目录

六大设计准则

1. 单一职责原则：

接口： 一个接口只实现一个功能
类： 尽可能做到只有一个原因可以引起类的改变

2. 里氏替换原则

1. 子类必须实现父类的 virtual 函数

2. 子类可以有自己的个性
3. 覆盖或实现父类的方法时，形参可以被放大

```
class Father{
    public void doSomething(HashMap map){
        System.out.println("father doSomething");
    }
}
class Son extends Father{
    //此方法不是重写，而是重载
    public void doSomething(Map map){
        System.out.println("son doSomething");
    }
}
```

4. 覆盖或实现父类的方法时，返回值可以被缩小

3. 接口隔离原则

客户端不应该依赖它不需要的接口；
一个类对另一个类的依赖，应该建立在最小的接口上

4. 迪米特法则：最少知道原则

一个类应该尽可能少的知道另一个类的细节

5. 开闭原则

定义：对扩展开放，对修改封闭
简单的说：软件实体应该在尽量不要修改原有代码的基础上，进行扩展
实现技巧：动态绑定，运行时的多态

6. 依赖倒置原则

1. 抽象不依赖于细节，细节应当依赖于抽象
2. 高层（业务逻辑）不依赖于底层（业务逻辑具体实现），二者都依赖于抽象

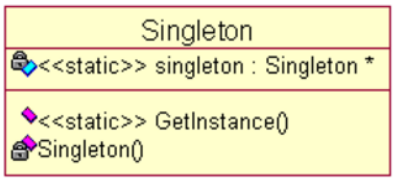
01. 单例模式

特点：
（1）确保类只有一个实例：private static Singleton* m_singleton;
（2）提供能够对该实例访问的全局接口: public static Singleton* GetInstance()
1.构造函数私有化
2.static 的单例指针：静态变量必须在类外初始化
3.static 函数：获取单例指针的全局的入口点接口
应用场景：
每台计算机可以有若干个**打印机**，但只能有一个 Printer Spooler，以避免两个打印作业同时输出到打印机中。每台计算机可以有若干**通信端口**，系统应当集中管理这些通信端口，以避免一个通信端口同时被两个请求同时调用。

饿汉式：不管你使用不使用我，我一开始就把你 new 出来
优点：不存在线程安全的问题；
缺点：占用内存空间；使用饿汉式的程序开发少

```cpp
class Sigleton
{
private:
    Sigleton(){}
    static Sigleton* single;
public:
    static Sigleton* getInstance()
    {
        return single;
    }
};                  变量初始化
Sigleton* Sigleton::single = new Sigleton(); //创建对象
int main()
{
    Sigleton* single1 = Sigleton::getInstance();
    Sigleton* single2 = Sigleton::getInstance();
    if (single1 == single2)
        cout << "同一个对象" << endl;
}
```

懒汉式：只有在使用时，才把它 new 出来



```cpp
class Singleton
{
private:
    // 1.private的构造函数
    Singleton(){}
    Singleton(const Singleton&){}
    Singleton& operator=(const Singleton&){}
    static Singleton *m_singleton; // 2.私有的静态变量
public:
    static Singleton* getInstance() //3.对外提供一个获得instance的接口
    {
        if (m_singleton == NULL) // double-check机制
        {
            Lock();
            if (m_singleton == NULL)
            {
                m_singleton = new Singleton(); //创建实例
            }
            Unlock();
        }
        return m_singleton;
    }
};
Singleton *Singleton::m_singleton = NULL;  //static变量必须在类外部进行初始化
```

```cpp
int main()
{
    Singleton *sp1 = Singleton::getInstance();
    Singleton *sp2 = Singleton::getInstance();
    if (sp1 == sp2)
        cout << "相等" << endl;
}
```

02.简单工厂模式（静态工厂模式）

静态：用枚举变量去判断创建对象的类型

要点：只有一个工厂类；多个产品类

使用方法（两部曲）：

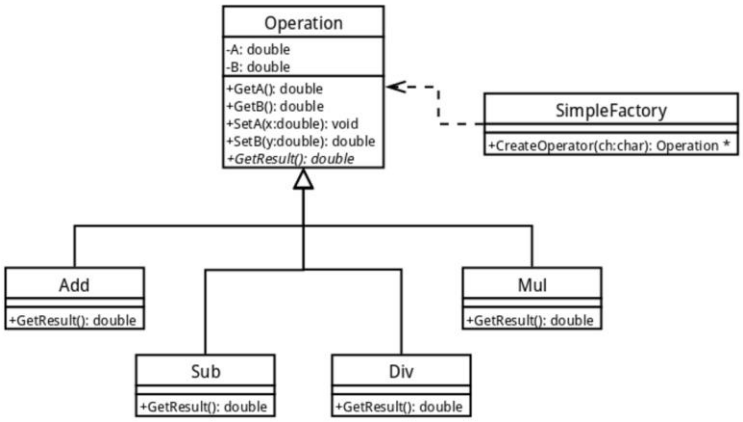
(1)先 new 一个具体工厂：Factory* factory = new Factory ();

(2)用具体工厂，采用枚举变量创建自己想要的对象：

Product* ProductA = factory->createProduct(枚举变量);

核心：对于一个父类的多个继承子类，工厂对象的工厂函数根据用户输入，自动 new 出一个子类对象并返回其父类的指针，这样利用父类的指针执行父类的虚函数，就可以动态绑定子类的重写函数，从而实现多态。

缺点：createOperator 函数中，有大量的 if...else...，可扩展性差



代码实现：

```
class Operate 运算类: + - *
{
protected:
    int a, b;
public:
    Operate():a(0),b(0){}
    void setA(int a){ this->a = a; }
    void setB(int b){ this->b = b; }
public:
    virtual int operate() = 0;虚函数
};
class Add :public Operate
{
public:
    int operate(){ return (a + b); }
};
class Sub :public Operate
{
public:
    int operate(){ return (a - b); }
};
class Mul :public Operate
{
public:
    int operate(){ return (a * b); }
};
```

```
class Factory 工厂类，用于创建Operate对象：可以根据输入c，判断创建哪个Operate子对象
{
public:
    Operate* CreateOperate(char c) //工厂模式中只有一个函数，用来生产Operate对象
    {                              输入c
        switch (c)
        {
        case '+':
            return new Add();
        case '-':
            return new Sub();
        case '*':
            return new Mul();
        }
    }
};
```

根据输入c，判断应该创建哪个Operate对象-->用new创建Operate对象，然后返回

可以看到，

简单工厂类，仅仅是为了创建已经存在的类的“对象”！

创建完对象后，可以使用该“对象”，调用基类中的virtual函数

罗克韦尔实习中使用过 simple facetory 模式：

创建 IPCBace 对象时，使用 switch…case…判断输入字符串的是 CriticalSection，Mutext，Semphore，还是 Event，进而通过 Factory 的 IPCBase* CreateIPCBase(string str)创建以上对象。

用户需求在扩增：假如此时需要在原有的+ - *运算的基础上，添加/运算

那么如果使用简单工厂方法，需要在 swtich…case…中加入 case ‘/‘ →→→这样破坏了“封闭原则”，程序的改动非常大→→→因此简单工厂模式在实际的应用中比较少，缺点很明显→→→引出工厂方法

03.工厂方法 Factory Meythod（动态工厂模式）

动态：用具体工厂对象，调用虚函数，创建对应的具体产品

要点：多个个工厂类；多个产品类

使用工厂方法创建对象（两部曲）：

先 new 一个具体工厂：Factory* factory = new ConcreteFactory();

用具体工厂创建具体产品对象：Product* product = factory->createProduct();

产品对象扩展（两部曲）：

在 AbstractFactory 类下增加具体工厂类；

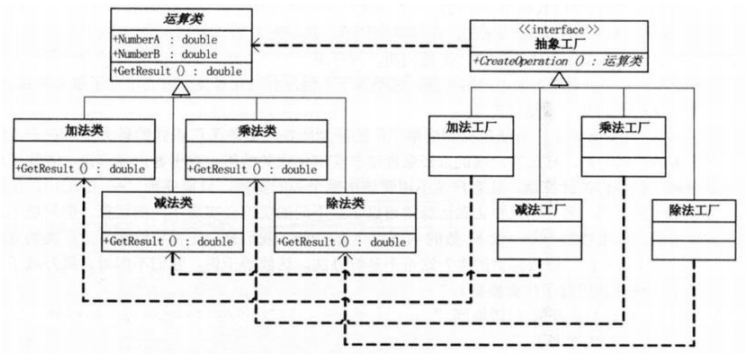
在 AbstractProduct 类下增加具体产品类

在简单工厂的基础上，进行下面修改

（1） 将简单工厂中的工厂抽象出来

（2） 抽象工厂类，每一个具体工厂都对应生产一个具体产品

好处：对于新增的产品，只需要在抽象工厂类下，创建新的工厂子类→→不破坏“开闭原则”



代码实现：

```
class Operate
{
protected:
    int a, b;
public:
    Operate():a(0),b(0){}
    void setA(int a){ this->a = a; }
    void setB(int b){ this->b = b; }
public:
    virtual int operate() = 0;
};
class Add :public Operate
{
public:
    int operate(){ return (a + b); }
};
class Sub :public Operate
{
public:
    int operate(){ return (a - b); }
};
class Mul :public Operate
{
public:
    int operate(){ return (a * b); }
};
```

```
class Factory
{
public:
    virtual Operate* CreateOperate() = 0;
};
class FactoryAdd :public Factory
{
public:
    Operate* CreateOperate()
    {
        return new Add();
    }
};
class FactorySub :public Factory
{
public:
    Operate* CreateOperate()
    {
        return new Sub();
    }
};
class FactoryMul :public Factory
{
public:
    Operate* CreateOperate()
    {
        return new Mul();
    }
};
```

```
int main()
{
    Factory* m_factory = new FactoryAdd();
    Operate* m_operate = m_factory->CreateOperate();

    m_operate->setA(10);
    m_operate->setB(10);
    int res = m_operate->operate();
}
```

04.抽象工厂

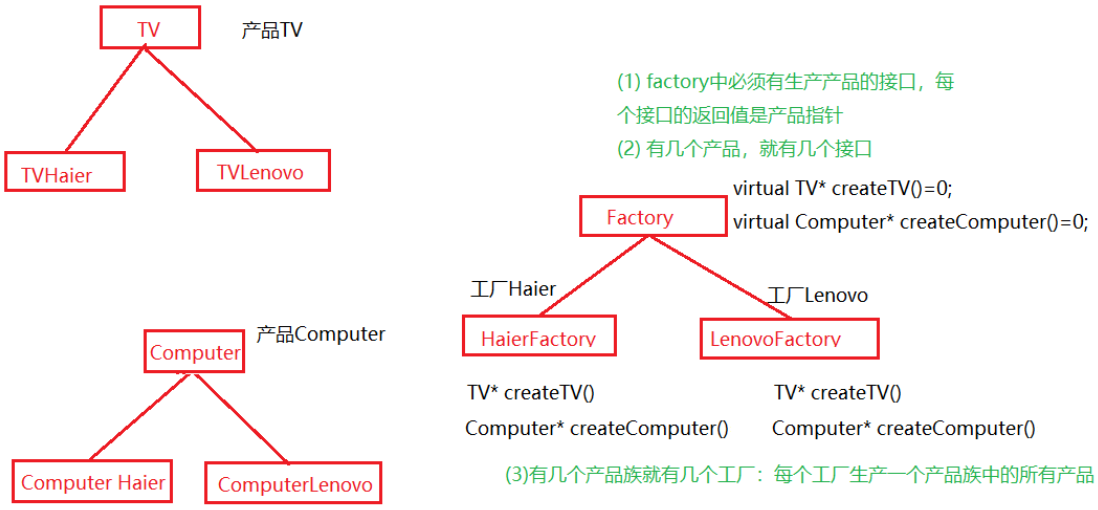
要点： 工厂方法只能生产一个产品 →→ 抽象工厂可以生成一个产品族（因此抽象工厂中有多个 createProduct 的接口）

举例：现在有两种产品 TV 和 Computer；有两个公司 Haier 和 Lenovo 都生产 TV 和 Computer；

分析：

(1)有 2 种产品：TV 和 Computer → 因此 Factory 类中有两个接口

(2)有两个产品族：Haier 和 Lenovo → 因此有 Factory 有两个子类，即有两个工厂



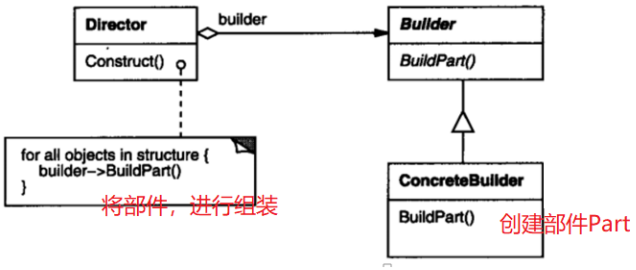
05.Builder 模式

将一个复杂对象的**构建（Director 的设计蓝图）**和**表示（Builder 的 BuildPart）**分离→Direct 设计蓝图 BuildProduct() ，由 Builder 去创建部分 BuildPart()

举例：建房子

1. 设计师 Director 设计如何创建房子：先创建墙壁，再创建门，最后创建窗户
2. 建造者 Builder 按照 Director 的设计蓝图，去实现创建墙壁，创建门，创建窗户的函数

设计好处：将 Builder 类抽象成虚基类，多个子类去继承 Builder 类，都重写 Builder 的虚函数 BuildPart()接口→→可以有各种各样的 Builder 对象，使得创建出来的产品多种多样。



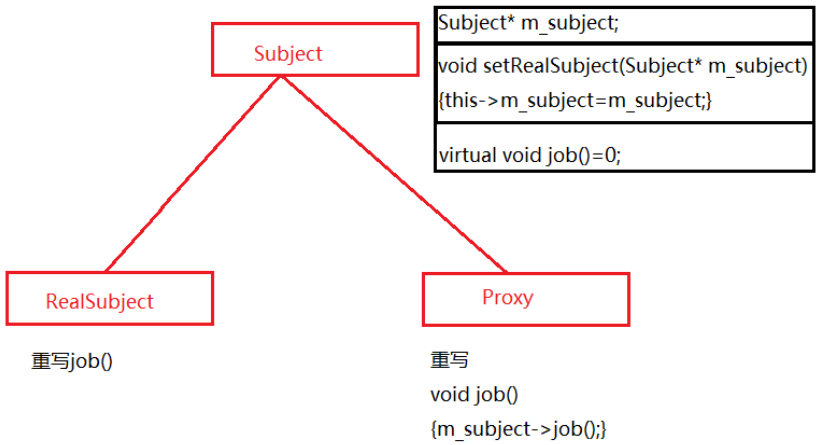
代码实现：

```cpp
class Builder                    虚类
{
public:
    virtual void buildHead(){}
    virtual void buildLeg(){}
};
class BuilderPerson :public Builder
{                实现类
public:                                          创建部件
    void buildHead(){ cout << "创建Head" << endl; }
    void buildLeg(){ cout << "创建Leg" << endl; }
};

class Direct  指挥着
{
public:
    Builder* m_builder;  存在Builder*成员
public:
    Direct(Builder* m_builder)
    {
        this->m_builder = m_builder;
    }
    void CreatePerson()
    {                                  进行组装
        m_builder->buildHead();
        m_builder->buildLeg();
    }
};

int main()
{
    Direct* p_direct = new Direct(new BuilderPerson());
    p_direct->CreatePerson();
}
```

```
C:\Users\GuoJawee\Desktop\MemoryMana
创建Head
创建Leg
```

06.代理模式

A 中包含 B 类：AB 类实现协议类



代理无处不在：游戏代刷等级

代理模式非常的简单，举个通俗易懂的例子：假设大力 RealSubject 要送花给探探，但是由于自己没有时间，因此大力 RealSubject 将送花的任务委托给东东 Proxy 去做。这就是一个很简单的代理模式

分析：

在代理模式中，有两个角色——任务委托者（大力），任务执行者（东东），东东是大力的代理。执行送花动作的人虽然是东东，但是实际上 RealSubject 实际主题确实大力。

```
class Subject
{
protected:
    Subject* m_subject;
public: //给Proxy设置RealSubject
    void setRealSubject(Subject* m_subject){ this->m_subject = m_subject; }
public:
    virtual void job() = 0; 代理的任务
};
class RealSubject :public Subject
{
public:      RealSubject执行的job
    void job(){ cout << "大力送花给探探" << endl; }
};
class Proxy :public Subject
{
public:          代理帮RealSubject完成job
    void job(){ m_subject->job(); }
};
```

07.装饰模式

核心：接口（函数）包装

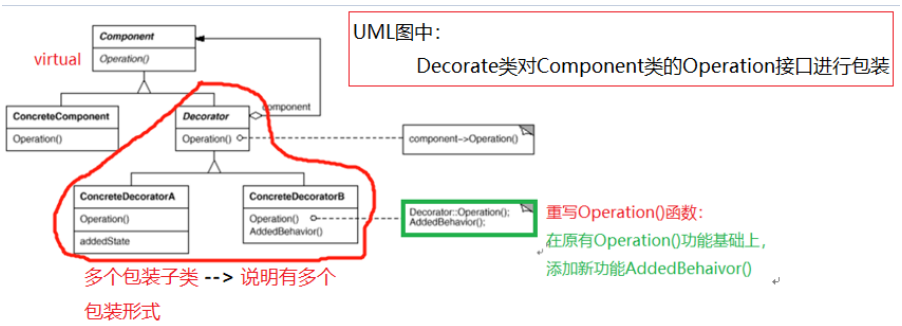
使用装饰者模式，可以动态的给**一个对象（而不是给整个类）**添加一些额外的职责。

实现注意点：接口一致性

装饰对象的接口必须与**它所装饰的类的接口**是一致的。体现了“装饰”过程不能改变对象的“本质”。→ 因此只是对原有的接口函数进行功能的扩展

举例 1：茶→柠檬茶→苹果柠檬茶，只是在茶的口味上进行包装，本质上是对口味这个接口进行功能扩展。

举例 2：车可以跑→添加：车可以飞→添加：车可以游泳



```
class Tea 被装饰的类
{
public:
    virtual void getDescription(){ cout << "Tea " << endl; }
};
class CoffeTea :public Tea
{
public:
    void getDescription(){ cout << "CoffeTea " << endl; }
};
class MochaTea :public Tea
{
public:
    void getDescription(){ cout << "MochaTea " << endl; }
};

int main()
{
    Tea* tea = new CoffeTea(); 创建被包装类对象tea
    Tea* lemon = new DecorateLemon(tea);创建包装类lemon，对对象tea进行包装
    lemon->getDescription();
}
```

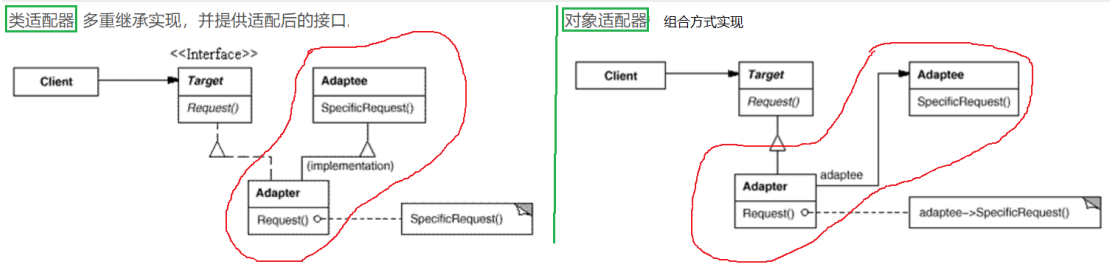
```
C:\Users\GuoJawe
Lemon CoffeTea
```

```
装饰类：对Tea类的接口getDescription()进行功能扩展
class Decorate:public Tea
{
规律：如果一个类Decorate继承和组合同一个类Tea，那么这个
    Tea* m_tea;
类Decorate十有八九是类Tea的包装类！
public:
    Decorate(Tea* m_tea) :m_tea(m_tea){}
    virtual void addMateria() = 0;
};
class DecorateLemon :public Decorate
{
public:
    DecorateLemon(Tea* m_tea) :Decorate(m_tea){}
    void addMateria(){ cout << "Lemon "; }
    void getDescription()
    {
        addMateria();
        m_tea->getDescription();
    }
};
class DecorateApple :public Decorate
{
public:
    DecorateApple(Tea* m_tea) :Decorate(m_tea){}
    void addMateria(){ cout << "Apple " << endl; }
    void getDescription()
    {
        addMateria();
        m_tea->getDescription();
    }
};
```

08.适配器模式

说明：用的最多的结构型设计模式

对已经存在的类，进行接口的改造，衍生出一个新类→→新类所有的接口，都是通过调用已经存在的类的接口实现的，例如：对 deque 进行适配，产生 stack 和 queue。



应用场景：

STL 源码剖析中，有三种适配器：容器适配器，迭代器适配器，仿函数适配器

废话不多说，直接看 STL 源码部分 stack：

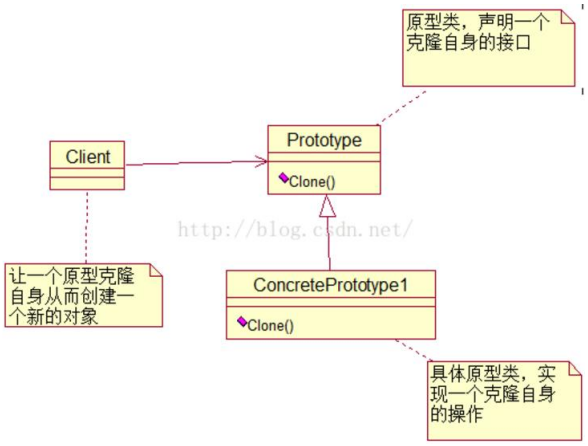
```
class stack 适配后产生的新类stack
{
protected:    被适配的类deque
    deque c;     // the underlying container
public:
    bool empty() const
    {   // test if stack is empty
        return (c.empty()); 用c调用旧接口，进行改造生成新类stack可以使用的接口
    }

    void push(const value_type& _Val)    将旧接口进行改造
    {   // insert element at end
        c.push_back(_Val);
    }

    void pop()    将旧接口进行改造
    {   // erase last element
        c.pop_back();
    }
    ...
        ...
};
```

09.原型模式 *

原型模式，本质上就是实现一个 Clone 函数，使得能用一个原型对象克隆出另一个对象。

深拷贝问题：Clone 接口：另外开辟一个空间，将原型对象中的内容拷贝过去

举例：简历复制



```cpp
class Resume
{
private:
    string name;
    int age;
public:
    Resume(string name, int age) :name(name), age(age){}
public:
    Resume* Clone()
    {
        return new Resume(*this);  //浅拷贝
    }
};
int main()
{
    Resume* gjw = new Resume("gjw", 24);
    Resume* wl = gjw->Clone();
}
```

10.桥接模式

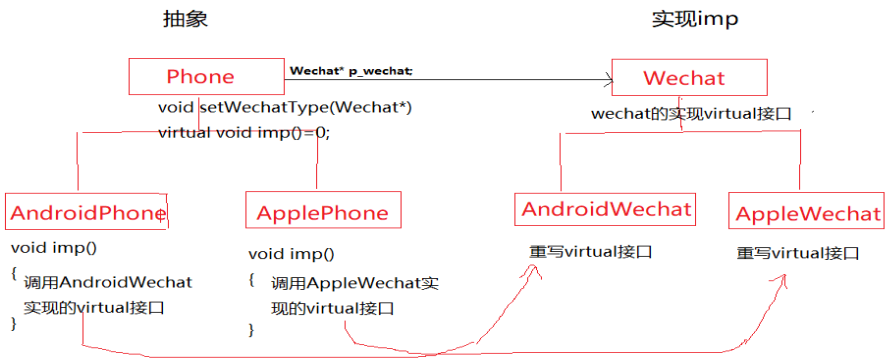
将**抽象部分**和**它的实现部分**相分离，使得它们能独立的变化。

将类中的实现，抽象出来，在另一个类（imp）中实现

举例：Apple 手机平台和 Android 手机平台；二者都实现 wechat 程序，其中 wechat 程序对应也有两个版本，AppleWechat 和 AndroidWechat→要实现 Apple 手机运行 AppleWechat 程序，Android 手机运行 AndroidWechat 程序→

方案一（NO）：采用继承，会造成类的膨胀

方案二（Yes）：可以将运行 Wechat 程序抽象成接口；供手机 Phone 类调用→从而实现二者都可以独立的变化



罗克韦尔实习中使用过 Bridge 模式：

平台 Client/Server 使用不同的加锁方式 IPCBase(CriticalSection,Mutext,Semphore,Event 等)

举例 1：各种各样的车，安装各种各样的发动机

举例 2：各种各样的图像，填充各种各样的颜色

11.模板方法

本质：不变部分封装到父类，可变部分通过继承的方式在子类实现

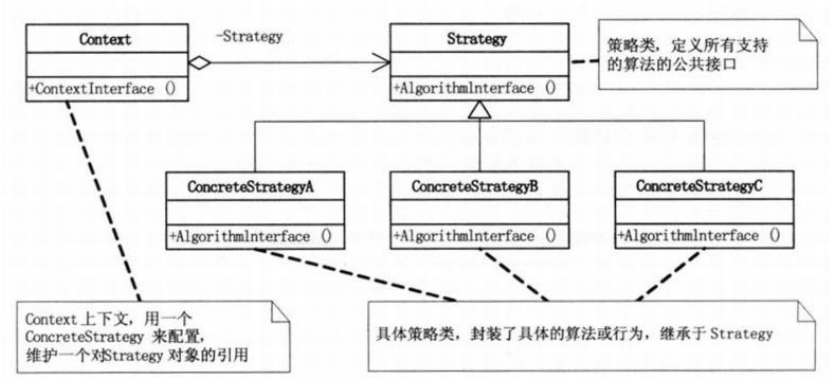
在抽象类中定义统一的操作逻辑步骤的接口；让子类去实现

```cpp
class Parents
{
public:
    void run()  //逻辑不变的地方-->封装在父类中
    {
        step1();
        step2();
    }
    virtual void step1() = 0; //逻辑改变的部分，延迟到子类中实现
    virtual void step2() = 0; //逻辑改变的部分，延迟到子类中实现
};

class SonA :public Parents
{
public:
    void step1(){ cout << "A step1" << endl; }
    void step2(){ cout << "A step2" << endl; }
};
class SonB :public Parents
{
public:
    void step1(){ cout << "B step1" << endl; }
    void step2(){ cout << "B step2" << endl; }
};
int main()
{
    Parents* sonA = new SonA();
    sonA->run();

    Parents* sonB = new SonB();
    sonB->run();
}
```

12.策略模式

使用场景：当使用一个算法的判断条件过于复杂时，有大量的 if..else…

将 Alogrithm 抽象成接口封装到一个类中 Strategy，用 Context 去配置 Stragety，从而使 Client 端只需要知道 Context，就能调用相应的 Strategy。



Head First：会飞的鸭子

举例：商场售卖商品，有的商品原价出售，有的商品打折（7 折）销售；

对于商品有很多销售策略→→因此可以将销售策略抽象成 strategy interface，在 Context 中包含 strategy*变量，client 通过 Context 绑定 strategy，进而执行不同的 strategy。

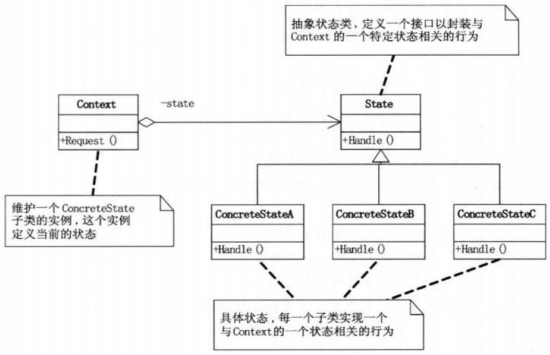
```cpp
class Strategy  商品打折策略：抽象成类
{
public:
    virtual double GetCurPrice(double price) = 0;
};
class Strategy10 :public Strategy
{
public:                                 //商品价格不打折销售
    double GetCurPrice(double price){ return price*1.0; }
};
class Strategy7 :public Strategy
{
public:                                 //商品价格打7折销售
    double GetCurPrice(double price){ return price*0.7; }
};
class Context
{
private:
    double OriginalPrice; //商品原价
    Strategy* m_strategy; //包含打折策略的指针用Context去配置Stragety
public:
    Context(double OriginalPrice){ this->OriginalPrice = OriginalPrice; }
public:
    void SetSoldStrategy(Strategy* m_strategy){ this->m_strategy = m_strategy; }//绑定销售策略
    double GetPrice(){ return m_strategy->GetCurPrice(OriginalPrice); } //获得商品打折后的价格
};
```

```cpp
int main()
{
    Context* m_context = new Context(100);

    m_context->SetSoldStrategy(new Strategy10());
    double price1 = m_context->GetPrice();  // 100.0

    m_context->SetSoldStrategy(new Strategy7());
    double price2 = m_context->GetPrice();  // 70.0
}
```

13.状态模式

使用场景：当控制一个对象状态的判断条件过于复杂时，有大量的 if..else…

State 模式很好地实现了对象的状态逻辑和动作实现的分离，状态逻辑分布在 State 的派生类中实现，而动作实现则可以放在 Context 类中实现（这也是为什么 State 派生类需要拥有一个指向 Context 的指针）。这使得两者的变化相互独立，改变 State 的状态逻辑可以很容易复用 Context 的动作，也可以在不影响 State 派生类的前提下创建 Context 的子类来更改或替换动作实现。



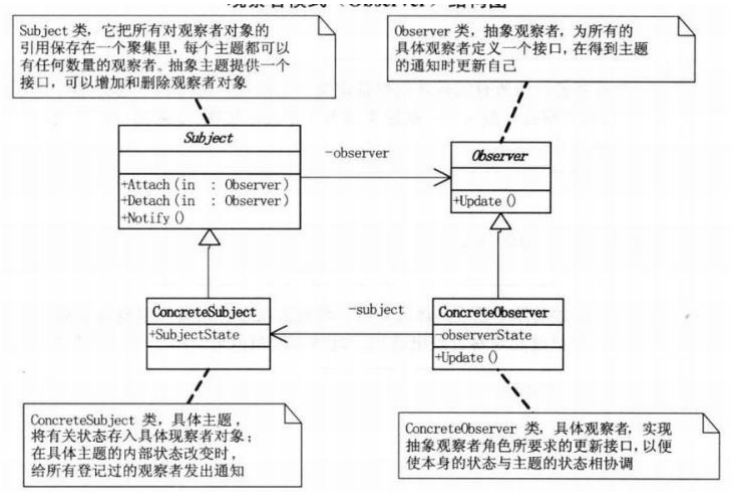
举例：一个 Worker 从早上起床到晚上下班，要经过以下时间（每个时间段都对应一个状态）：forenoonState,NoonState,EveningState,AfterworkState→→将 worker 的 state 从 worker 中抽象出来，使得 worker 与 state 解耦合。

其中 state 类中，只有一个接口 void doSth(Worker* worker)

14.Observer 模式（消息发布-订阅模式）

观察者模式：observer 订阅 subject，当 subject 发出消息 notify()时，所有的 observer 都进行 update()→→**Subject:Observer=1:N**

定义对象间一种一对多的依赖关系，使得一个对象状态改变 Notify，则所有依赖于它的对象的状态都会更新。



举例：

（1）数据库 Subject 在进行更新时，将以饼状图/柱状图/线型图/表格等 Observer 形式呈现

（2）在工作生活中，有时经常会有员工开小差，比如：小王在看 NBA，小明在打王者荣耀，小红在化妆等；此时有一个人（假设名字叫做 subject）对老板进行监控，当老板来时，subject

对小王/小明/小红发出通知 notify()，与此同时小王/小明/小红收到通知后，立刻装作“努力工作”。

```
class Observer
{
public:
    virtual void update() = 0; 观察者，只有一个接口：收到notify()后，做出反应
};
class Observer1:public Observer
{
public:
    void update(){ cout << "小王收到subject通知，停止看NBA，装作努力工作" << endl; }
};
class Observer2 :public Observer
{
public:
    void update(){ cout << "小明收到subject通知，停止打王者荣耀，装作努力工作" << endl; }
};
class Observer3 :public Observer
{
public:
    void update(){ cout << "小红收到subject通知，停止化妆，装作努力工作" << endl; }
};
```

```
class Subject   主题类:
{
public:
    void Attach(Observer* ob){ m_list.push_back(ob); }
    void Detach(Observer* ob){ m_list.remove(ob); }
    void notify() 发布消息
    {
        cout << "通知！通知！老板来了..." << endl;
        for (auto ob : m_list)
        {
            ob->update();
        }
    }
private:
    list<Observer*> m_list; 维护着观察者list
};
```

```
int main()
{
    Subject* m_subject = new Subject();

    Observer* ob1 = new Observer1();
    Observer* ob2 = new Observer2();
    Observer* ob3 = new Observer3();

    //ob1,ob2,ob3订阅subject
    m_subject->Attach(ob1);
    m_subject->Attach(ob2);
    m_subject->Attach(ob3);
    //subject发出通知，与此同时，所有订阅了subject的observer都会update自己的行为
    m_subject->notify();
}
```

15.组合模式（树形结构）*

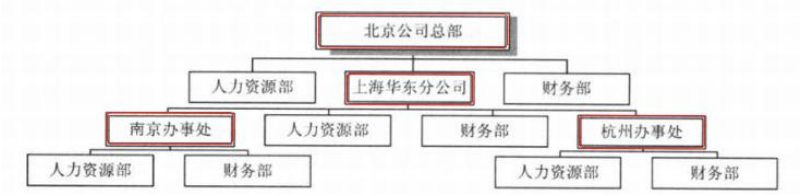
使用场景：组合模式让客户可以一致地使用组合对象和单个对象的接口

当你发现需求中是体现**部分与整体**层次的结构时，以及希望用户可以忽略组合对象与单个对象的不同，统一地使用组合结构中的所有对象时：用户不用关心到底是处理一个叶节点还是处理一个组合组件，也就不用写判断语句辨别是单一对象还是组合对象

基本对象可以被组合成更复杂的组合对象，而这个组合对象又可以被组合，这样不断地递归下去。

举例 1：打印一个目录的结构

举例 2：



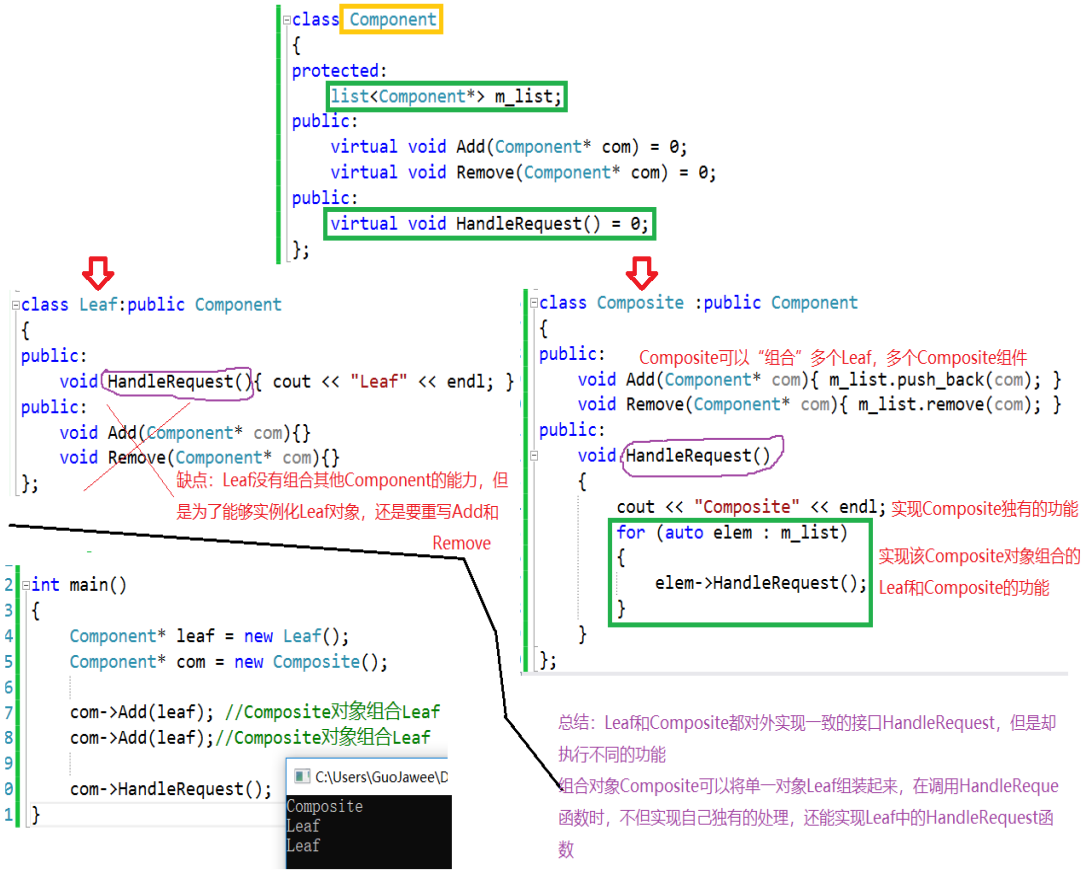
小菜已经开发出了人力资源部，财务部的办公管理系统，需求：某个大公司希望将这套办公管理系统在全公司进行推广。

此时此刻，小菜已经开发好的人力资源部办公管理系统，财务部办公管理系统

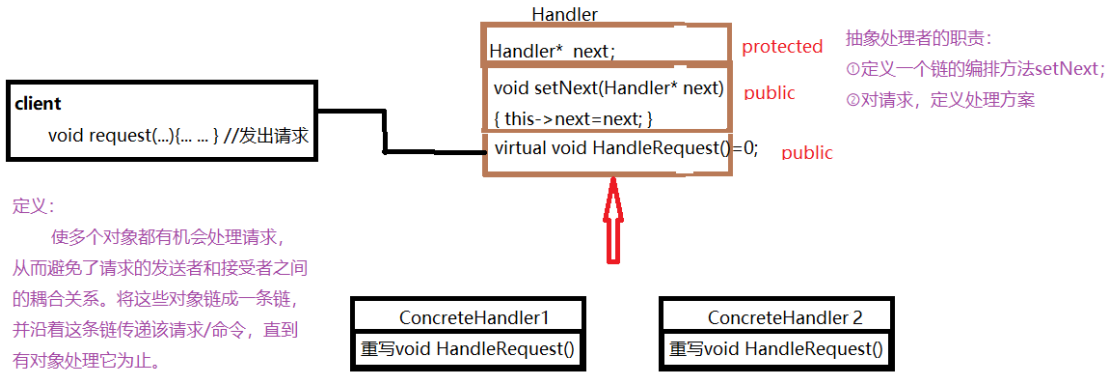
解决方案 1：对于每个分公司，都将人力资源部，财务部的管理系统都复制一份，工作量巨大。

解决方案 2：将人力资源部，财务部的管理系统当作 Leaf，将分公司当作 Composite，因此

Composite 可以通过 Add(Leaf)，实现 Leaf 功能的复用。



16.职责链模式（链表结构）



举例：小王申请“加薪水”，但是加薪水可不是那么简单的事情，操作流程那是相当的复杂：

（1）如果申请增加的薪水小于 500 元，经理可以直接决定是否批准；

（2）如果申请增加的薪水大于 500 元，经理就没有权限决定是否批准，将会将消息向主管汇报，由主管决定是否批准

这种运作模式，形成一条链表，当前节点不能处理请求，就将请求转交给 next 节点，直到请求被处理。

```cpp
class Handler
{
protected:
    Handler* next; //关键:指向基类Handler的指针，通过next设置下一个节点
public:
    void setNext(Handler* next){ this->next = next; }
    virtual void HandleRequest(int value) = 0;//处理请求
};
class Manager :public Handler
{
public:
    void HandleRequest(int value)
    {
        if (value <= 500)
            cout << "Manager: 允许加薪水" << value << "元" << endl;
        else
        {
            cout << "Manager: 我没有权限批准是否加薪水" << value << "元"<<", 请询问Director" << endl;
            next->HandleRequest(value); 将请求转交给下一级处理
        }
    }
};
class Director :public Handler
{
public:
    void HandleRequest(int value)
    {
        if (value <= 1000)
            cout << "Director: 允许加薪水" << value << "元" << endl;
        else
        {
            cout << "Director: 不允许加薪水超过1000" << endl;
        }
    }
};
```

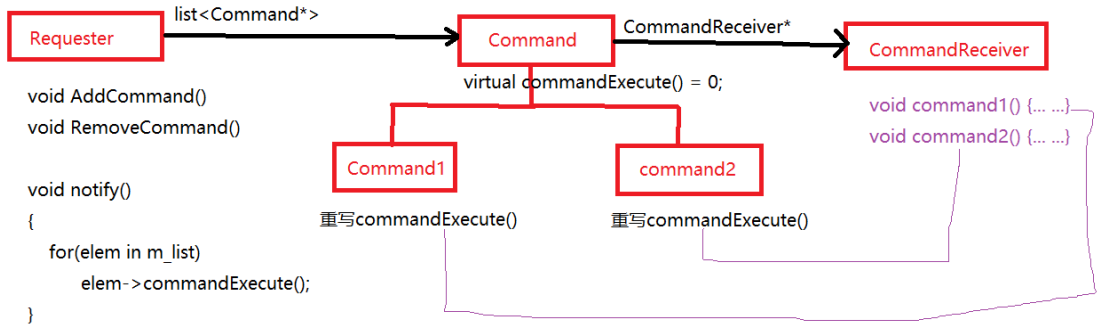
```cpp
int main()
{
    Client* c = new Client();
    int value = c->request(1100);

    Handler* m_Manager = new Manager();
    Handler* m_Director = new Director();

    m_Manager->setNext(m_Director);
    m_Director->setNext(NULL);

    m_Manager->HandleRequest(value);
}
```

```cpp
class Client
{
private:
    int value;
public:
    int request(int value){ cout << "请求加薪水: " << value << "元" << endl; return value; }
};
```

17.命令模式 Command

三个角色：命令请求者，命令，命令实现者

将命令封装成对象，使得命令请求者和命令实现者解耦合



命令请求者 Requester：包含请求的命令列表 list<Command*>，添加/删除命令操作，通知命令列表中的命令执行的操作 notify()

命令 Command：包含命令接收者 CommandReceiver；只有一个 virtual 函数命令执行接口

命令接收者 CommandReceiver：有几个命令子类，就对应有几个函数，真正的指向命令的内容

举例：客户(main 函数)找到 Waiter(命令请求者)进行点菜(命令)，Waiter 将点的餐放入菜单列表，点完菜后，Waiter 通知厨师做菜。

```cpp
class Barbecur // 命令执行者
{
public:
    void MakeMakeMutton() { cout << "烤羊肉串" << endl; }
    void MakeChickwing(){ cout << "烤鸡翅" << endl; }
};

class Command //命令
{
protected:
    Barbecur* barbecur; //命令中包含命令执行者
public:
    Command(Barbecur* barbecur) :barbecur(barbecur){}
public:
    virtual void Execute() = 0; //只有这一个虚函数
};
class MakeMuttonCommand :public Command
{
public:
    MakeMuttonCommand(Barbecur* barbecur) :Command(barbecur){}
public:
    void Execute() { barbecur->MakeMakeMutton(); }
};
class MakeChickwingCommand :public Command
{
public:
    MakeChickwingCommand(Barbecur* barbecur) :Command(barbecur){}
public:
    void Execute(){ barbecur->MakeMakeMutton(); }
};
```

```cpp
class Waiter //命令请求者
{
private:
    list<Command*> m_list; //命令列表
public:
    void AddCommand(Command* command){ m_list.push_back(command); }
    void RemoveCommand(Command* command){ m_list.remove(command); }

    void notify() // 一次通知，所有请求的命令都执行
    {
        for (auto elem : m_list)
            elem->Execute();
    }
};
```

```cpp
int main()
{
    Barbecur* barbecur = new Barbecur(); //命令执行者

    Command* makechickwingcommand = new MakeChickwingCommand(barbecur);//可选命令
    Command* makemuttoncommand = new MakeMuttonCommand(barbecur);//可选命令

    Waiter* girl = new Waiter();//命令请求者
    girl->AddCommand(makemuttoncommand);
    girl->AddCommand(makechickwingcommand);
    girl->notify(); //一次通知，执行所有的命令

}
```

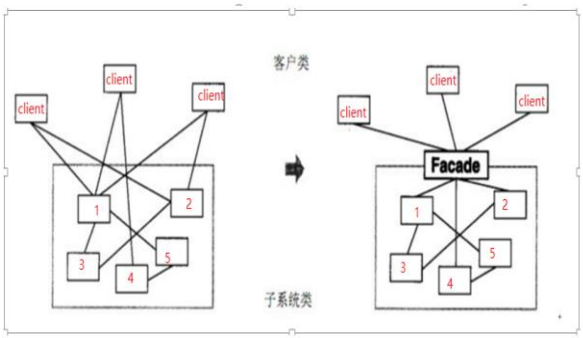
18.门面（外观）模式 Facade

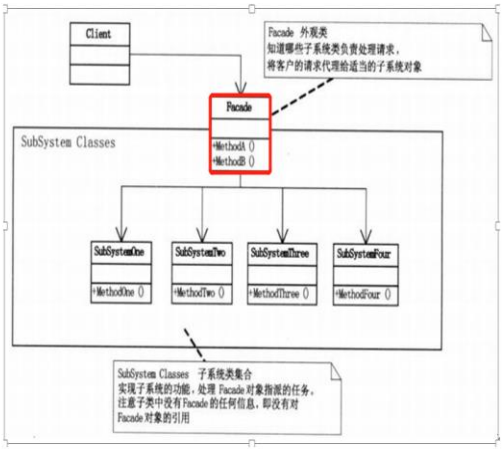
子系统内部和外部(client)之间的解耦合（用户只需要和 Facede 交互）

解决的问题：客户类不需要与子系统中的类直接打交道，而是通过 Façade 提供接口与子系统中的类打交道→→客户端不直到子系统中类的实现细节，只调用 Façade 提供的接口即可→→使得 client 与子系统中的类解耦合，系统维护更加简单。

举例：大力想组装电脑，如果自己组装电脑，那么他需要和 cpu，主板，硬盘等各个厂商直接打交道，自己动手，丰衣足食；但是他嫌麻烦，因此大力直接找到组装电脑的厂家(facade)，只和组装电脑的厂家打交道，让厂家去和 cpu，主板，硬盘等各个厂商打交道→→大大节省了时间。

举例：东东找炒股公司炒股





```cpp
class CPU
{
public:
    void buyCPU(){ cout << "购买CPU" << endl; }
};
class Memory
{
public:
    void buyMemory(){ cout << "购买Memory" << endl; }
};
class NVIDIA
{
public:
    void buyNVIDIA(){ cout << "购买显卡" << endl; }
};
```

```cpp
class Facade
{
private:
    CPU* cpu;
    Memory* memory;
    NVIDIA* nvidia;
public:
    Facade(CPU* cpu, Memory* memory, NVIDIA* nvidia) :cpu(cpu), memory(memory), nvidia(nvidia){}
    void DIY()
    {
        cpu->buyCPU();
        memory->buyMemory();
        nvidia->buyNVIDIA();
    }
};
```

包含了所有子系统中类对象的引用

```cpp
class Client
{
private:
    Facade* facade;
public:
    Client(Facade* facade) :facade(facade){}
    void DIY(){ facade->DIY(); }
};
```

```cpp
int main()
{
    CPU* cpu = new CPU();
    Memory* memory = new Memory();
    NVIDIA* nvidia = new NVIDIA();

    Facade* facade = new Facade(cpu, memory, nvidia);

    Client* client = new Client(facade);
    client->DIY();
}
```

client只需要调用一个接口，不需要直到内部子系统的具体实现，就可以完成想要的功能，该功能由Facade完成

19.中介者模式

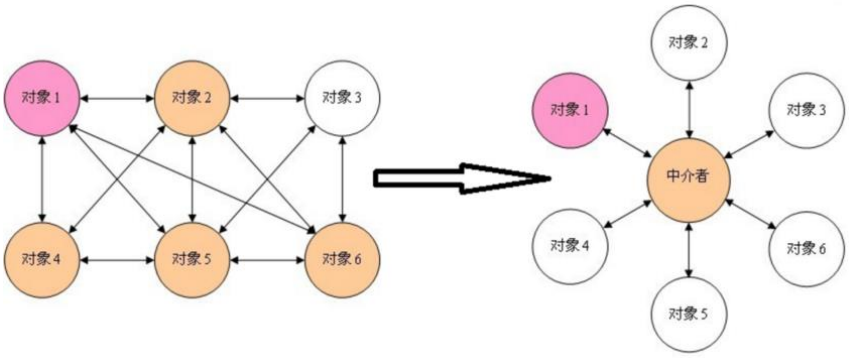
概念：用一个中介者 Mediator 对象，封装一系列的对象交互；使得对象之间不用强耦合

实现技巧：Mediator 有交互类的指针/引用；交互类的函数接口中有 Mediator 的指针/引用

举例 1：类 A 和类 B 交互，如 A 类要修改 B 类的内容，B 类也要修改 A 类的内容

解决方案 1：A 类中组合 B，B 类中组合 A，两个类可以直接修改→但是这种强耦合关系带来维护上的困难，尤其是在后来系统扩展时，交互类的个数增加并且各个类之间的交互变得复杂时，越来越力不从心

解决方案 2：A 类和 B 类都与中介类 Mediator 打交道，Mediator 类中包含 A 和 B



举例 2：QQ 和微信平台的使用——年轻人在进行聊天时，根本不用直到聊天工具内部子系统的设计，它们只需要直到如何使用聊天平台设计的接口(Facade)就可以了，至于剩下的消息传输的工作全部交给聊天平台。

```cpp
class A
{
private:
    string str;
public:
    A(string str) :str(str){}
    void setStr(string str){ this->str = str; }
    void changeBstr(Mediator* mediator, string str);
};
class B
{
private:
    string str;
public:
    B(string str) :str(str){}
    void setStr(string str){ this->str = str; }
    void changeAstr(Mediator* mediator, string str);
};
```

```cpp
class Mediator
{
private:
    A* a;
    B* b;
public:
    Mediator(A* a, B* b) :a(a), b(b){}
    A* getA(){ return a; }
    B* getB(){ return b; }
    void changeAstr(string str){ a->setStr(str); }
    void changeBstr(string str){ b->setStr(str); }
};
```

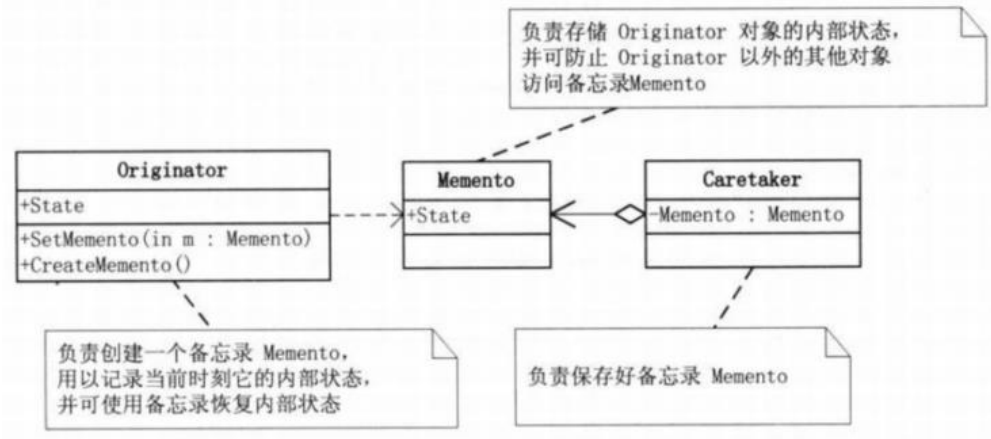
中介者中，有交互类的指针

```cpp
void A::changeBstr(Mediator* mediator, string str){ mediator->getB()->setStr(str); }
void B::changeAstr(Mediator* mediator, string str){ mediator->getA()->setStr(str); }
```

```cpp
int main()
{
    A* a = new A("wl");
    B* b = new B("gjw");
    Mediator* mediator = new Mediator(a, b);
    a->changeBstr(mediator, "gjw better");
    b->changeAstr(mediator, "wl beautiful");
}
```

20.备忘录模式

在不破坏封装性的前提下，捕获一个对象的内部状态，并在对象之外保存这个状态。这样以后就可将该对象恢复到原先的保存状态



备忘录类：必须具有 Originator 的所有的属性变量

Originator 类：具有 CreateMemento 和 LoadMemento 接口

为了简单理解：该程序，暂时不考虑备忘录管理者，只包括 Original 和 Memento

```cpp
class GameRole
{
private:
    int m_vitality; //生命值
    int m_attack;   //进攻值
    int m_defense;  //防守值
public:
    GameRole() : m_vitality(100), m_attack(100), m_defense(100) {}
    Memento* CreateMemento() //备份状态
    {
        return new Memento(m_vitality, m_attack, m_defense);
    }
    void LoadMemento(Memento* memento)
    {
        m_vitality = memento->getVitality();
        m_attack = memento->getAttack();
        m_defense = memento->getDefense();
    }
    void showAttr()
    {
        cout << m_vitality << "+" << m_attack << "+" << m_defense << endl;
    }
};
```

```cpp
class Memento //备忘录类中具有和GameRole类相同的属性
{
public:
    int m_vitality; //生命值
    int m_attack;   //进攻值
    int m_defense;  //防守值
public:
    Memento(int vitality, int attack, int defense) :
        m_vitality(vitality), m_attack(attack), m_defense(defense){}
    int getVitality(){ return m_vitality; }
    int getAttack(){ return m_attack; }
    int getDefense(){ return m_defense; }
};
```

```cpp
int main()
{
    GameRole* role = new GameRole();
    Memento* memento = role->CreateMemento();

    GameRole* tmp = new GameRole();
    tmp->LoadMemento(memento);
    tmp->showAttr();
}
```

21.享元（共享）模式 Flyweight

定义：对大量细粒度对象进行共享

常见的例子：在 word 文档中，有大量的文字，每个文字都是一个对象

需求：修改 word 文字对象的字体格式

解决方案 1：给每一个文字对象都创建一个字体格式对象，供文字对象改变字体格式使用。这是十分占用内存资源的：试想一下，一篇文章又成千上万的文字，那么就对应有同样个数的字体对象，这是多么庞大的开销啊。

思考：由于字体格式的操作都是一样的，那为啥不让所有的文字对象都使用同一个字体格式对象呢？

解决方案 2：将字体格式类变成一个共享资源，让每个文本对象都使用这同一个字体格式对象，那么无论在多的字体出现，都只对应一个字体格式对象，这样的话，节省了巨大的开销。

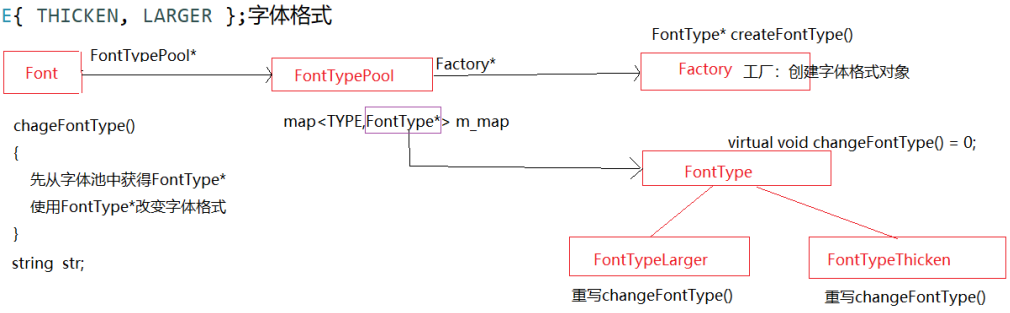
实现技巧：

Flyweight 功能对象类：实现享元类的功能

FactoryFlyweight 类：创建享元对象

FlyweightPool：维护多个不同的享元对象的池子

```cpp
enum TYPE{ THICKEN, LARGER };字体格式
```



代码实现：享元模式，简单工厂模式，享元池（管理每种享元对象）

```cpp
enum TYPE{ THICKEN, LARGER };
class FontType 字体格式
{
public:
    virtual void chageFont(string str) = 0;
};
class FontTypeTHICKEN :public FontType
{
public:
    void chageFont(string str)
    {
        cout << str << "加粗" << endl;
    }
};
class FontTypeLARGER :public FontType
{
public:
    void chageFont(string str)
    {
        cout << str << "变大" << endl;
    }
};
```

```cpp
class Factory
{
public:        生产字体格式
    FontType* createFontType(TYPE type)
    {
        switch (type)
        {
        case THICKEN:
            return new FontTypeTHICKEN()
        case LARGER:
            return new FontTypeLARGER();
        default:
            return new FontTypeTHICKEN()
        }
    }
};
```

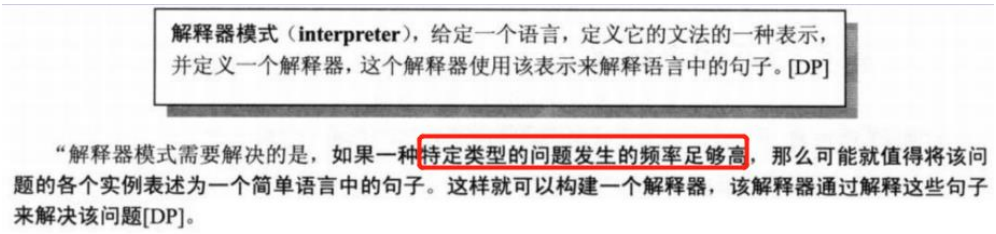
```cpp
class FontTypePool 字体池: 获取/添加字体格式
{
private:
    Factory* factory;  //简单工厂,用于创建字体类型
    map<TYPE, FontType*> m_map;
public:
    FontType* getFontType(TYPE type)
    {
        map<TYPE, FontType*>::iterator it = m_map.find(type);
        if (it != m_map.end())
            return it->second;
        else
        {
            FontType* tmp = factory->createFontType(type);
            m_map.insert(pair<TYPE, FontType*>(type, tmp));
            return tmp;
        }
    }
};
```

```cpp
class Font 字体类
{
private:
    FontTypePool* pool;  字体池
    string str;
public:
    Font(string str) :str(str){ pool = new FontTypePool(); }  }
    void chageFont(TYPE type) 改变字体
    {
        FontType* fontType = pool->getFontType(type);
        fontType->chageFont(str);
    }
};
```

```cpp
int main()
{
    Font* font = new Font("gjw");

    font->chageFont(THICKEN);
    font->chageFont(LARGER);
    font->chageFont(LARGER);
}
```

```
C:\Users\Guc
gjw加粗
gjw变大
gjw变大
```

22.解释器模式



23.迭代器模式

顺序访问集合中的每一个元素，而又无需暴露该对象的内部表示

常见：STL 中的 iterator

24.访问者模式 Visitor