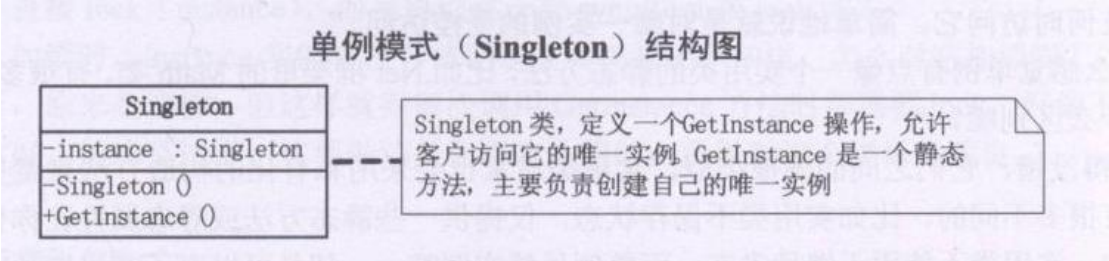
GOOD：提供一種方法循序存取一個聚斂物件的各個元素，而又不暴露該物件的內部表示。

為遍歷不同的聚集結構提供如開始，下一個，是否結束，當前一項等統一介面。



（十七）單例模式

GOOD：保證一個類僅有一個實例，並提供一個訪問它的全域訪問點



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;//注意靜態變數類外初始化
```

用戶端：

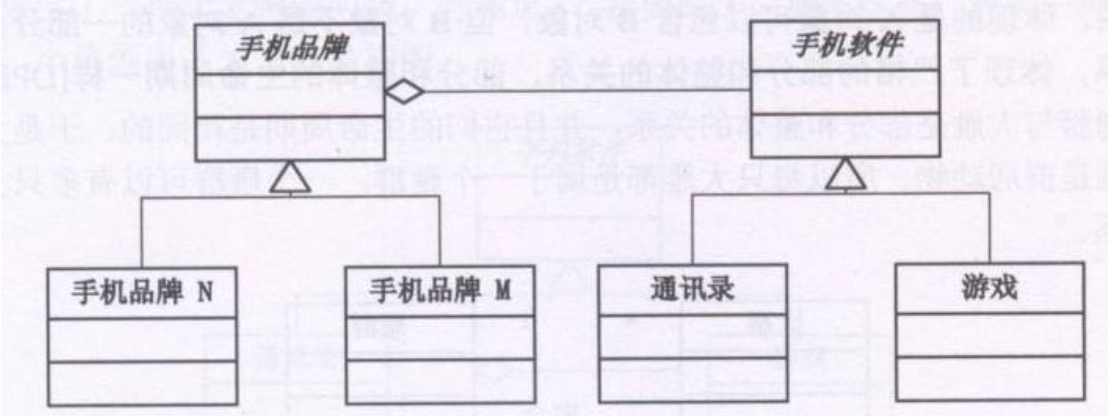
```
int main()
{
    Singelton* s1=Singelton::GetInstance();
    Singelton* s2=Singelton::GetInstance();
    if(s1 == s2)
        cout<<"ok"<<endl;
    else
        cout<<"no"<<endl;
```

```
    return 0;
}
```

（十八）橋接模式

GOOD：將抽象部分與實現部分分離，使它們可以獨立變化。

這裡說的意思不是讓抽象基類與具體類分離，而是現實系統可能有多角度分類，每一種分類都有可能變化，那麼把這種多角度分離出來讓它們獨立變化，減少它們之間的耦合性，即如果繼承不能實現“開放－封閉原則”的話，就應該考慮用橋接模式。如下例：讓“手機”既可以按品牌分類也可以



例：

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

//手機軟體
class HandsetSoft
{
public:
    virtual void Run()=0;
};

//遊戲軟體
class HandsetGame : public HandsetSoft
{
public:
    virtual void Run()
    {
        cout<<"運行手機遊戲"<<endl;
    }
};

//通訊錄軟體
```

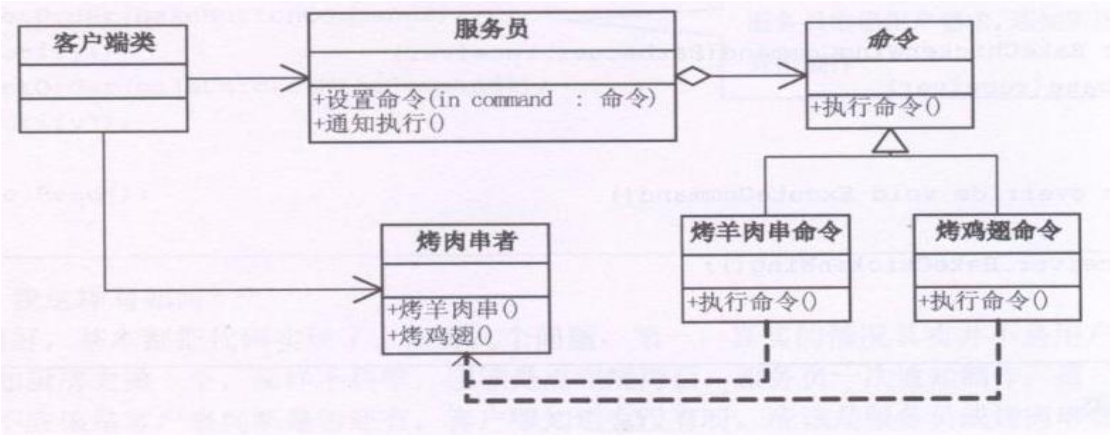
```
class HandSetAddressList : public HandsetSoft
{
public:
	virtual void Run()
	{
		cout<<"手機通訊錄"<<endl;
	}
};

//手機品牌
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();
	}
};

//用戶端
int main()
```

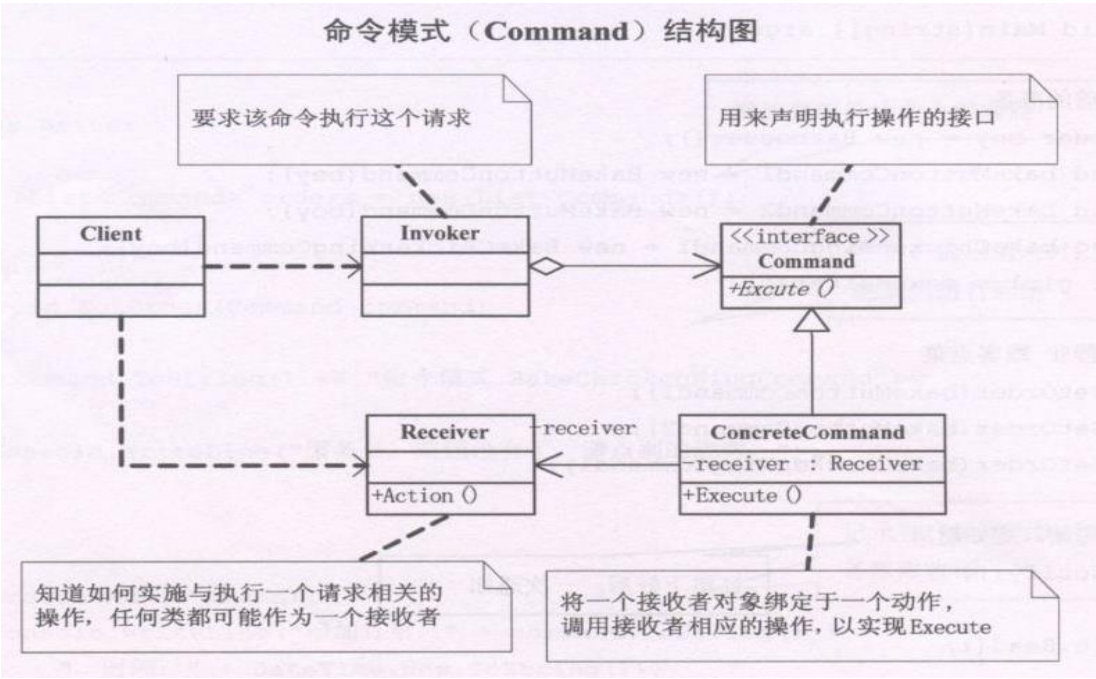
```
{
    HandsetBrand *brand;
    brand = new HandsetBrandM();
    brand->SetHandsetSoft(new HandsetGame());
    brand->Run();
    brand->SetHandsetSoft(new HandSetAddressList());
    brand->Run();

    return 0;
}
```

（十九）命令模式

GOOD：一、建立命令佇列；二、可以將命令記入日誌；三、接收請求的一方可以拒絕；四、添加一個新命令類不影響其它類；

命令模式把請求一個操作的物件與知道怎麼操行一個操作的物件分開





例：

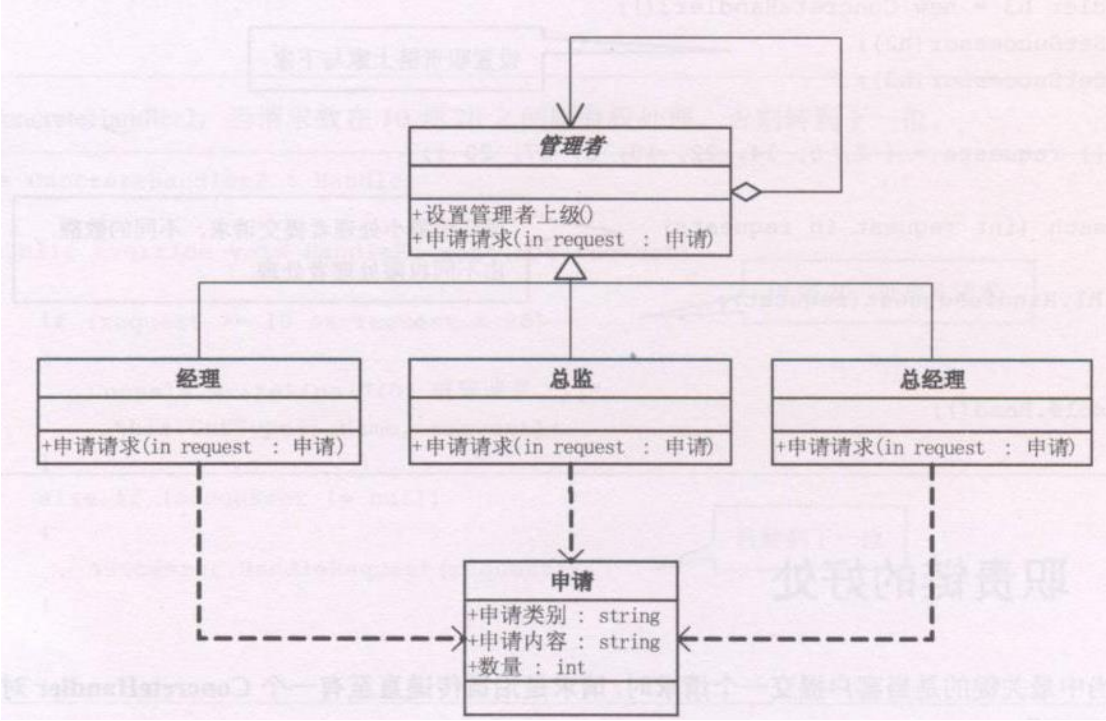
```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

//烤肉師傅
class Barbucer
{
public:
	void MakeMutton()
	{
		cout<<"烤羊肉"<<endl;
	}
	void MakeChickenWing()
	{
		cout<<"烤雞翅膀"<<endl;
	}
};

//抽象命令類
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();
	}
};
```

```
//烤雞翅
class ChickenWingCmd : public Command
{
public:
	ChickenWingCmd(Barbucer* temp) : Command(temp){}

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

//服務員類
class Waiter
{
protected:
	vector<Command*> m_commandList;
public:
	void SetCmd(Command* temp)
	{
		m_commandList.push_back(temp);
		cout<<"增加定單"<<endl;
	}

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

//用戶端
int main()
{
	//店裡添加烤肉師傅、功能表、服務員等顧客
	Barbucer* barbucer=new Barbucer();
	Command* cmd= new BakeMuttonCmd(barbucer);
	Command* cmd2=new ChickenWingCmd(barbucer);
	Waiter* girl = new Waiter();
```

```
    //點菜
    girl->SetCmd(cmd);
    girl->SetCmd(cmd2);
    //服務員通知
    girl->Notify();
    return 0;
}
```

（二十）責任鏈模式

GOOD：使多個物件都有機會處理請求，從而避免請求的發送者和接收者之間的耦合關係。將這個物件連成一條鏈，並沿著這條鏈傳遞該請求，直到有一個物件處理為止。



例：

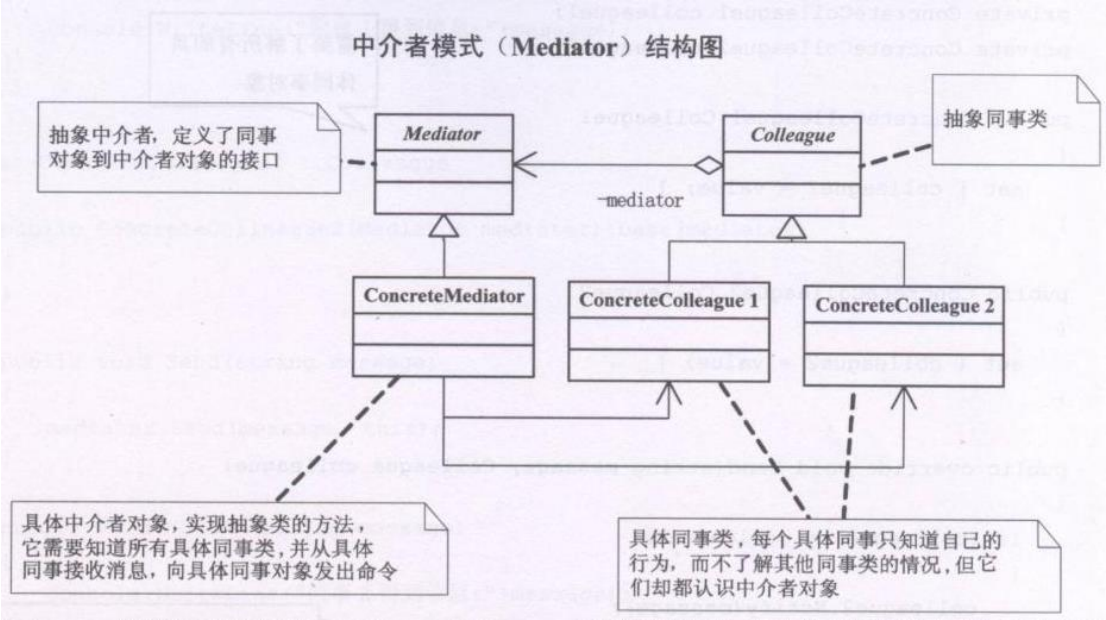
```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
//請求
class Request
{
public:
    string m_strContent;
    int m_nNumber;
};
//管理者
```

```
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;
};
//經理
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<<"處理了"<<request->m_nNumber<<"個請求"<<endl;
		}
		else
		{
			manager->GetRequest(request);
		}
	}
};
//總監
class MajorDomo : public Manager
{
public:
	MajorDomo(string name) : Manager(name){}

	virtual void GetRequest(Request* request)
	{
		if(request->m_nNumber>=10)
		{
			cout<<name<<"處理了"<<request->m_nNumber<<"個請求"<<endl;
```

```
            }
        }
};

//用戶端
int main()
{
        Manager * common = new CommonManager("張經理");
        Manager * major = new MajorDomo("李總監");

        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）仲介者模式

GOOD：用一個仲介物件來封裝一系列的物件交互，仲介者使各物件不需要顯示的相互引用，從而降低耦合；而且可以獨立地改變它們之間的交互。



例：

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

```
using namespace std;

class Colleague;
//仲介者類
class Mediator
{
public:
	virtual void Send(string message,Colleague* col) = 0;
};
//抽象同事類
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<<"同事一獲得了消息"<<strMessage<<endl;
	}
};

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

	void Send(string strMessage)
```

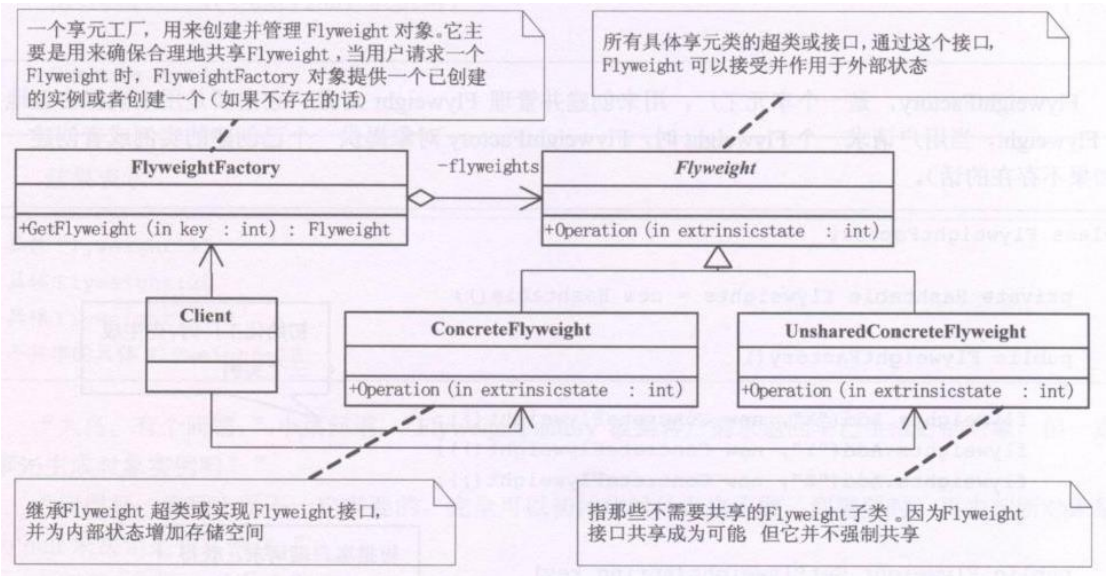
```
    {
        mediator->Send(strMessage,this);
    }

    void Notify(string strMessage)
    {
        cout<<"同事二獲得了消息"<<strMessage<<endl;
    }
};

//具體仲介者類
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);
    }
};

//用戶端：
int main()
{
    ConcreteMediator * m = new ConcreteMediator();

    //讓同事認識仲介
    Colleague1* col1 = new Colleague1(m);
    Colleague2* col2 = new Colleague2(m);

    //讓仲介認識具體的同事類
    m->col1 = col1;
    m->col2 = col2;

    col1->Send("吃飯了嗎？");
    col2->Send("還沒吃，你請嗎？");
    return 0;
}
```

（二十二）享元模式

GOOD：運用共用技術有效地支援大量細細微性的物件（對於 C++來說就是共用一個區塊啦，物件指標指向同一個地方）。

如果一個應用程式使用了大量的物件，而這些物件造成了很大的存儲開銷就應該考慮使用。

還有就是物件的大多數狀態可以外部狀態，如果刪除物件的外部狀態，那麼可以用較少的共用物件取代多組物件，此時可以考慮使用享元。



例：

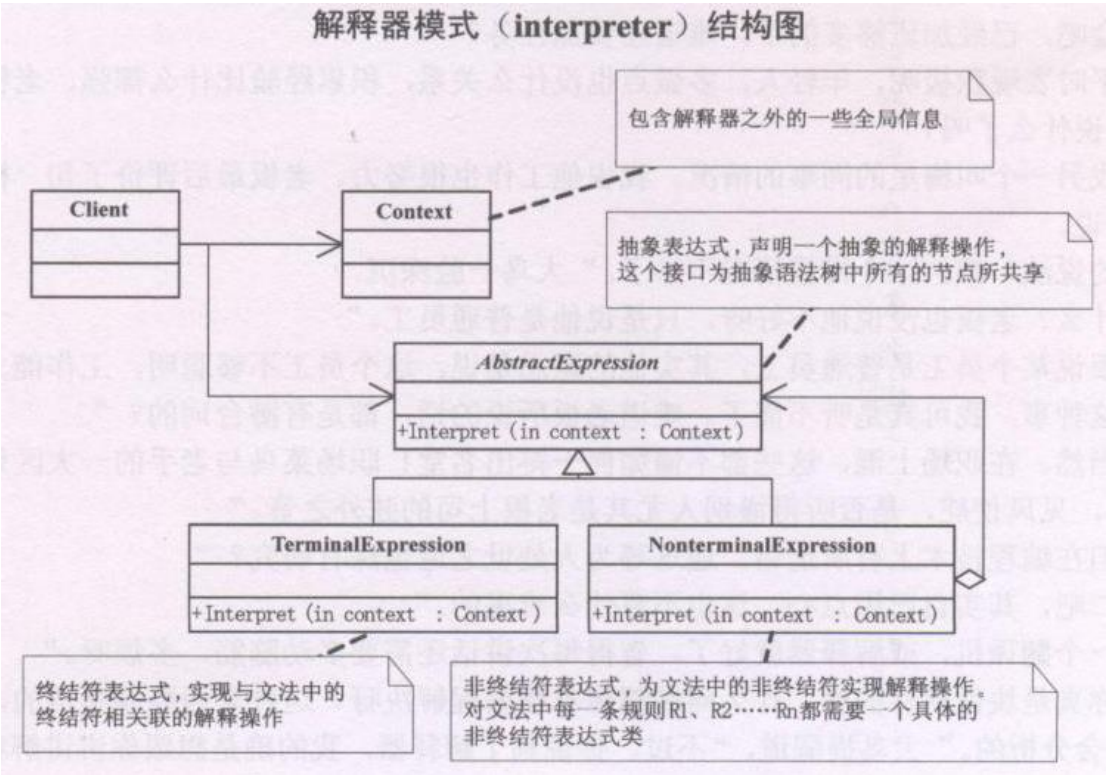
```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

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

//具體的共用網站
class ConcreteWebSite : public WebSite
{
private:
	string name;
public:
	ConcreteWebSite(string strName)
	{
		name = strName;
	}
	virtual void Use()
```

```
        {
            cout<<"網站分類："<<name<<endl;
        }
};

//不共用的網站
class UnShareWebSite : public WebSite
{
private:
        string name;
public:
        UnShareWebSite(string strName)
        {
            name = strName;
        }
        virtual void Use()
        {
            cout<<"不共用的網站："<<name<<endl;
        }
};

//網站工廠類，用於存放共用的 WebSite 物件
class WebFactory
{
private:
        vector<WebSite*> websites;
public:
        WebSite* GetWeb()
        {
            vector<WebSite*>::iterator p = websites.begin();
            return *p;
        }
        WebFactory()
        {
            websites.push_back(new ConcreteWebSite("測試"));
        }
};

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

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

    //不共用的類
    WebSite* ws3 = new UnShareWebSite("測試");
    ws3->Use();
    return 0;
}
```

（二十三）解譯器模式

GOOD：通常當一個語言需要解釋執行，並且你可以將該語言中的句子表示成為一個抽象的語法樹時，可以使用解譯器模式。



例：

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

class Context;
class AbstractExpression
{
public:
```

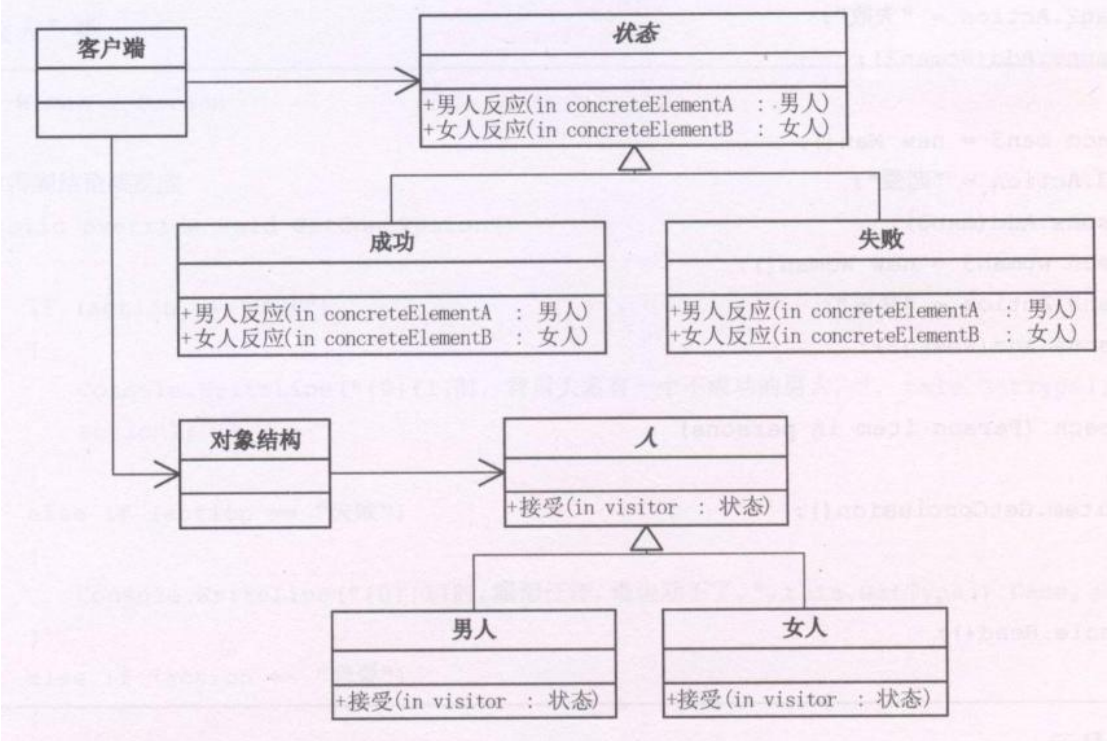
```
    virtual void Interpret(Context* context)=0;
};

class Expression : public AbstractExpression
{
public:
    virtual void Interpret(Context* context)
    {
        cout<<"終端解譯器"<<endl;
    };
};

class NonterminalExpression : public AbstractExpression
{
public:
    virtual void Interpret(Context* context)
    {
        cout<<"非終端解譯器"<<endl;
    }
};

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

//用戶端
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）訪問者模式

GOOD：適用於資料結構穩定的系統。它把資料結構和作用於資料結構上的操作分離開，使得操作集合

優點：新增加操作很容易，因為增加新操作就相當於增加一個訪問者，訪問者模式將有關的行為集中到一個訪問者物件中



例：

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

class Man;
class Woman;
//行為
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<<"男人成功時，背後有個偉大的女人"<<endl;
	}
	virtual void GetWomanConclusion(Woman* concreteElementB)
	{
		cout<<"女人成功時，背後有個沒用的男人"<<endl;
	}
};

//失敗
class Failure : public Action
{
public:
	virtual void GetManConclusion(Man* concreteElementA)
	{
		cout<<"男人失敗時，背後有個偉大的女人"<<endl;
	}
	virtual void GetWomanConclusion(Woman* concreteElementB)
	{
		cout<<"女人失敗時，背後有個沒用的男人"<<endl;
	}
};

//抽象人類
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);
	}
};

//物件結構類
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++;
		}
	}
};

//用戶端
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;
}
```