

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

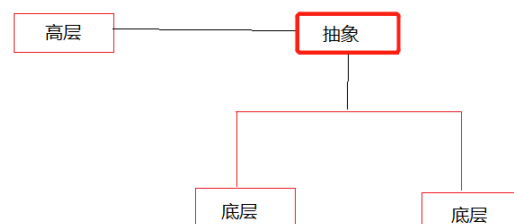
定义：对扩展开放，对修改封闭

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

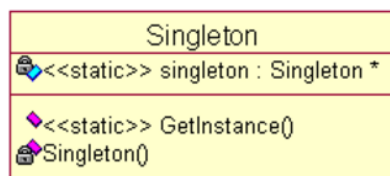
特点:

- (1) 确保类只有一个实例: `private static Singleton* m_singleton;`
- (2) 自行实例化:
- (3) 提供外界可以获得该实例的方法: `public static Singleton* GetInstance()`

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。

代码实现:



```
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

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

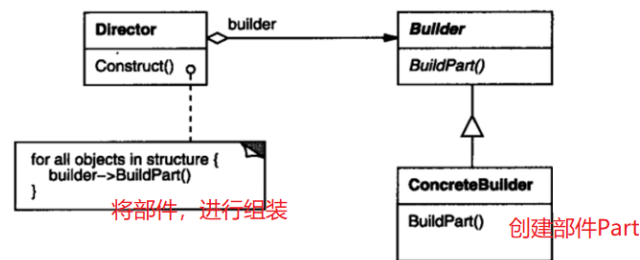
02.Builder 模式

将一个复杂对象的**构建（组装）**和**表示（部分创建）**分离→相同的构建过程，获得不同的构建结果。

举例：建房子

- (1) 房子的墙，门，窗户的规格不一样，在不同的厂子 builder 中生产
- (2) 但是，建造房子时，组装墙，门，窗户的顺序都是一致的

最后生产出来的房屋，组装顺序虽然相同，但是由于不同工厂生产的墙，门，窗户的规格不同，导致最后建造的房屋也具有很多样貌。



```
class Builder 虚类
{
public:
    virtual void buildHead(){}
    virtual void buildLeg(){}
};

class BuilderPerson :public Builder
{
public:
    实现类
    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()
    {
        m_builder->buildHead();
        m_builder->buildLeg();
    }
};

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

创建Head
创建Leg

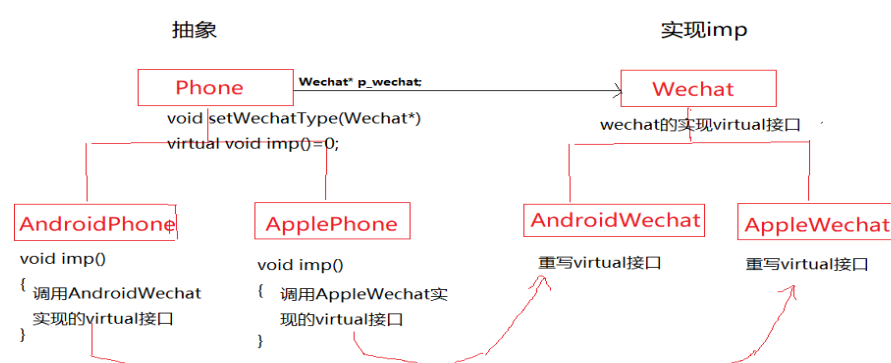
03.桥接模式

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。

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



```
//软件: Android版本的软件 Apple版本的软件
class Wechat 虚基类, 抽象出来的行为interface
{
public:
    virtual void run() = 0; //相同的行为抽象成接口
};

class AndroidWechat :public Wechat
{
public:
    //Android特有的行为
    void runAndroid(){ cout << "Android版本的wechat正在运行" << endl; }
public:
    void run(){ runAndroid(); } 重写interface
};

class AppleWechat :public Wechat
{
public:
    //Apple特有的行为
    void runApple(){ cout << "Android版本的wechat正在运行" << endl; }
public:
    void run(){ runApple(); } 重写interface
};

//手机: Android , Apple
class Phone 调用interface的平台虚基类
{
protected:
    Wechat* p_wechat; 里面有Wechat*指针
public:
    将Phone平台, 与Wechat接口进行绑定
    void setWechat(Wechat* p_wechat){ this->p_wechat = p_wechat; }
    virtual void run() = 0; run();调用Wechat的抽象interface
};

class AndroidPhone:public Phone
{
public: 重写
    void run(){ p_wechat->run(); }
};

class ApplePhone :public Phone
{
public: 重写
    void run(){ p_wechat->run(); }
};

int main() //主函数
{
    Wechat* p_AndroidWechat = new AndroidWechat(); 定义interface
    Phone* p_AndroidPhone = new AndroidPhone(); 定义平台
    p_AndroidPhone->setWechat(p_AndroidWechat); 将平台和interface进行绑定
    p_AndroidPhone->run(); 平台通过run()调用interface中的run(), 由于interface已经重写了run(), 因此调用绑定的interface重写的run()
    Wechat* p_AppleWechat = new AppleWechat();
    Phone* p_ApplePhone = new ApplePhone();
    p_ApplePhone->setWechat(p_AppleWechat);
    p_ApplePhone->run();
}
```

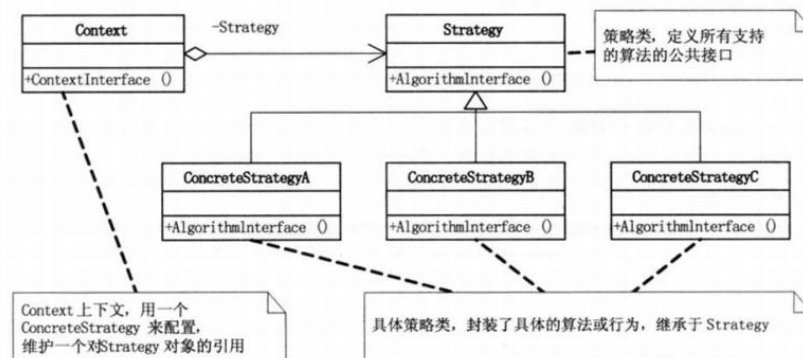
罗克韦尔实习中使用过 Bridge 模式：

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

04.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。

```
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; }
};

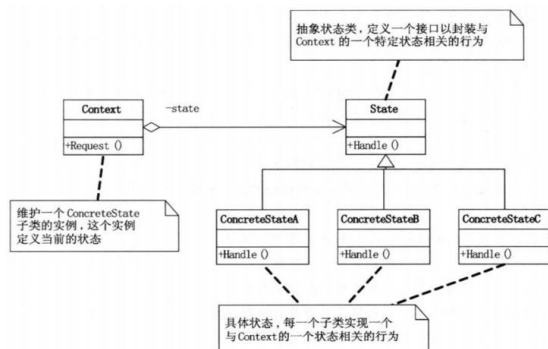
class Context
{
private:
    double OriginalPrice; //商品原价
    Strategy* m_strategy; //包含打折策略的指针用Context去配置Stragety
public:
    Context(double OriginalPrice){ this->OriginalPrice = OriginalPrice; }
public:
    void SetSoldStrategy(Strategy* m_strategy){ this->m_strategy = m_strategy; } //绑定销售策略
    double GetPrice(){ return m_strategy->GetCurPrice(OriginalPrice); } //获得商品打折后的价格
};

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
}
```

05.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)

```
class State
{
public:
    virtual void doSth(Worker* worker) = 0;
};
class forenoonState : public State
{
public:
    void doSth(Worker* worker);
};
class NoonState : public State
{
public:
    void doSth(Worker* worker);
};
class EveningState : public State
{
public:
    void doSth(Worker* worker);
};
class AfterworkState : public State
{
public:
    void doSth(Worker* worker);
};

#include<iostream>
using namespace std;

class Worker;
class forenoonState;
class NoonState;
class EveningState;
class AfterworkState;

void forenoonState::doSth(Worker* worker)
{
    if (worker->getHour() < 12)
        cout << "上午工作，精神百倍" << endl;
    else
    {
        delete(worker->getState());
        worker->setState(new NoonState());
        worker->getState()->doSth(worker);
    }
}

void NoonState::doSth(Worker* worker)
{
    if (worker->getHour() < 13)
        cout << "饿了，午饭，犯困，午休" << endl;
    else
    {
        delete(worker->getState());
        worker->setState(new EveningState());
        worker->getState()->doSth(worker);
    }
}

void EveningState::doSth(Worker* worker)
{
    if (worker->getHour() < 20)
        cout << "加班哦，疲惫之极" << endl;
    else
    {
        delete(worker->getState());
        worker->setState(new AfterworkState());
        worker->getState()->doSth(worker);
    }
}

void AfterworkState::doSth(Worker* worker)
{
    cout << "终于下班了，回家喽" << endl;
}

int main()
{
    while (1)
    {
        Worker* worker = new Worker(new forenoonState());
        for (int i = 1; i <= 24; i++)
        {
            worker->setHour(i);
            worker->doSth();
        }
    }
}
```

```
C:\Users\Guo\awee\Desktop
上午工作，精神百倍
上午工作，精神百倍
上午工作，精神百倍
上午工作，精神百倍
上午工作，精神百倍
上午工作，精神百倍
上午工作，精神百倍
上午工作，精神百倍
上午工作，精神百倍
饿了，午饭，犯困，午休
加班哦，疲惫之极
加班哦，疲惫之极
加班哦，疲惫之极
加班哦，疲惫之极
加班哦，疲惫之极
加班哦，疲惫之极
终于下班了，回家喽
终于下班了，回家喽
终于下班了，回家喽
终于下班了，回家喽
终于下班了，回家喽
```

06.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

```
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; }
};

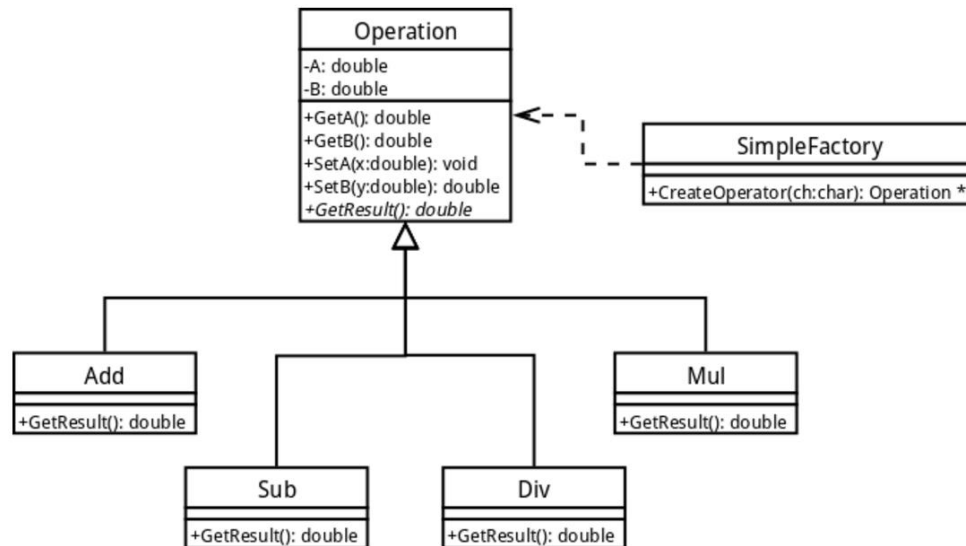
class SonB :public Parents
{
public:
    void step1(){ cout << "B step1" << endl; }
    void step2(){ cout << "B step2" << endl; }
};

int main()
{
    Parents* sonA = new SonA();
    sonA->run();

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

07.简单工厂模式

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。



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* CreateOperator(char c) //工厂模式中只有一个函数，用来生产Operate对象
    {
        switch (c)
        {
            case '+':
                return new Add(); //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
            case '-':
                return new Sub();
            case '*':
                return new Mul();
        }
    }
};
```

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 factory 模式：

创建 IPCBase 对象时，使用 switch...case...判断输入字符串的是 CriticalSection，Mutex，Semaphore，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

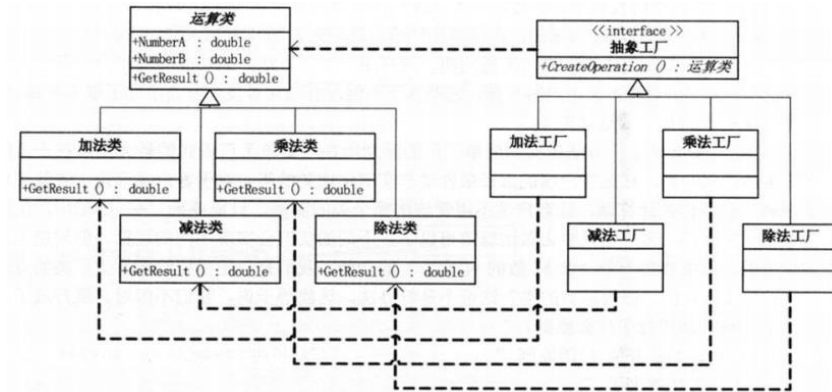
那么如果使用简单工厂方法，需要在 switch...case...中加入 case '/' →→→这样破坏了“封闭原则”，程序的改动非常大→→→因此简单工厂模式在实际的应用中比较少，缺点很明显→→→引出**工厂方法**

08.工厂方法 Factory Method

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();

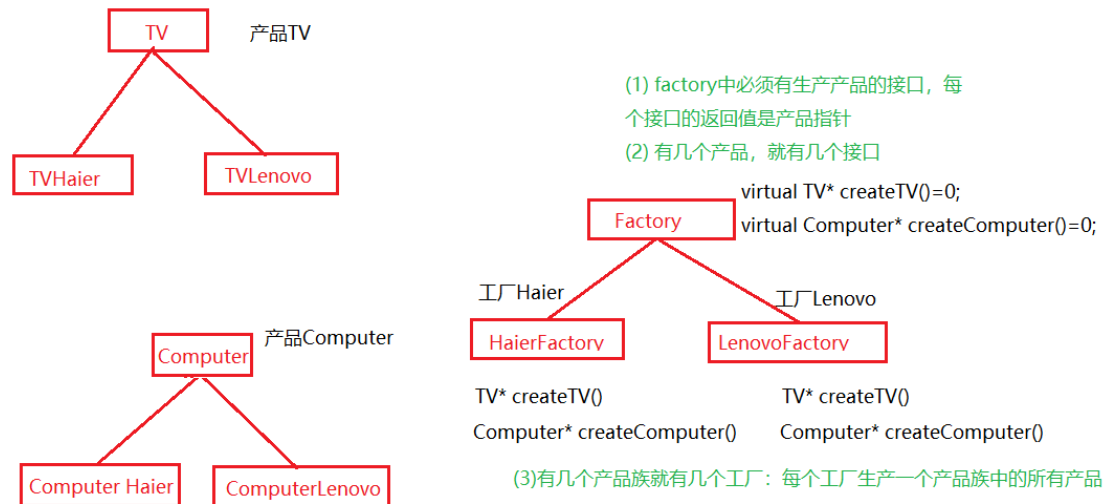
    m_operate->setA(10);
    m_operate->setB(10);
    int res = m_operate->operate();
}
```

09.抽象工厂

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



总结：

简单工厂：工厂类中，根据条件决定一个接口由哪个具体产品类来实现。

工厂方法：在简单工厂的基础上，将工厂抽象出来→每个工厂对应一个产品

生产 TV 和 Computer，用两个工厂 FactoryTV 和 FactoryComputer

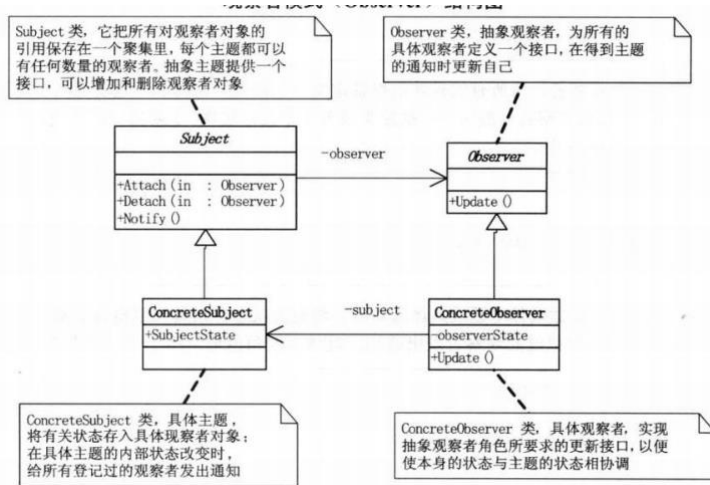
抽象工厂：

有几个产品，Factory 中就有几个 create 方法；

有几个产品族，就有几个 Factory 工厂子类（每个工厂，生产产品族中的所有产品）

10.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



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();
}
```

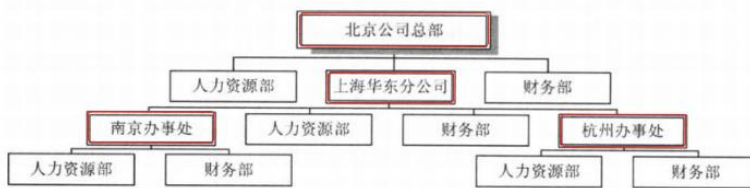
11. 组合模式（树形结构）

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：



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。

```
class Component
{
protected:
    list<Component*> m_list;
public:
    virtual void Add(Component* com) = 0;
    virtual void Remove(Component* com) = 0;
public:
    virtual void HandleRequest() = 0;
};

class Leaf:public Component
{
public:
    void HandleRequest(){ cout << "Leaf" << endl; }
public:
    void Add(Component* com){}
    void Remove(Component* com){}
};

class Composite :public Component
{
public:
    Composite可以“组合”多个Leaf, 多个Composite组件
    void Add(Component* com){ m_list.push_back(com); }
    void Remove(Component* com){ m_list.remove(com); }
public:
    void HandleRequest()
    {
        cout << "Composite" << endl; 实现Composite独有的功能
        for (auto elem : m_list)
        {
            elem->HandleRequest(); 实现该Composite对象组合的 Leaf和Composite的功能
        }
    }
};

int main()
{
    Component* leaf = new Leaf();
    Component* com = new Composite();

    com->Add(leaf); //Composite对象组合Leaf
    com->Add(leaf); //Composite对象组合Leaf

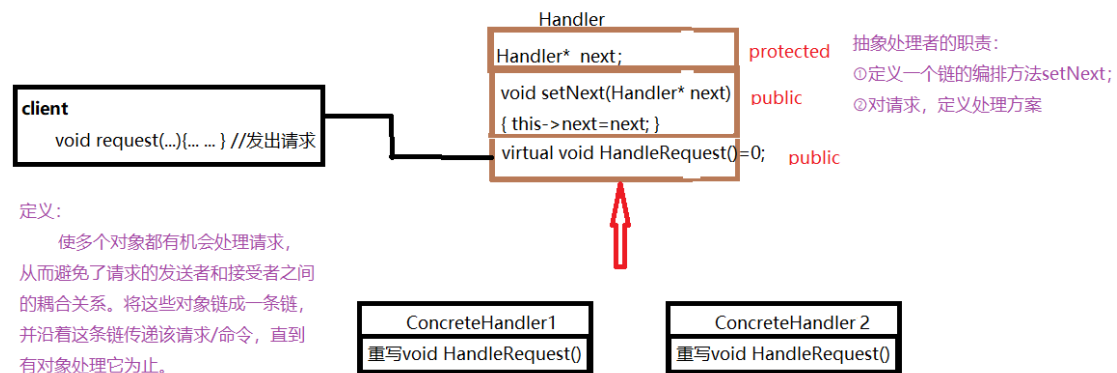
    com->HandleRequest();
}
```

缺点：Leaf没有组合其他Component的能力，但是为了能够实例化Leaf对象，还是要重写Add和Remove

总结：Leaf和Composite都对外实现一致的接口HandleRequest，但是却执行不同的功能

组合对象Composite可以将单一对象Leaf组装起来，在调用HandleRequest函数时，不但实现自己独有的处理，还能实现Leaf中的HandleRequest函数

12. 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。

```
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;
        }
    }
};

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

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

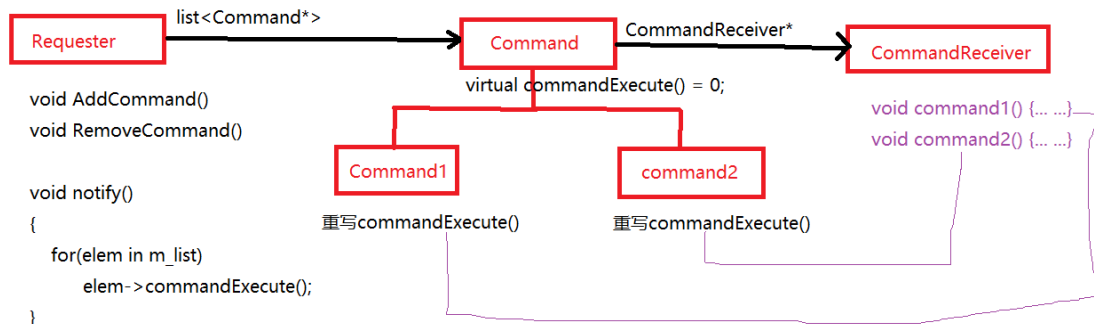
    m_Manager->setNext(m_Director);
    m_Director->setNext(NULL);

    m_Manager->HandleRequest(value);
}

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

13. 命令模式

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。

```
7 class Barbecur // 命令执行者
8 {
9 public:
10     void MakeMutton() { cout << "烤羊肉串" << endl; }
11     void MakeChickwing() { cout << "烤鸡翅" << endl; }
12 };

class Command // 命令
{
protected:
    Barbecur* barbecur; // 命令中包含命令执行者
public:
    Command(Barbecur* barbecur) : barbecur(barbecur) {}
    virtual void Execute() = 0; // 只有一个虚函数
};

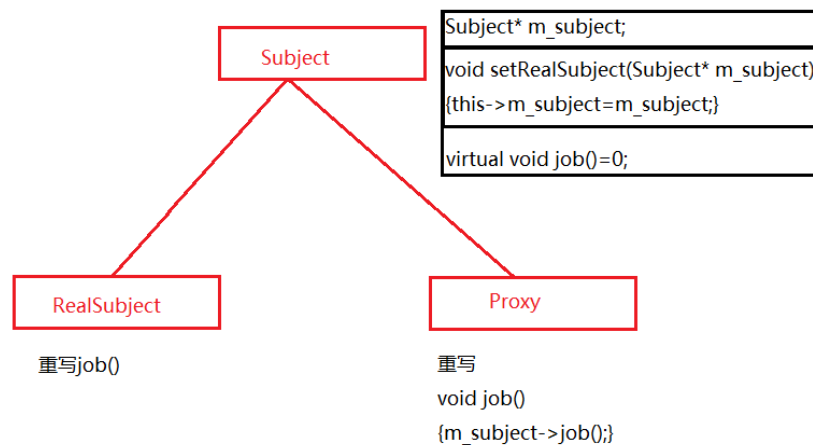
class MakeMuttonCommand : public Command
{
public:
    MakeMuttonCommand(Barbecur* barbecur) : Command(barbecur) {}
    void Execute() { barbecur->MakeMutton(); }
};

class MakeChickwingCommand : public Command
{
public:
    MakeChickwingCommand(Barbecur* barbecur) : Command(barbecur) {}
    void Execute() { barbecur->MakeChickwing(); }
};

7 class Waiter // 命令请求者
8 {
9 private:
10     list<Command*> m_list; // 命令列表
11 public:
12     void AddCommand(Command* command) { m_list.push_back(command); }
13     void RemoveCommand(Command* command) { m_list.remove(command); }
14
15     void notify() // 一次通知，所有请求的命令都执行
16     {
17         for (auto elem : m_list)
18             elem->Execute();
19     }
20 };

2 int main()
3 {
4     Barbecur* barbecur = new Barbecur(); // 命令执行者
5
6     Command* makechickwingcommand = new MakeChickwingCommand(barbecur); // 可选命令
7     Command* makemuttoncommand = new MakeMuttonCommand(barbecur); // 可选命令
8
9     Waiter* girl = new Waiter(); // 命令请求者
10    girl->AddCommand(makemuttoncommand);
11    girl->AddCommand(makechickwingcommand);
12    girl->notify(); // 一次通知，执行所有的命令
13 }
14 }
```

14.代理模式



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：

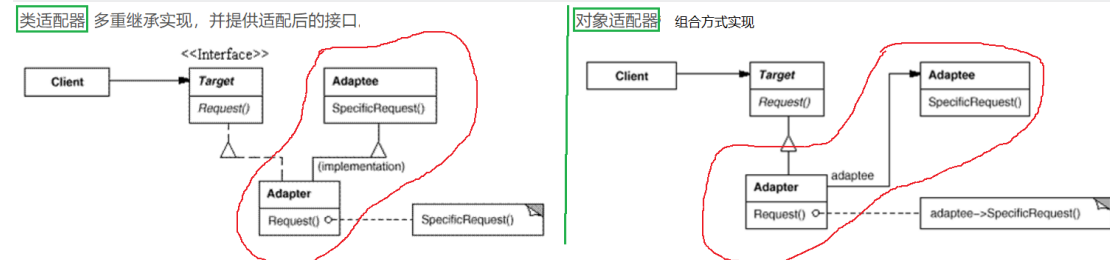
在代理模式中，有两个角色——任务委托者（大力），任务执行者（东东），东东是大力的代理。执行送花动作的人虽然是东东，但是实际上 **RealSubject** 实际主题确实大力。

```
3 class Subject
9 {
9     protected:
1         Subject* m_subject;
2     public: //给Proxy设置RealSubject
3         void setRealSubject(Subject* m_subject){ this->m_subject = m_subject; }
4     public:
5         virtual void job() = 0; 代理的任务
5     };
7 class RealSubject :public Subject
3 {
9     public:      RealSubject执行的job
9         void job(){ cout << "大力送花给探探" << endl; }
1    };
2 class Proxy :public Subject
3 {
4     public:      代理帮RealSubject完成job
5         void job(){ m_subject->job(); }
5     };
- }
```

15. 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();
    }
    ...
};
```

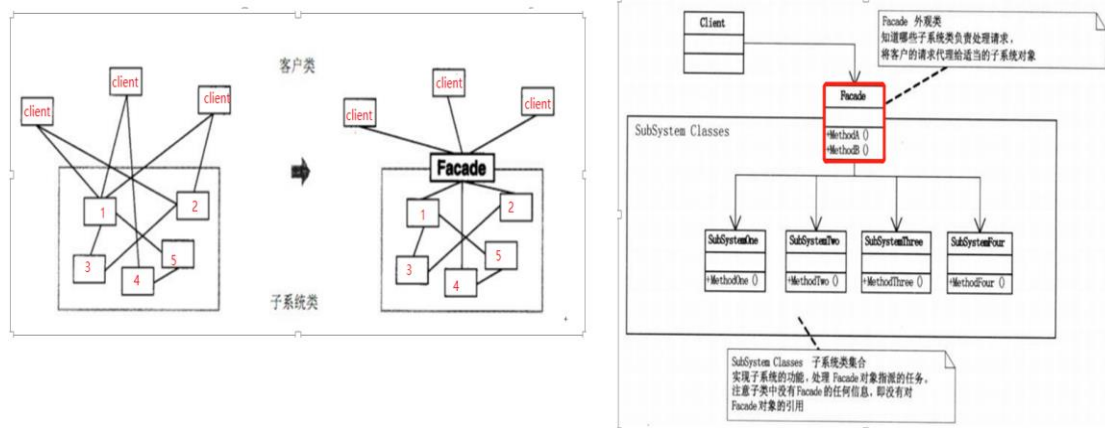

16.门面（外观）模式 Facade

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

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



```
3 class CPU
4 {
5 public:
6     void buyCPU(){ cout << "购买CPU" << endl; }
7 };
8 class Memory
9 {
10 public:
11     void buyMemory(){ cout << "购买Memory" << endl; }
12 };
13 class NVIDIA
14 {
15 public:
16     void buyNVIDIA(){ cout << "购买显卡" << endl; }
17 };
18
19 class Client
20 {
21 private:
22     Facade* facade;
23 public:
24     Client(Facade* facade) : facade(facade){}
25     void DIY(){ facade->DIY(); }
26 };
27
28 class Facade
29 {
30 private:
31     CPU* cpu;
32     Memory* memory;
33     NVIDIA* nvidia;
34 public:
35     Facade(CPU* cpu, Memory* memory, NVIDIA* nvidia) : cpu(cpu), memory(memory), nvidia(nvidia){}
36     void DIY()
37     {
38         cpu->buyCPU();
39         memory->buyMemory();
40         nvidia->buyNVIDIA();
41     }
42 };
43
44 int main()
45 {
46     CPU* cpu = new CPU();
47     Memory* memory = new Memory();
48     NVIDIA* nvidia = new NVIDIA();
49
50     Facade* facade = new Facade(cpu, memory, nvidia);
51
52     Client* client = new Client(facade);
53     client->DIY();
54 }
```

包含了所有子系统中类
对象的引用

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

17. 中介者模式

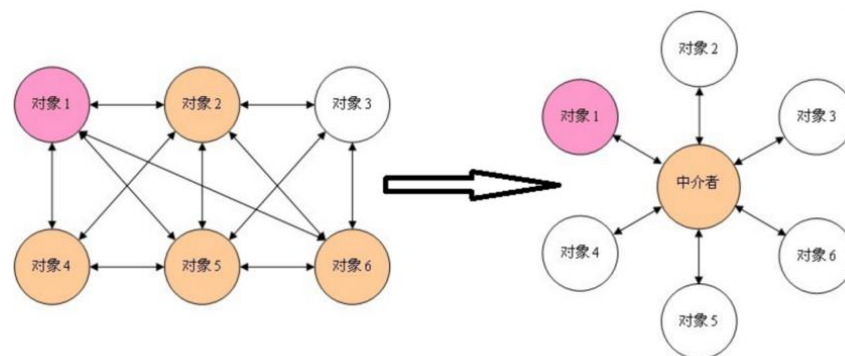
概念：用一个中介者 Mediator 对象，封装一系列的对象交互；使得对象之间不用强耦合

实现技巧：Mediator 有交互类的指针

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。

```
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);
};

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

class Mediator
{
private:
    A* a;  // 中介者中，有交互类的指针
    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); }
};

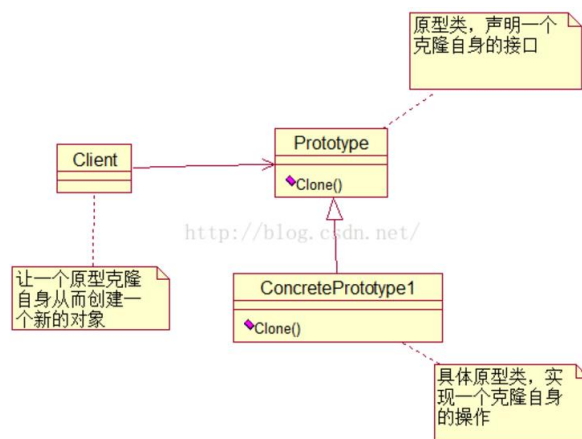
int main()
{
    A* a = new A("wl");
    B* b = new B("gfw");
    Mediator* mediator = new Mediator(a, b);
    a->changeBstr(mediator, "gfw better");
    b->changeAstr(mediator, "wl beautiful");
}
```

18. 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。

注意：拷贝构造函数必须写对

举例：简历复制

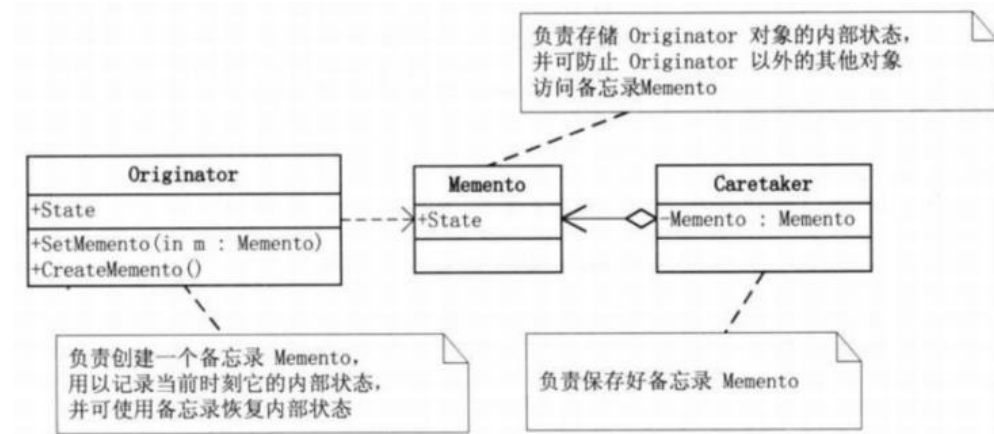


```
class Resume
{
private:
    string name;
    int age;
public:
    Resume(string name, int age) :name(name), age(age){}
public:
    Resume* Clone()
    {
        return new Resume(*this); //浅拷贝
    }
};

int main()
{
    Resume* gjw = new Resume("gjw", 24);
    Resume* wl = gjw->Clone();
}
```

19. 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**

```
9 class GameRole
10 {
11 private:
12     int m_vitality; //生命值
13     int m_attack; //进攻值
14     int m_defense; //防守值
15 public:
16     GameRole() : m_vitality(100), m_attack(100), m_defense(100) {}
17     Memento* CreateMemento() //备份状态
18     {
19         return new Memento(m_vitality, m_attack, m_defense);
20     }
21     void LoadMemento(Memento* memento)
22     {
23         m_vitality = memento->getVitality();
24         m_attack = memento->getAttack();
25         m_defense = memento->getDefense();
26     }
27     void showAttr()
28     {
29         cout << m_vitality << "+" << m_attack << "+" << m_defense << endl;
30     }
31 };

7 class Memento //备忘录类中具有和GameRole类相同的属性
8 {
9 public:
10     int m_vitality; //生命值
11     int m_attack; //进攻值
12     int m_defense; //防守值
13 public:
14     Memento(int vitality, int attack, int defense) :
15         m_vitality(vitality), m_attack(attack), m_defense(defense){}
16     int getVitality(){ return m_vitality; }
17     int getAttack(){ return m_attack; }
18     int getDefense(){ return m_defense; }
19 };

3 int main()
4 {
5     GameRole* role = new GameRole();
6     Memento* memento = role->CreateMemento();
7
8     GameRole* tmp = new GameRole();
9     tmp->LoadMemento(memento);
10    tmp->showAttr();
11 }
```

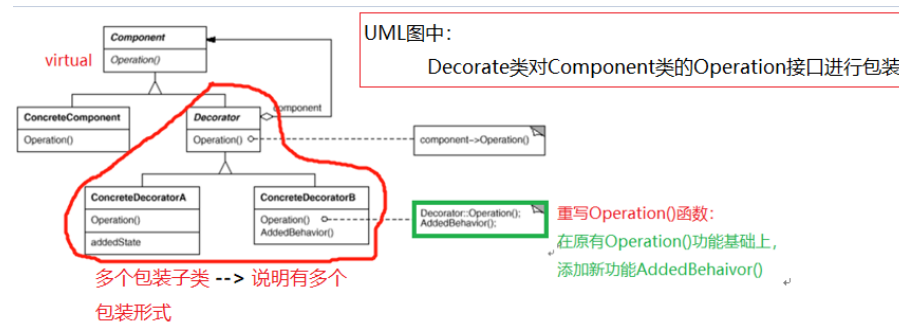
20.装饰模式

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，如：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。



```
class Tea 被装饰的类
{
public:
    要进行功能扩展的接口
    virtual void getDescription(){ cout << "Tea " << endl; }
};
class CoffeTea :public Tea
{
public:
    要进行功能扩展的接口
    void getDescription(){ cout << "CoffeTea " << endl; }
};
class MochaTea :public Tea
{
public:
    要进行功能扩展的接口
    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();
}
```

```
1 装饰类：对Tea类的接口getDescription()进行功能扩展
2 class Decorate:public Tea
3 {
4     规律：如果一个类Decorate继承和组合同一个类Tea，那么这个
5     Tea* m_tea; 类Decorate十有八九是类Tea的包装类！
6 public:
7     Decorate(Tea* m_tea) :m_tea(m_tea){}
8     virtual void addMateria() = 0;
9 };
10 class DecorateLemon :public Decorate
11 {
12 public:
13     DecorateLemon(Tea* m_tea) :Decorate(m_tea){}
14     void addMateria(){ cout << "Lemon "; }
15     void getDescription()
16     {
17         addMateria();
18         m_tea->getDescription();
19     }
20 };
21 class DecorateApple :public Decorate
22 {
23 public:
24     DecorateApple(Tea* m_tea) :Decorate(m_tea){}
25     void addMateria(){ cout << "Apple " << endl; }
26     void getDescription()
27     {
28         addMateria();
29         m_tea->getDescription();
30     }
31 };
```

21. 享元模式 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。

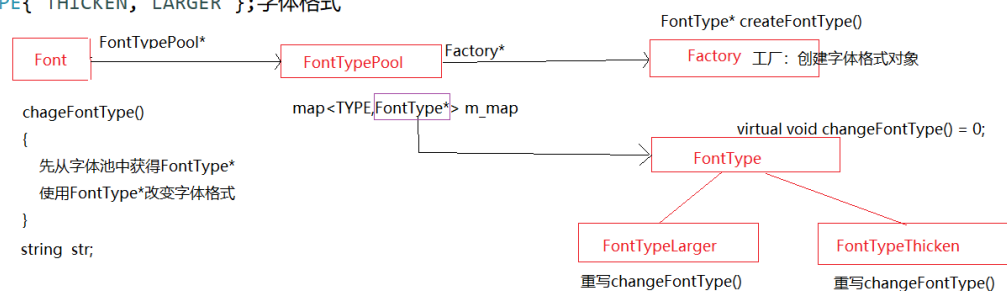
实现技巧：

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

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）

```
5 enum TYPE{ THICKEN, LARGER };
7 class FontType 字体格式
8 {
9 public:
10     virtual void chageFont(string str) = 0;
11 };
12 class FontTypeTHICKEN :public FontType
13 {
14 public:
15     void chageFont(string str)
16     {
17         cout << str << "加粗" << endl;
18     }
19 };
20 class FontTypeLARGER :public FontType
21 {
22 public:
23     void chageFont(string str)
24     {
25         cout << str << "变大" << endl;
26     }
27 };
28
29 class Factory
30 {
31 public:
32     生产字体格式
33     FontType* createFontType(TYPE type)
34     {
35         switch (type)
36         {
37             case THICKEN:
38                 return new FontTypeTHICKEN();
39             case LARGER:
40                 return new FontTypeLARGER();
41             default:
42                 return new FontTypeTHICKEN();
43         }
44     };
45 };
46
47 class FontTypePool 字体池: 获取/添加字体格式
48 {
49 private:
50     Factory* factory; //简单工厂,用于创建字体类型
51     map<TYPE, FontType*> m_map;
52 public:
53     FontType* getFontType(TYPE type)
54     {
55         map<TYPE, FontType*>::iterator it = m_map.find(type);
56         if (it != m_map.end())
57             return it->second;
58         else
59         {
60             FontType* tmp = factory->createFontType(type);
61             m_map.insert(pair<TYPE, FontType*>(type, tmp));
62             return tmp;
63         }
64     };
65 };
66
67 class Font 字体类
68 {
69 private:
70     FontTypePool* pool; 字体池
71     string str;
72 public:
73     Font(string str) :str(str){ pool = new FontTypePool(); }
74     void chageFont(TYPE type) 改变字体
75     {
76         FontType* fontType = pool->getFontType(type);
77         fontType->chageFont(str);
78     };
79 };
80
81 int main()
82 {
83     Font* font = new Font("gfw");
84     font->chageFont(THICKEN);
85     font->chageFont(LARGER);
86     font->chageFont(LARGER);
87 }
```

C:\Users\Gur
gfw加粗
gfw变大
gfw变大

22. 解释器模式

解释器模式 (interpreter), 给定一个语言, 定义它的文法的一种表示, 并定义一个解释器, 这个解释器使用该表示来解释语言中的句子。[DP]

“解释器模式需要解决的是, 如果一种特定类型的问题发生的频率足够高, 那么可能就值得将该问题的各个实例表述为一个简单语言中的句子。这样就可以构建一个解释器, 该解释器通过解释这些句子来解决该问题[DP]。

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