设计模式学习笔记

——udc577

01、简单工厂模式、工厂方法、抽象工厂、策略模式

工厂主要用于创建对象。策略模式则引入间接层，不直接返回对象。

简单工厂方法

根据输入条件创建不同的类实例。新增产品类时需要改动工厂类，且客户端要知道工厂类的实现（即要知道输入条件对应什么产品类）。其相似：策略模式。其改进形式：工厂方法。

```
Class COperation {
    Int m_first;
    Int m_second;

    Virtual int GetResult() {
        Return 0;
    }
};
```

```
Class AddOperation : public COperation {
    Virtual int GetResult() {
        Return m_first + m_second;;
    }
};

Class SubOperation : public COperation {
    Virtual int GetResult() {
        Return m_first - m_second;
    }
};
```

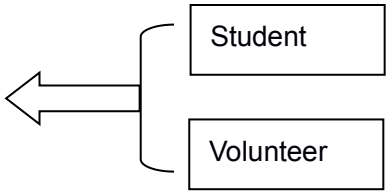
```
Class CCalculatorFactory {
    Static COperation *Create(char cOperator) {
        Switch (cOperator) {
        Case '+': return new AddOperation();
        Case '-': return new SubOperation();
        Default: return NULL;
    }
};
```

```
Int main() {
    COperation *op = CCalculatorFactory::Create('-');

    Op->m_first = 1;
    op->m_second = 2;
    Cout << op->GetResult() << endl;

    Return 0;
}
```

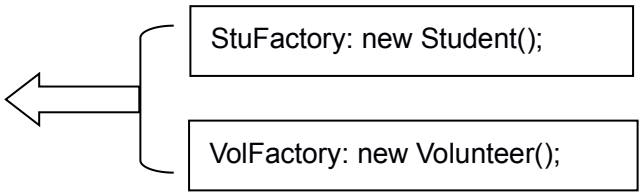
工厂方法

简单工厂的改进，工厂类不是一个，而是多个，都是从一个工厂基类派生出去，故新增产品类时**不需要修改**工厂类，**只需新增**其配套的工厂类，客户端修改 new 出来的工厂对象即可，客户端不需知道工厂类的实现。图示中的空心箭头表示继承，后同。

```
Class LeiFeng {
    Virtual void Sweep() { }
};
```



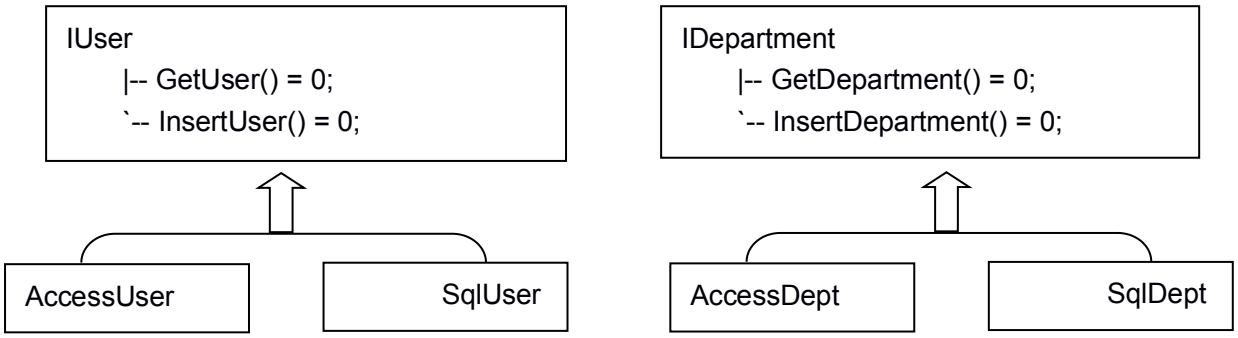

```
Class LeiFengFactory {
    Virtual LenFeng* CreateLeiFeng() {
        Return new LeiFeng();
    }
```

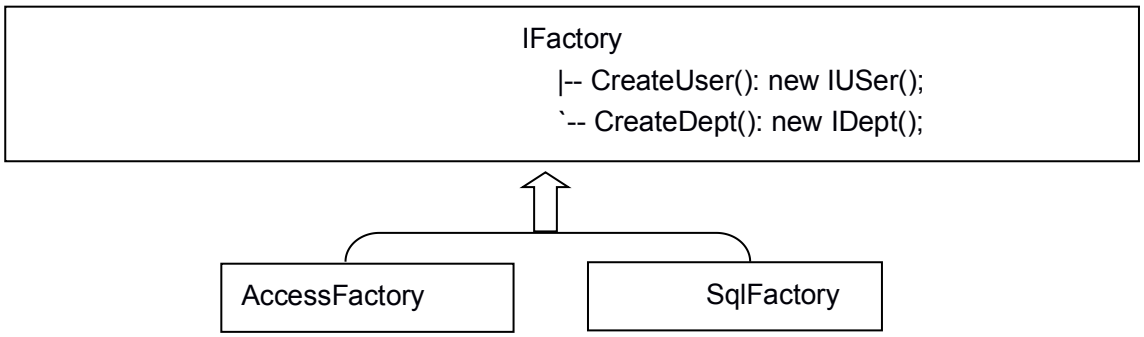



```
Int main() {
    LeiFengFactory *factory = new LeiFengFactory(); //  需求有变时改此行即可
    LeiFeng *leifeng = factory->CreateLeiFeng();
    leifeng->Sweep();
    Return 0;
}
```

抽象工厂

工厂方法的增强版：一个工厂对应一个产品，即为工厂方法；一个工厂多个产品，即为抽象工厂。这是一种格式一致的批处理，每个产品类从对应一个工厂类，改为对应唯一工厂类的其中一个方法。

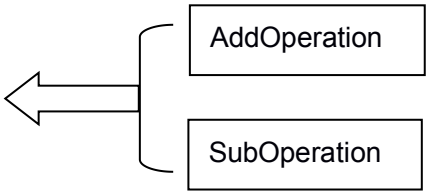




策略模式

与简单工厂相似，但不直接创建及返回对象，而是引入一个间接层（Context 类），通过该层（根据输入条件）去初始化及调用相应的工具类。**与简单工厂不同的是，策略模式的输入条件是在客户端确定**，所以客户端不需要知道策略类及产品类的实现，有新需求时只需增加新的产品类，而不需要修改现有的类。

```
Class COperation {
    Virtual int GetResult() {
        Return 0;
    }
};
```




```
Class Context {
Private:
    COperation *op;
Public:
    Context(COperation *op) {
        This->op = op;
    }
    Int GetResult() {
        Return op->GetResult();
    }
};
```

```
In main() {
    Int a, b;           cin >> a >> b;
    Char c;             cin >> c;
    Context *ctx = ('+' == c) ? (new Context(new AddOperation(a, b)))
        : (new Context(new SubOperation(a, b)));
    Cout << ctx->GetResult() << endl;
    Return 0;
}
```

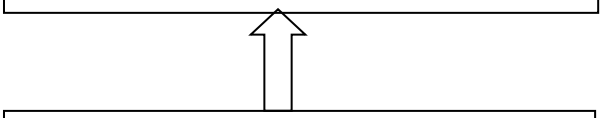
02、装饰模式、外观模式、代理模式

分别是添加不常用的额外职责、提供统一操作接口或界面、对不便被直接调用的对象进行代理的区别，详见各个模式的分析。

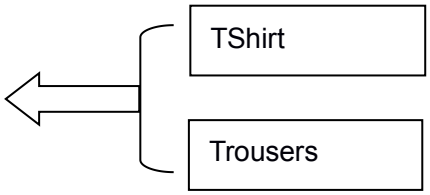
装饰模式

动态地给一个对象添加一些额外的职责（指不重要、偶尔执行的功能）。**存疑：与装扮模式/门面模式有何联系？**

```
Class Person {
    Person(const char *name){}
    Virtual void Show() {}
    // Other primary functionalities ...
};
```



```
Class Finery {
Private:
    Person *m_component;
Public:
    Void Decorate(Person *comp) {
        M_component = comp;
    }
    Virtual void Show() {
        M_component->Show();
    }
};
```




```
Int main() {

    Person *p = new Person(“小李”);

    Trousers *tr = new Trousers();
    TShirt *ts = new TShirt();
    Tr->Decorate(p);
    Ts->Decorate(tr); // ??

    Ts->Show(); //  输出什么？

    Return 0;
}
```

外观模式

为子系统的一组接口提供一个一致的界面，化繁为简，使之更易用。**存疑：是否门面模式，或与之有何区别？**

```
Class SubOne {
    Void MethodOne() {
        Cout << “One” << endl;
    }
};

Class SubTwo {
    Void MethodTwo() {
        Cout << “Two” << endl;
    }
};

Class SubThree {
    Void MethodThree() {
        Cout << “Three” << endl;
    }
};
```

```
Class Facade {
Private:
    SubOne *s1;
    SubTwo *s2;
    SubThree *s3;
Public:
    Facade() {
        New SubOne();
        New SubTwo();
        New SubThree();
    }

    FacadeMethod() {
        S1->MethodOne();
        S2->MethodTwo();
        S3->MethodThree();
    }
};
```

```
Int main() {
    Facade *test = new Facade();

    Test->FacadeMethod();

    Return 0;
}
```

代理模式

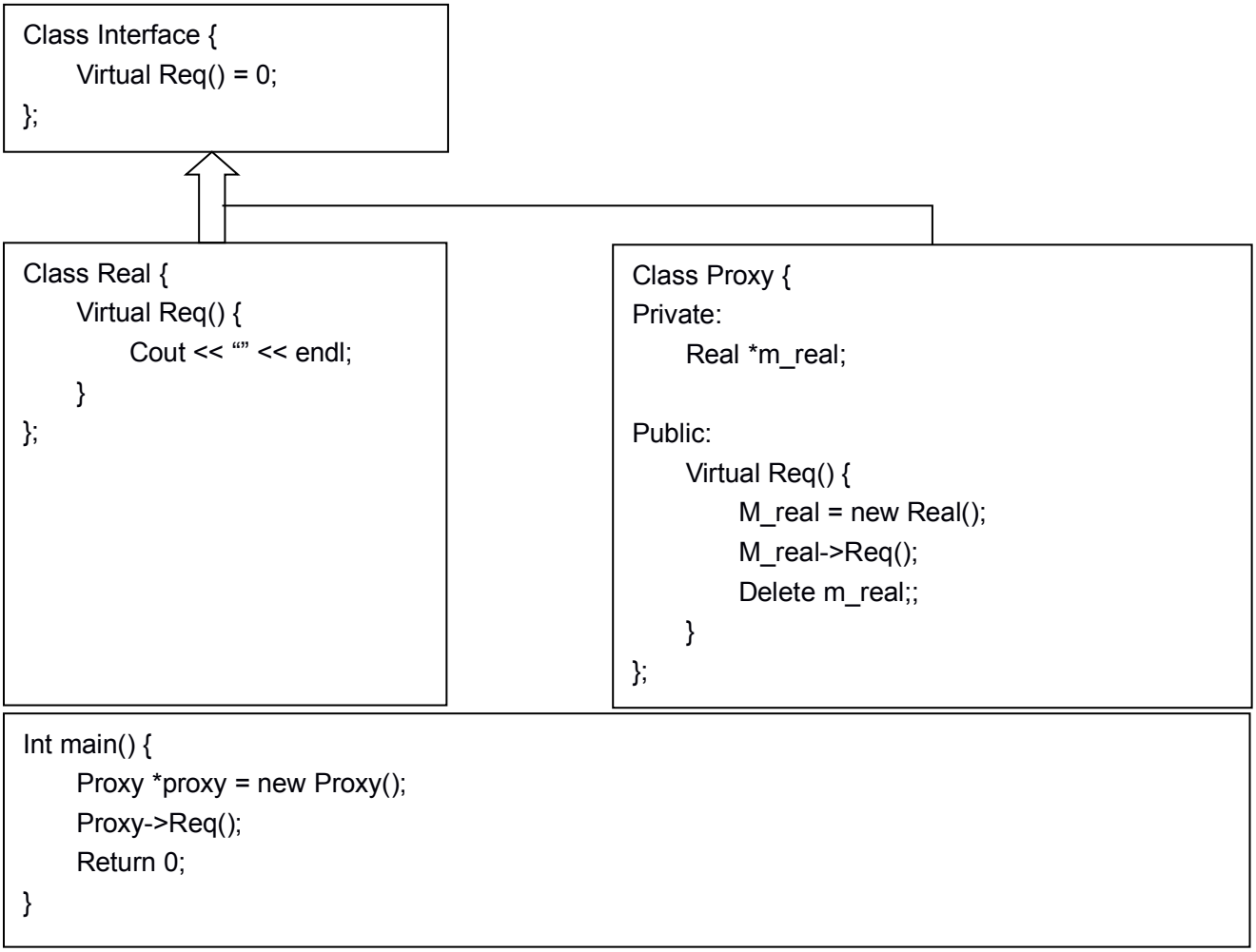
出于某些考虑，某些操作不能在本地进行，或不能直接调用对象及其方法，会采用代理模式。在以下代理场景：

远程代理：可以隐藏一个对象在不同地址空间的事实（例如远程过程调用 RPC？）。

虚拟代理：需要存放实例化很长时间的对象。

安全代理：控制真实对象的访问权限。

智能代理：当调用真实对象时，代为处理另外一些事（例如管理内存）。



03、原型模式

即复制技术，每个新的工具类均要实现 Clone()接口（或约定的其它名称），自定义如何复制自身，以供客户端调用。示例略。

04、模板方法模式

即公共代码（基本功能、必需功能……）放在父类，可变代码（个性化功能）放到子类虚函数，通过多态技术来动态调用。示例略。

05、迭代器模式

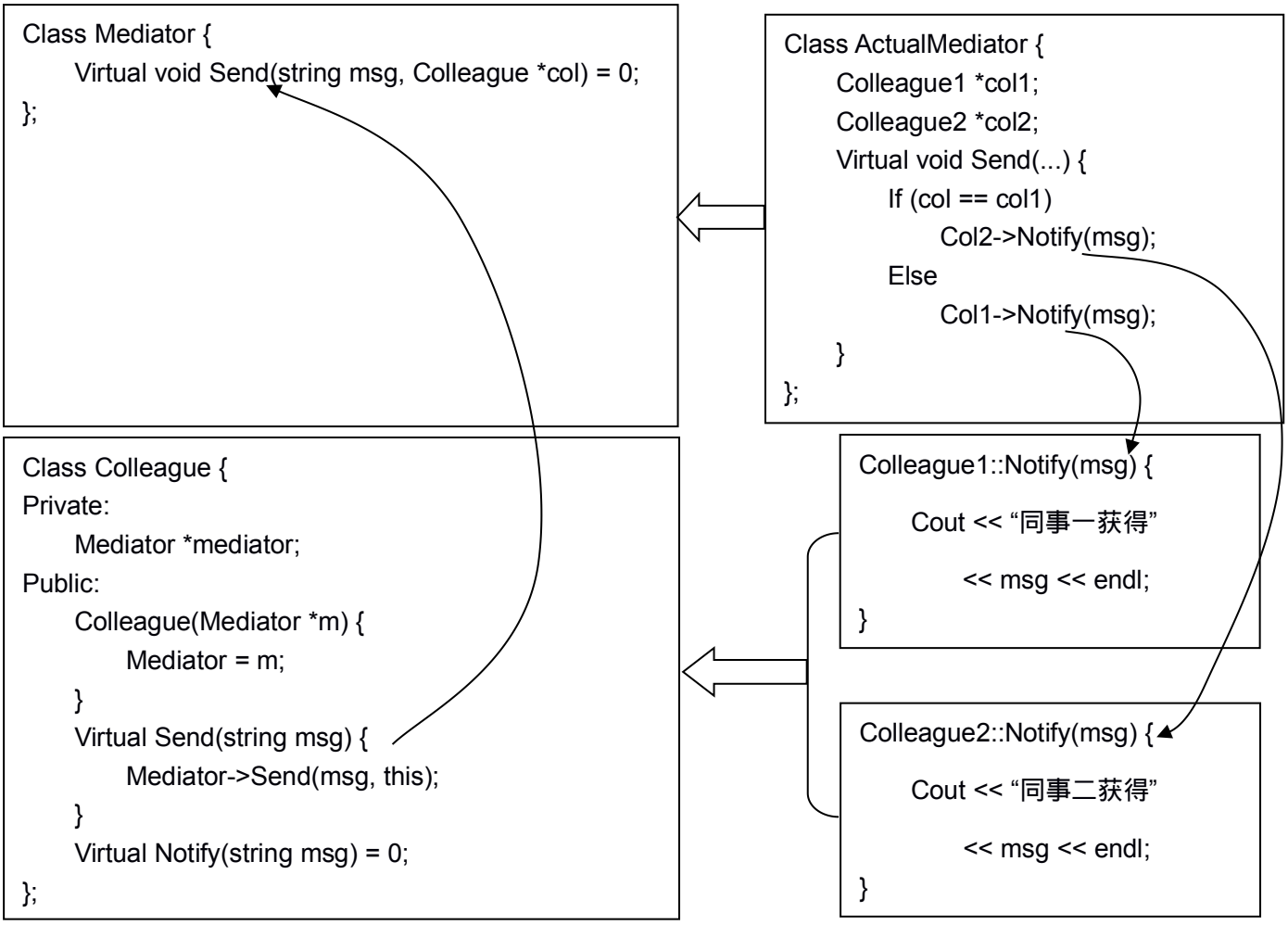
提供一种方法顺序访问一个聚敛对象的各个元素，而又不暴露其内部表示。示例略。

06、中介者模式、适配器模式、桥接模式

分别是主动解耦各模块、被逼为不能修改的第三方模块协同工作而增加间接层、解决继承不能满足“开放-封闭”原则的场合的区别，详见各模式分析。

中介者模式

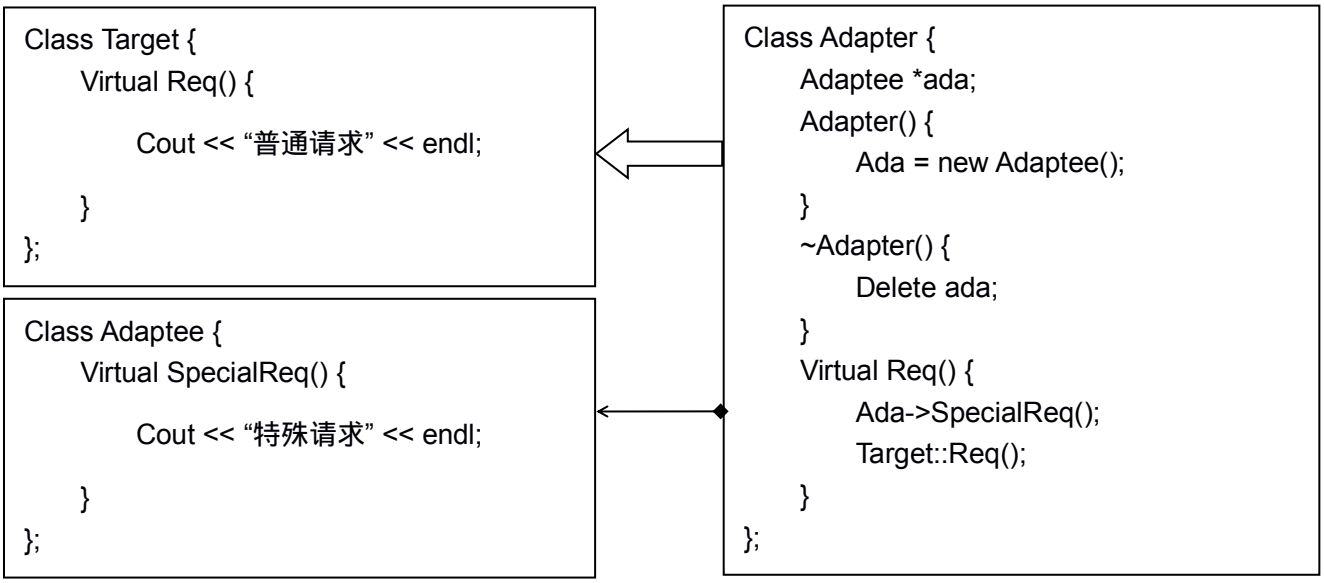
用一个中介对象来封装多个对象之间的交互，使这些对象不必有直接的耦合。




```
Int main() {
    Mediator *m = new ActualMediator();
    Colleague1 *col1 = new Colleague1(m); // introduces the mediator to colleague 1
    Colleague2 *col2 = new Colleague2(m); // introduces the mediator to colleague 2
    M->col1 = col1; m->col2 = col2; // introduces colleagues to the mediator

    Col1->Send(“吃饭了吗？”); Col2->Send(“没，你请吗？”);

    Return 0;
}
```

适配器模式

客户端和底层（或两个第三方库）都不适合修改时，可以引入适配器来协同它们的交互。



```
Class Target {
    Virtual Req() {
        Cout << “普通请求” << endl;
    }
};
```

```
Class Adaptee {
    Virtual SpecialReq() {
        Cout << “特殊请求” << endl;
    }
};
```

```
Class Adapter {
    Adaptee *ada;
    Adapter() {
        Ada = new Adaptee();
    }
    ~Adapter() {
        Delete ada;
    }
    Virtual Req() {
        Ada->SpecialReq();
        Target::Req();
    }
};
```

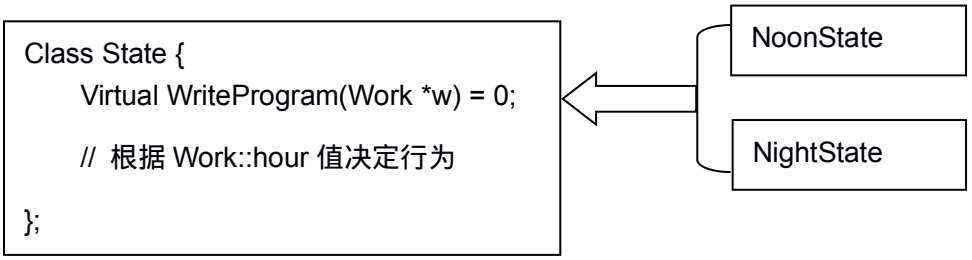
```
Int main() {
    Adapter *adr = new Adapter();
    Adr->Req();
    Return 0;
}
```

桥接模式

若使用继承不能满足“开放-封闭原则”，应考虑该模式，将抽象部分与实现部分分享，使它们可独立变化。示例待补充。

07、状态模式

当对象的行为取决于其状态，且必须在运行时根据状态来改变其行为时，用该模式。



```
Class State {
    Virtual WriteProgram(Work *w) = 0;
    //  根据 Work::hour 值决定行为
};
```

NoonState

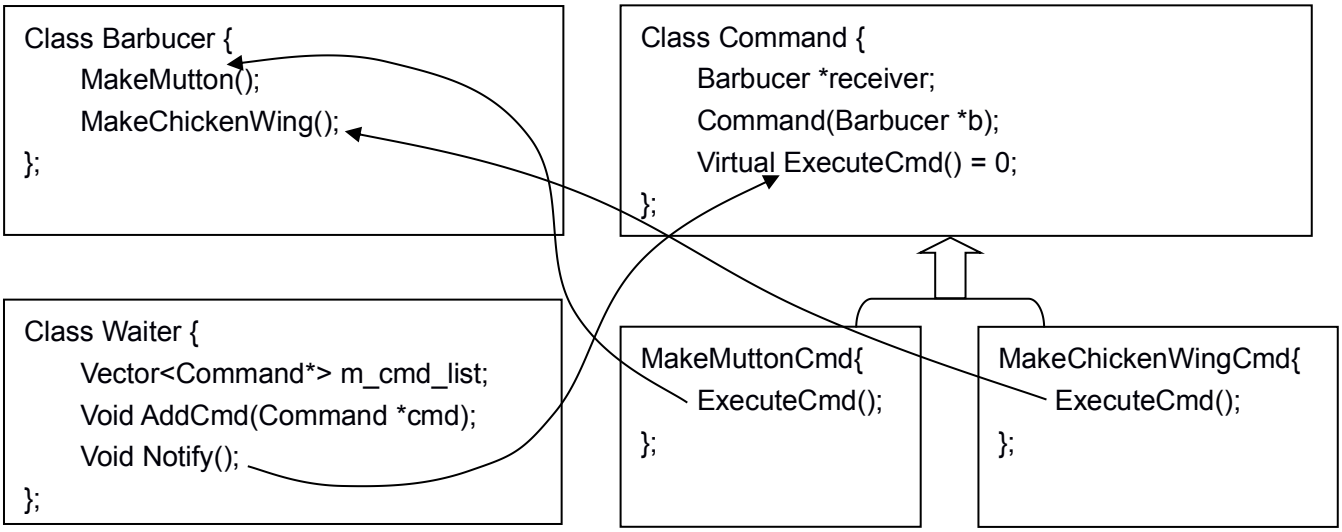
NightState

```
Class Work {
    State *current;
    Int hour;
    Void SetState(State *cur) {
        Current = cur;
    }
    Virtual WriteProgram() {
        Current->WriteProgram(this);
    }
};
```

```
Int main() {
    Work *work = new Work();
    Work->hour = 9;
    Work->WriteProgram();
    Work->hour = 14;
    Work->WriteProgram();
    Return 0;
}
```

08、命令模式

将请求操作的对象和执行操作的对象分开。使之：（1）可建立命令队列（2）可将命令记入日志（3）接收命令的一方可以拒绝（4）添加一个新命令类不影响现有类。



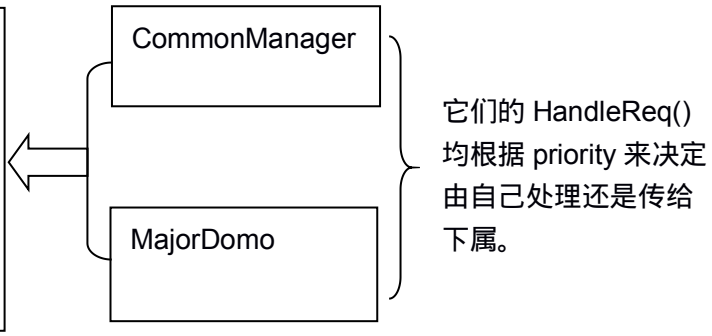
```
Int main() {
    Barbucer *bar = new Barbucer();
    Command *cmd1 = MakeMuttonCmd(bar);
    Command *cmd2 = MakeChickenWingCmd(bar);
    Waiter *waiter = new Waiter();

    Waiter->AddCmd(cmd1);
    Waiter->AddCmd(cmd2);
    Waiter->Notify();

    Return 0;
}
```

09、责任链模式

请求会沿着一条链寻找合适的处理者，处理且仅处理一次（或报错），类似不同级别的管理者执行不同权限的操作，且管理者之间存在关联/归属关系。

```
Class Req {
    String contents;
    Int priority;
};
```

```
Class Manager {
    Manager *successor;
    String name;
    Manager(string name);
    Void SetSuccessor(Manager *successor);
    Virtual void HandleReq(Req *req) = 0;
};
```



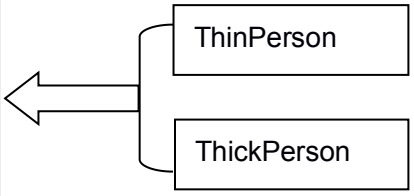

```
Int main() {

    Manager *common = new CommonManager(“张经理”);

    Manager *major = new MajorDomo(“李总监”);

    Major->SetSuccessor(common);
    Req *req = new Req();
    Req->priority = 33;          major->HandleReq(req);
    Req->priority = 3;           major->HandleReq(req);
    Return 0;
}
```

10、建造者（Builder）/生成器（Generator）模式

当创建复杂对象的算法应该独立于该对象的组成及其装配方式时适用。

与工厂方法区别：工厂方法用于创建简单对象，对象本身可创建自己的完整内容。建造者模式则用于复杂对象，对象只提供各个基础组件的创建方法，完整创建过程则由建造者来装配，类似于芯片设计公司提供设计方案并授权代工厂进行生产。

```
Class Person {
    Virtual CreateHead() = 0;
    Virtual CreateHands() = 0;
    Virtual CreateBody() = 0;
    Virtual CreateFeet() = 0;
};
```




```
Class Director {
    Person *m_person;
    Director(Person *p);
    Void Create() {
        M_person->CreateHead();
        M_person->CreateHands();
        M_person->CreateBody();
        M_person->CreateFeet()
};
```

```
Int main() {
    Person *p = new ThickPerson();
    Director *d = new Director(p);
    D->Create();
    Delete d;
    Delete p;
    Return 0;
}
```

11、观察者模式

一个通知者，多个观察者。核心是 Notifier::NotifyAll()和 Observer::UpdateSelf()。

```
Class StockObserver {
    String m_name;
    Secretary *m_notifier;
    StockObserver(string name, Secretary *notifier);
    Void UpdateSelf();
};
```

```
Class Secretary {
    Vector<StockObserver*> m_observers;
    Public: String action;
    Void Add(StockObserver *ob);
    Void NotifyAll() {
        For_each(ob : m_observers)
            Ob->UpdateSelf();
};
```

```
Int main() {
    Secretary *se = new Secretary();
    StockObserver *ob1 = new StockObserver("小李", se);

    StockObserver *ob2 = new StockObserver("小王", se);
    Se->Add(ob1);
    Se->Add(ob2);
    Se->action = "老板来了";
    Se->NotifyAll();
    Return 0;
}
```

12、备忘录模式

不破坏封装的前提下，保存及恢复一个对象的状态。

```
Class Originator {
    String state;
    Memo *CreateMemo();
    Void SetMemo(Memo *memo);
};
```

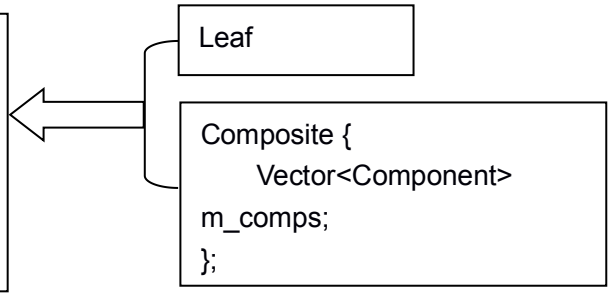
```
Class Memo {
    String state;
    Memo(string state);
};
```

```
Int main() {
    Originator *orig = new Originator();
    Orig->state = "on";
    Memo *memo = orig->CreateMemo();    // saves status
    Orig->state = "off";
    ...
    Orig->SetMemo(memo);                // restores status

    Return 0;
}
```

13、组合模式

整体和部分可一致对待，例如 XML 树结构。

```
Class Component {
        String m_name;
        Component(string name);
        Virtual Add(Component *comp) = 0;
        Virtual Display(int depth) = 0;
};
```




```
Int main() {
        Composite *p = new Composite(“小王”);

        P->Add(new Leaf(“小李”));

        P->Add(new Leaf(“小赵”));

        Composite *p1 = new Composite(“小小王”);

        P1->Add(new Leaf(“大三”));

        P->Add(p1);
        P->Display(1);
        Return 0;
}
```

14、单例模式

对象在全局范围内只有一份，访问它只能通过一个全局访问点，不能随意构造该对象。该模式常用于代替全局变量。示例略。

15、享元模式

待补充……

16、解释器模式

待补充……

17、访问者模式

待补充……

18、几大原则、法则

“开放-封闭”原则

对扩展（新增）开放，对修改封闭。原因在于修改已有代码会有影响现有功能的风险。

里氏代换原则

在软件里，把父类替换成子类，程序的行为无变化，即用于父类的地方，一定适合子类。一般用于多态。

依赖倒转原则

抽象不应该依赖细节，而是细节依赖抽象。即针对接口编程，而不要对实现编程。

高层模块不能依赖低层模块，两者都应依赖抽象。

迪米特法则

类之间无通信则不接触，要接触可通过中介。

19、参考资料

《大话设计模式实现（C++版）》