2023年春季学期

软件设计实践

面向可复用性的设计

谢 涛　　马 郓

本讲概览

- 可复用性
  - 复用的形态
- 设计可复用的类
  - 继承
  - 委托
- 面向复用的设计模式
  - 适配器模式、装饰器模式、外观模式
  - 策略模式、模板方法模式

软件复用

- 软件复用：将已有的软件及其有效成分用于构造新的软件
  - 不仅是对程序和代码的复用，还包括对软件开发过程中其它劳动成果的复用，如文档、开发经验、设计决策等
- 复用的优点
  - 降低开发成本
  - 复用的程序已经过充分测试，可靠且稳定
  - 有助于实现标准化，在不同的软件中保持一致
- 复用的代价？

软件复用

- 开发可复用的软件（Development for reuse）
  - 由于要有足够高的适应性，开发成本高于一般软件
  - 由于针对更普适的场景进行开发，缺少足够的针对性，所以性能可能会差一些
- 复用已有软件进行开发（Development with reuse）
  - 通过软件库对可复用软件进行有效的管理
  - 通常无法直接就能使用，需要进行适配

软件复用

白盒复用

- 复制已有代码到正在开发的软件，并进行修改
  - ChatGPT?
- 优点：可定制化程度高
- 缺点：对已有代码的修改增加了软件的复杂度，且需要充分了解其内部实现

黑盒复用

- 通过接口（如类、函数调用）来复用已有的代码，无法修改代码
  - 作业7
- 优点：简单、清晰
- 缺点：某些情况适应性差

可复用的软件来源

- 组织内部的代码库
- 第三方提供的库
  - 如，C++的Boost库
- 程序设计语言自身提供的库
  - 如，C++的标准库、STL
- 来自教材、教程、论坛的代码示例
  - 如，CppReference
- 来自同学和同事的代码
- 已有系统内的代码
- 开源软件的代码
- 自己日积月累的代码

复用的形态

源代码
Source Code

模块
Module

库
Library

框架
Framework

- 复制-粘贴代码
- 可能的问题
  - 维护成本高
    - 同一个代码粘贴到多处地方，改动时需同时修改多处
  - 复制粘贴操作过程中出错
  - 需要看懂被复制的代码
- GitHub代码搜索: github.com/search

复用的形态

源代码
Source Code

模块
Module

库
Library

框架
Framework

- 复用“函数”或“类”
  - 封装性有助于复用
  - 规约或文档对使用者的重要意义
- 可能的问题
  - 版本更新导致后向兼容问题
  - 需要将相关的类或函数链接到一起
- 复用类的两种方式
  - 继承
  - 委托：将某些职责由其他类来完成

复用的形态

源代码
Source Code

模块
Module

库
Library

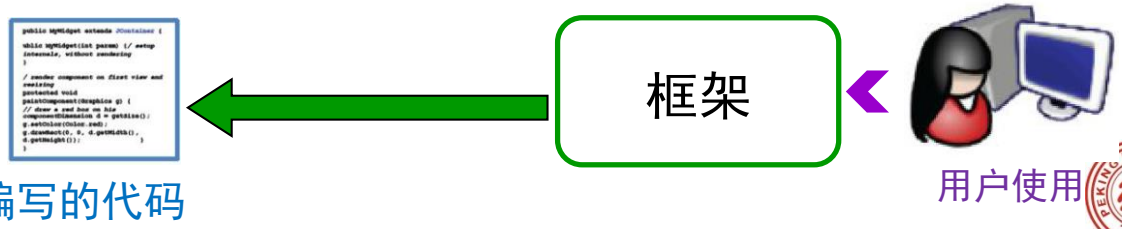
框架
Framework

❑ 库：提供可复用功能的一组类或函数

- ➢ 库提供的类和函数常被称为API（Application Programming Interface，应用编程接口）
- ➢ 开发者在编写软件的代码时调用库提供的API
  - 作业7

❑ 框架：一组类及其之间的连接关系

- ➢ 开发者根据框架的规约，将自己编写的代码填充到框架中，形成完整的软件
  - 大作业



本讲概览

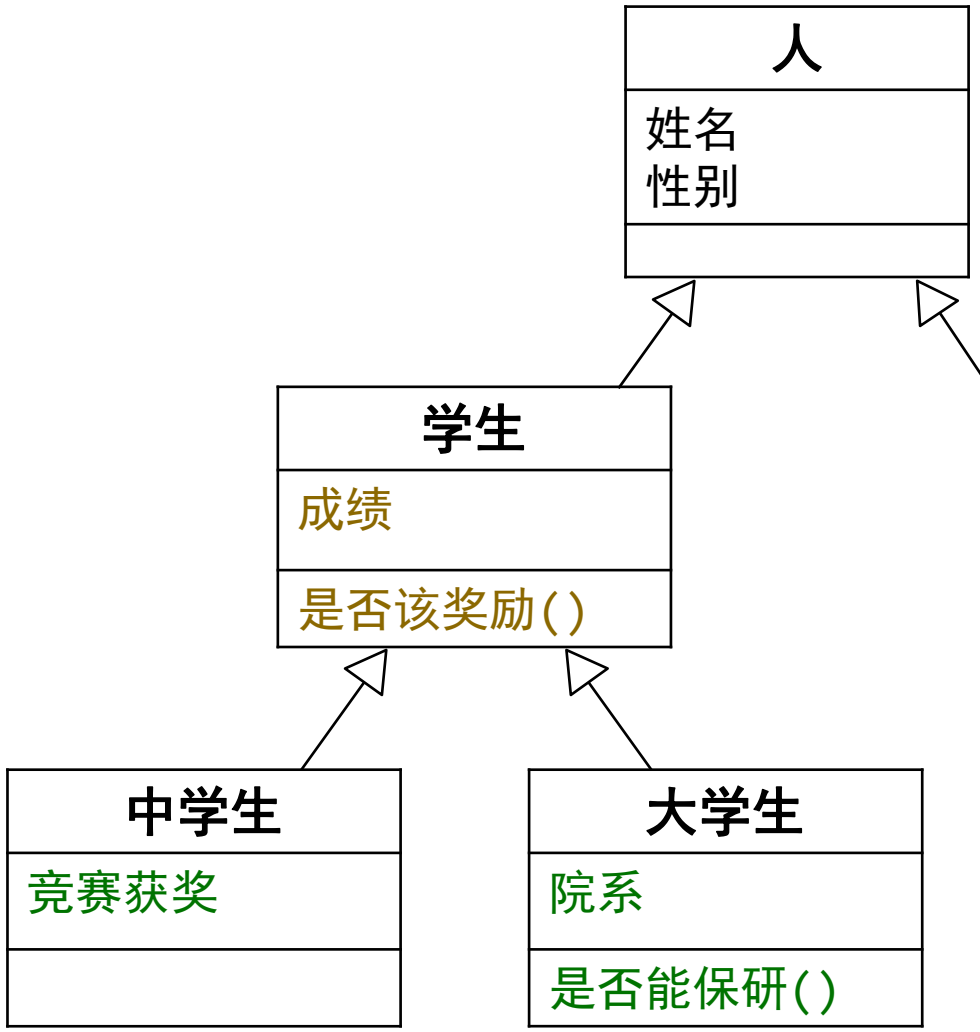
- 可复用性
  - 复用的形态
- 设计可复用的类
  - 继承
  - 委托
- 面向复用的设计模式
  - 适配器模式、装饰器模式、外观模式
  - 策略模式、模板方法模式

回顾：对象之间的关系

- 面向对象将现实世界中的概念抽象为“类”，而把符合这些概念的事物抽象为“对象”
- 现实世界中，两个对象之间可能存在各种各样的关系
  - 泛化：分类法中一般和特殊的关系
    - 是（is-a）：大学生**是**学生
  - 关联：静态结构上的关系，长期的、稳定的
    - 有（has-a）：汽车**有**轮子
    - 部分于（part-of）：例如头是人体的**一部分**
  - 依赖：动态行为上的关系，偶发的、临时性的
    - 使用（uses-a）：程序员**使用**键盘
    - 依赖于（depends-a）：花**依赖于**蜜蜂传粉
- 具体的关系和使用场景有关，需视情况分析

回顾：泛化关系

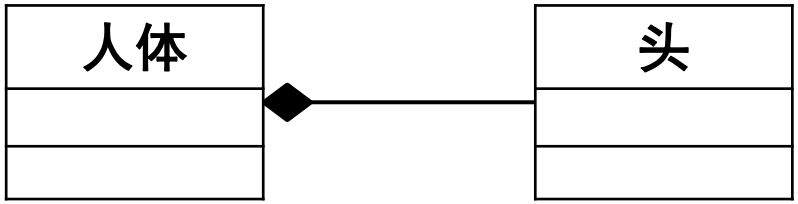
- 使用公有继承机制可以实现“是”关系
  - 私有继承和保护继承一般不能表示泛化关系




```
class Person;

class Student: public Person;

class MiddleSchoolStudent:
public Student;

class UndergraduateStudent:
public Student;

class Teacher: public Person;
```

回顾：关联关系

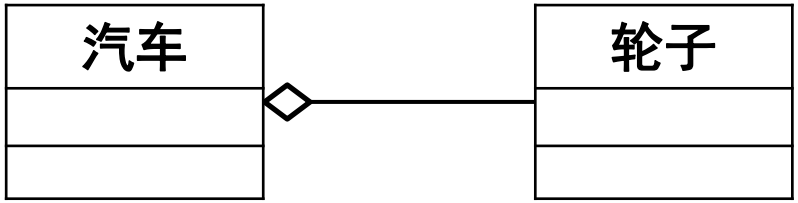
- 使用类的成员变量可以实现关联关系
  - 成员变量使用**类类型对象**可以表示“部分于”关系，UML中称为“组合”关系



UML中表示“组合”关系的符号

```
class Head;

class Body {
    Head h;
};
```

  - 成员变量使用**类类型的指针**可以表示“有”关系，UML中称为“聚合”关系

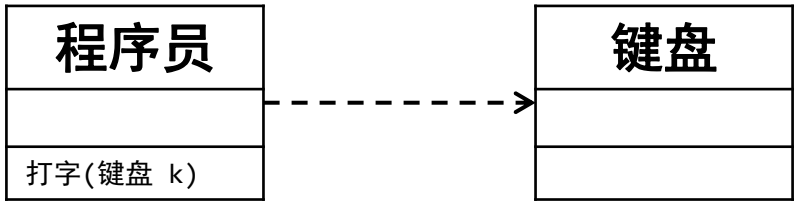


UML中表示“聚合”关系的符号

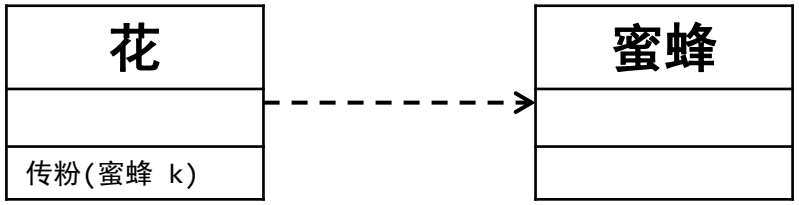
```
class Wheel;

class Car {
    Wheel *w;
};
```

回顾：依赖关系

- 使用类类型的成员函数参数可以实现依赖关系




```
class Keyboard;

class Programmer {
    void input(Keyboard kb);
};
```




```
class Bee;

class Flower {
    void pollinate(Bee b);
};
```

-----> UML中表示“依赖”关系的符号

通过继承实现复用

- 通过继承关系，子类可以继承父类已有的功能，在此基础上增加子类的特殊功能，从而实现复用
- 思考：下面继承关系的设计是否合理？
  - 鸟一般都会飞行，如燕子的飞行速度大概是每小时 120 千米，但是新西兰的几维鸟由于翅膀退化无法飞行

```
class Bird {
public:
    float flySpeed;
    void setSpeed(float v) {
        flySpeed = v;
    }
    float getFlyTime(float dis) {
        return dis/flySpeed;
    }
};
class Swallow : public Bird {

};
class BrownKiwi : public Bird {

};
```

```
int main(){
    Bird* b1 = new Swallow;
    Bird* b2 = new BrownKiwi;
    b1->setSpeed(300);
    b2->setSpeed(0);
    cout<<"b1 fly time is: " <<
b1->getFlyTime(300)<<endl;
    cout<<"b2 fly time is: " <<
b2->getFlyTime(300)<<endl;
}
```

实际上不是所有鸟的v都大于0，因此几维鸟对象替换基类对象时就会出错

通过继承实现复用

- 如何设计继承以减少因复用带来的bug？

里氏替换原则（LSP）

- 继承必须确保父类所拥有的性质在子类中仍然成立，当一个子类的对象能够替换任何其父类的对象时，它们之间才具有泛化（is-a）关系
- 里氏替换原则是继承复用的基石
  - 只有当派生类可以替换掉其基类，而软件功能不受影响时，基类才能真正被复用，派生类也才能够在基类的基础上增加新的行为
- LSP本质：同一个继承体系中的对象应有共同的行为特征
  - 例：企鹅是鸟吗？
  - 生物学：企鹅属于鸟类；
  - LSP原则：企鹅不属于鸟类，因为企鹅不会“飞”
- 违反LSP的后果：有可能需要修改调用父类的代码

通过继承实现复用

- 思考：下面的设计是否违反里氏替换原则？

```
class Rectangle {
public:
    int h, w;
    Rectangle(int h, int w): h(h), w(w) {}
    void scale(int factor) {
        w = w * factor;
        h = h * factor;
    }
};

class Square: public Rectangle {
public:
    Square(int w): Rectangle(w, w) {}
};
```

通过继承实现复用

- 思考：下面的设计是否违反里氏替换原则？

```
class Rectangle {
public:
    int h, w;
    Rectangle(int h, int w): h(h), w(w) {}
    void scale(int factor) {
        w = w * factor;
        h = h * factor;
    }
    void setWidth(int neww) {
        w = neww;
    }
};

class Square: public Rectangle {
public:
    Square(int w): Rectangle(w, w) {}
};
```

```
void scaleW(Rectangle &r, int factor) {
    r.setWidth(r.w * factor);
}
```

通过继承实现复用

里氏替换原则的实践指导

- 子类继承父类时，除添加新的方法完成新增功能外，尽量不要重写父类的方法，除非基类中的纯虚函数必须要重写
  - 虚函数可以重写
  - 普通函数不能或者尽量不要重写
- 子类可以实现父类的抽象方法（即，纯虚函数），但不能覆盖父类的非抽象方法
- 子类中可以增加自己特有的方法
- 当子类的方法重载父类的方法时，方法的前置条件（即方法的输入参数）要比父类的方法更宽松或相等
- 当子类的方法实现父类的方法时（重写/重载或实现抽象方法），方法的后置条件（即方法的的输出/返回值）要比父类的方法更严格或相等

通过委派实现复用

- 委派/委托：类B对象使用类A对象的功能，从而实现对类A的复用
  - 通过某种方式将类A的对象传入类B中
  - 在类B中通过调用类A的成员函数，实现复用

```
class A {
public:
    void foo() {
        cout << "a.bar";
    }
};

class B {
    A* a; //委派连接
public:
    B(A* a): a(a) {}

    void foo() {
        a.foo();
    }
};
```

```
A a;
B b(&a); //建立B和A之间的委派关系
```

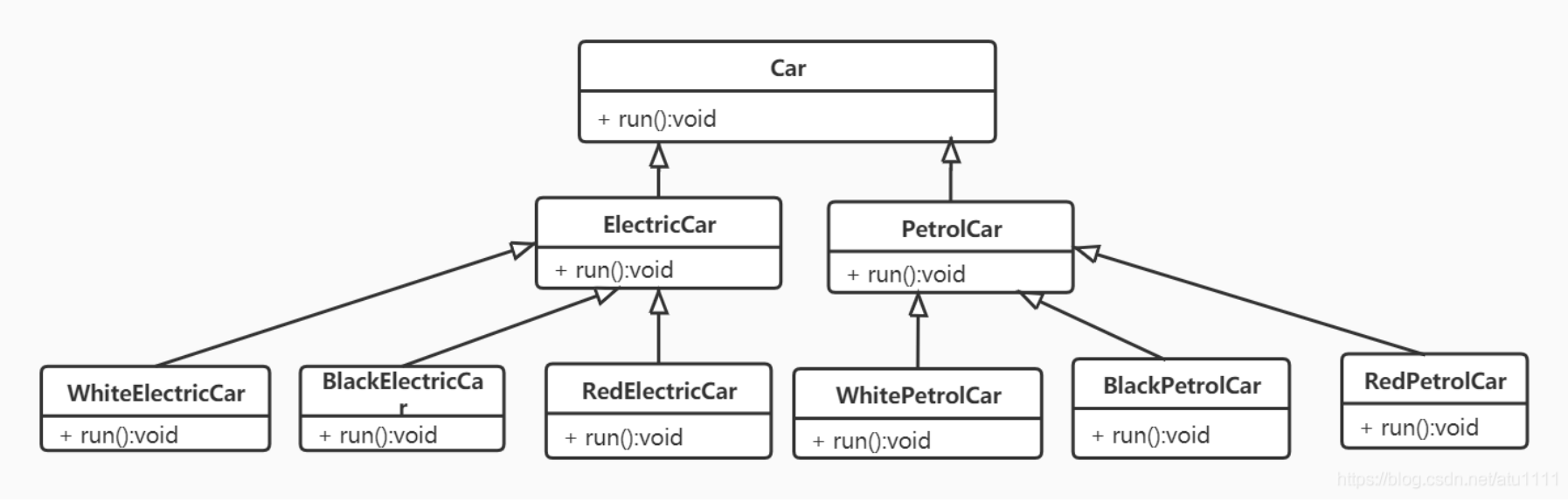
继承与委派

- 继承和委派常同时使用，来实现特定的设计目标
- 继承发生在“类”层面，委派发生在“对象”层面
  - 如果子类只需要复用父类中的一小部分方法，可以不需要使用继承，而是通过委派机制来实现
    - 可避免大量无用的方法
- 合成复用原则
  - 优先使用委派而不是继承来实现复用
  - 原因：降低类与类之间的耦合度
    - 通过继承来进行复用的主要问题在于继承复用会破坏系统的封装性，因为继承会将父类的实现细节暴露给子类
  - 实践方法：通过抽象类或接口来实现委派，兼具可复用性和可扩展性

继承与委派

- 例：汽车按“动力源”划分可分为汽油汽车、电动汽车等；按“颜色”划分可分为白色汽车、黑色汽车和红色汽车等。如果同时考虑这两种分类，其组合就有6种。

解法1：采用继承实现复用

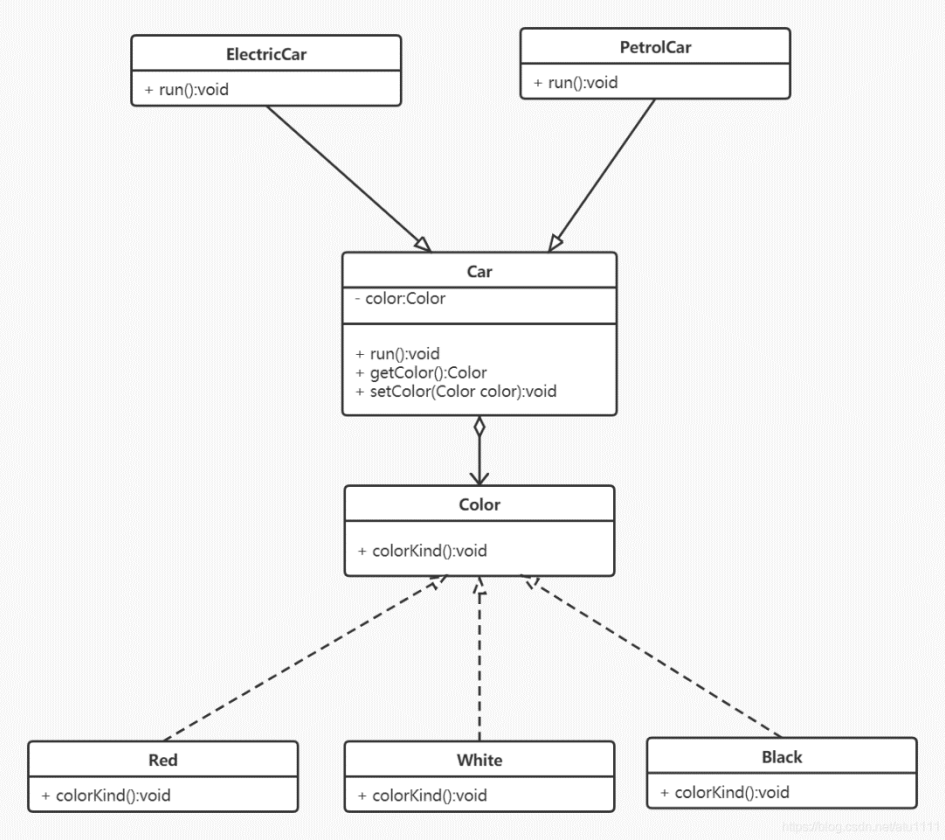


问题：这种实现方式会导致子类过多的情况，并且当增加新的“动力源”或者增加新的“颜色”都要修改源代码，这违背了开闭原则

继承与委派

- 例：汽车按“动力源”划分可分为汽油汽车、电动汽车等；按“颜色”划分可分为白色汽车、黑色汽车和红色汽车等。如果同时考虑这两种分类，其组合就有6种。

解法2：运用合成复用原则

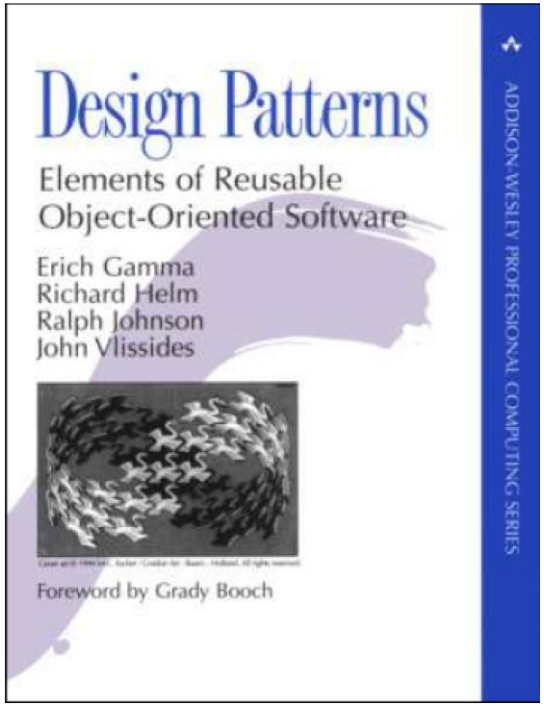


本讲概览

- 可复用性
  - 复用的形态
- 设计可复用的类
  - 继承
  - 委托
- **面向复用的设计模式**
  - **适配器模式、装饰器模式、外观模式**
  - **策略模式、模板方法模式**

软件设计模式

- 广义讲，软件设计模式是可解决一类软件问题并能重复使用的软件设计方案
- 狭义讲，软件设计模式是对被用来在特定场景下解决一般设计问题的类和相互通信的对象的描述
  - 是在类和对象的层次描述的、可重复使用的软件设计问题的解决方案
- 最初由GoF (Gang of Four)在1990年从建筑设计领域引入到计算机领域
  - Erich Gamma
  - Richard Helm
  - Ralph Johnson
  - John Vlissides



软件设计模式

- GoF总结了三大类、23种设计模式
- 创建型模式（Creational Patterns）5种
  - 关注对象的创建，提供在创建对象的同时隐藏创建逻辑的方式，可以更加灵活地创建对象
  - 工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式
- 结构型模式（Structural Patterns）7种
  - 关注类和对象的组合，通过改变代码结构来达到解耦的目的，使得代码容易维护和扩展
  - 适配器模式、装饰器模式、外观模式、代理模式、桥接模式、组合模式、享元模式
- 行为型模式（Behavioral Patterns）11种
  - 关注类之间的相互关系，将职责划分清楚，使得代码更加清晰
  - 策略模式、模板方法模式、观察者模式、访问者模式、状态模式、备忘录模式、迭代器模式、责任链模式、命令模式、中介者模式、解释器模式

思考：

- 假设已经实现了双向队列类Deque，支持在队列前后两端的高效插入与删除

```
class Deque {
public:
    void push_back(int x) { cout<<"Deque push_back"<<endl; }
    void push_front(int x) { cout<<"Deque push_front"<<endl; }
    void pop_back() { cout<<"Deque pop_back"<<endl; }
    void pop_front() { cout<<"Deque pop_front"<<endl; }
};
```

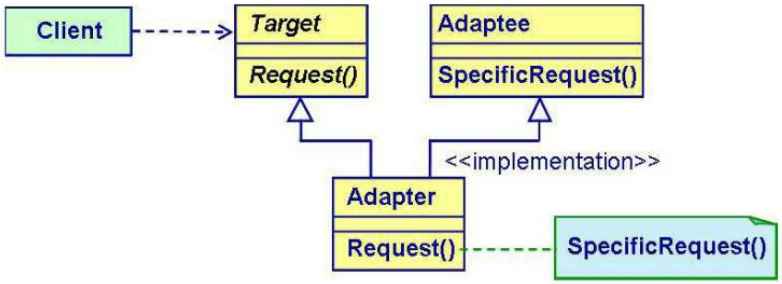
- 如果想复用Deque类实现栈和队列，如何做？
  - 队列使用Deque的后端插入、前端删除
  - 栈使用Deque的后端插入，后端删除

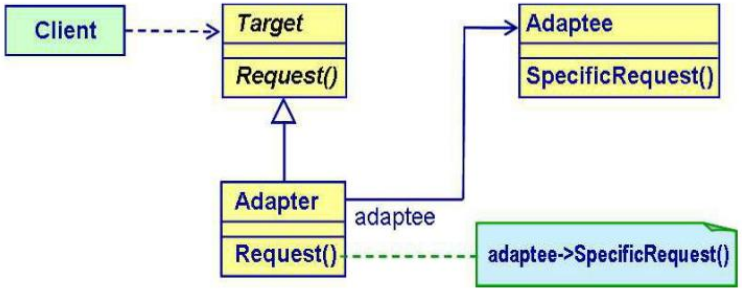
适配器模式

- 适配器（Adaptor）模式将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的类可以一起工作

- 类适配器
  - 用一个具体的Adapter类对Adaptee和Target进行适配，Adapter类多重继承自Adaptee和Target类
  - Adapter可重定义Adaptee的部分行为

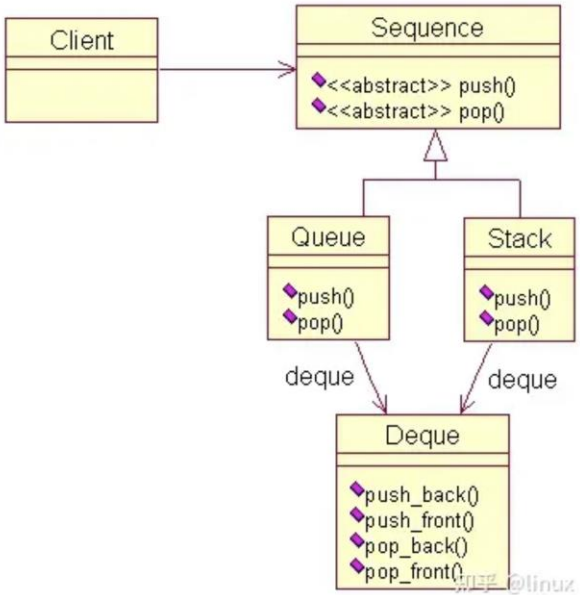
- 对象适配器
  - 使用组合或聚合将Adaptee的对象添加到Adapter中，允许一个Adapter与多个Adaptee同时工作
  - 不仅可以适配某个类，也可以适配该类的任何子类





适配器模式

❑ 例：将双向队列Deque适配为Queue和Stack




```
int main() {
    Sequence *s1 = new Stack();
    Sequence *s2 = new Queue();
    s1->push(1); s1->pop();
    s2->push(1); s2->pop();
    delete s1; delete s2;
    return 0;
}
```

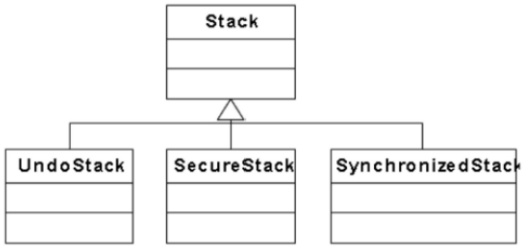
```
class Sequence {
public:
    virtual void push(int x) = 0;
    virtual void pop() = 0;
};

class Stack: public Sequence {
private:
    Deque deque; //双端队列
public:
    void push(int x) { deque.push_back(x); }
    void pop() { deque.pop_back(); }
};

class Queue: public Sequence {
private:
    Deque deque; //双端队列
public:
    void push(int x) { deque.push_back(x); }
    void pop() { deque.pop_front(); }
};
```

- 适配器模式的使用过程
  - 客户通过目标接口调用适配器的方法对适配器发出请求
  - 适配器使用被适配者接口把请求转换成被适配者的一个或者多个调用接口
  - 客户接收到调用的结果，但并未察觉这一切是适配器在起转换作用
- 优点
  - 方便设计者自由定义接口，不用担心匹配问题
- 缺点
  - 属于静态结构，不适用于多种不同的源适配到同一个目标（如支持多继承可解决此问题）

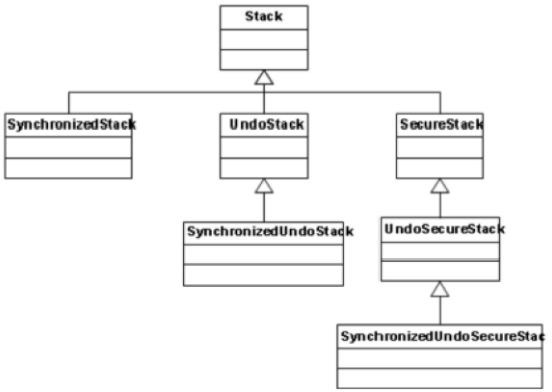
装饰器模式

❑ 思考：

- ➢ 假设已经实现了栈类Stack，希望在此基础上增加功能
  - UndoStack：可以撤销前一步操作的栈
  - SecureStack：需要输入密码才能操作的栈
  - SynchronizedStack：支持将并发访问转化为序列访问的栈
- ➢ 基本思路：通过继承，每个子类增加对应的特性



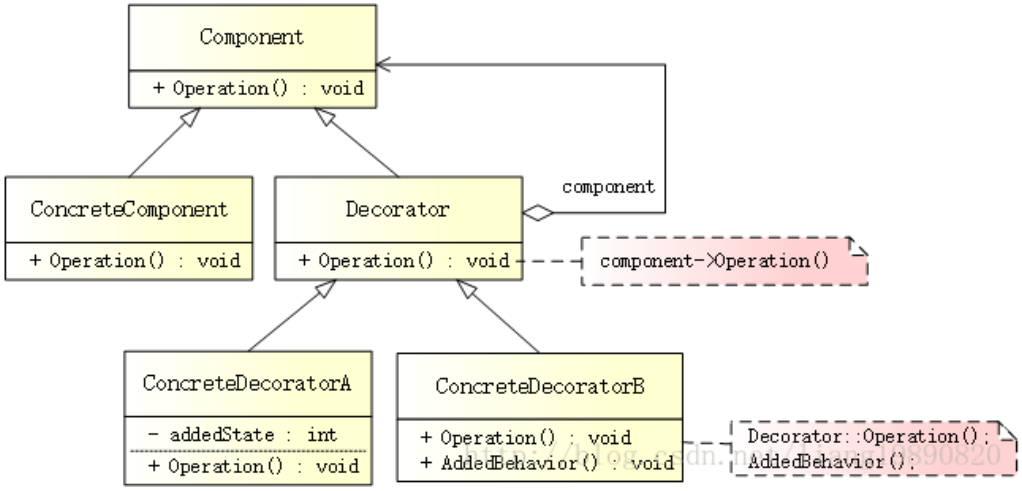
- ➢ 如果需要特性的任意组合该怎么做？
  - SecureUndoStack
  - SynchronizedUndoStack
  - SecureSynchronizedStack
  - SecureSynchronizedUndoStack



组合爆炸！大量重复代码

装饰器模式

- 解决思路：对每一个特性构造子类，通过委派机制增加到对象上
- 装饰器（Decorator）模式动态地给一个对象添加一些额外的职责



Component

- 对象接口：可以给对象动态地添加职责

ConcreteComponent

- 具体对象

Decorator

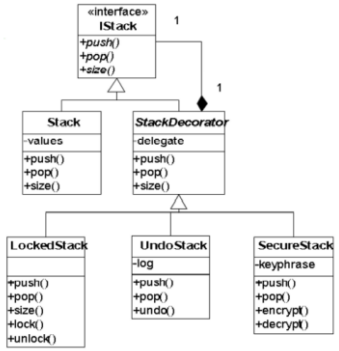
- 维持一个指向Component对象的指针，并定义一个与Component接口一致的接口

ConcreteDecorator

- 添加职责

Decorator将请求转发给它的Component对象，并有可能在转发请求前后执行一些附加的动作

例：Stack装饰器




```
class IStack {
public:
    virtual void push(Item e) = 0;
    virtual Item pop() = 0;
};
```

```
class ArrayStack : public IStack {
    std::vector<Item> items;
public:
    ArrayStack() {...}
    virtual void push(Item e) {
        ...
    }
    virtual Item pop() {
        ...
    }
};
```

```
class StackDecorator : public IStack {
protected:
    IStack* stack; //被装饰对象
public:
    StackDecorator(IStack* s) : stack(s) {}
    virtual void push(Item e) {
        stack->push(e);
    }
    virtual Item pop() {
        return stack->pop();
    }
};
```

```
class UndoStack : public StackDecorator {
private:
    std::vector<UndoLog> logs; //记录操作日志
public:
    UndoStack(IStack* s) : StackDecorator(s) {}
    virtual void push(Item e) {
        logs.push_back(UndoLog(UndoLog::PUSH, e));
        StackDecorator::push(e);
    }
    void undo() {
        ...
    }
};
```

```
IStack s = new ArrayStack();
IStack t = new UndoStack(new ArrayStack());
IStack t = new SecureStack(new SynchronizedStack(new UndoStack(s))
```

适用场景

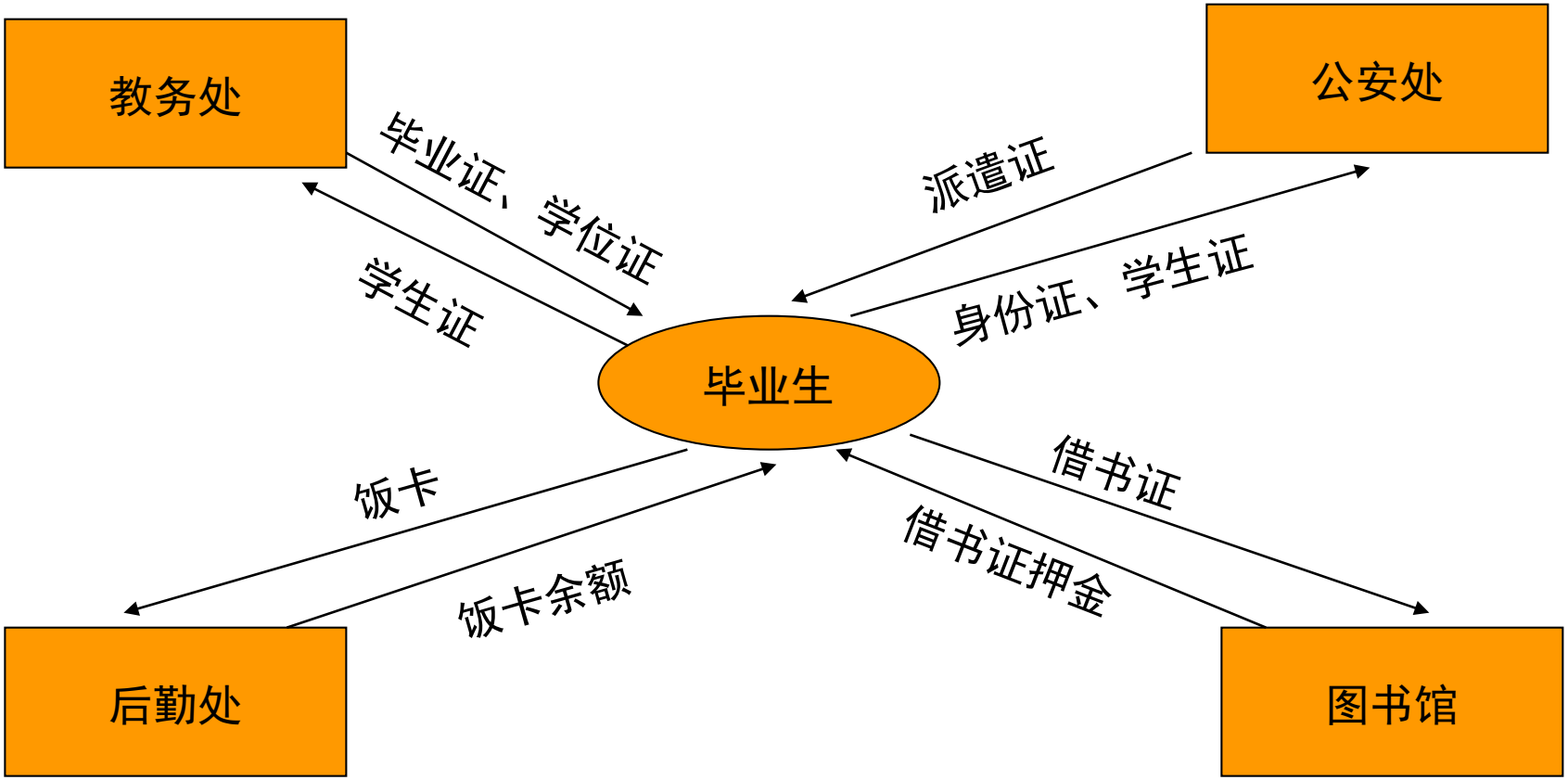
- 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责
- 当不能采用生成子类的方法进行复用时。一种情况是，可能有大量独立扩展，每一种组合将产生大量的子类，使得子类数目呈爆炸性增长。另一种情况是因为类定义被隐藏，或类定义不能用于生成子类

优缺点

- 使用Decorator模式可以很容易地向对象添加职责。可以用添加和分离的方法，在运行时添加和删除职责
- 使用Decorator模式可以很容易地重复添加一个特性，而两次继承则极容易出错
- 避免在层次结构高层的类有太多的特征：可以从简单的部件组合出复杂的功能。具有低依赖性和低复杂性
- 缺点是产生了许多小对象

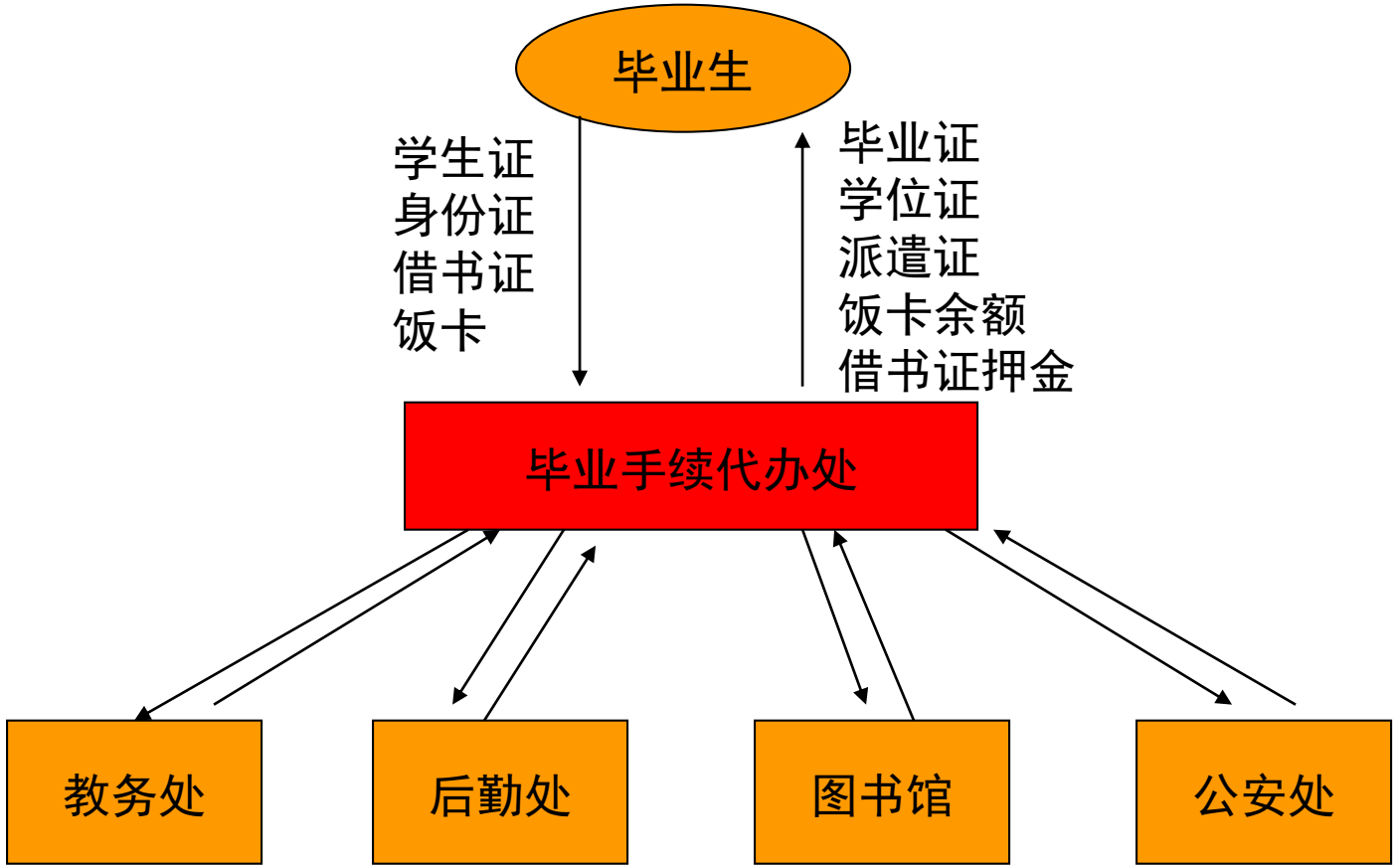
外观模式

- 思考：毕业生办毕业手续
  - 如何简化？



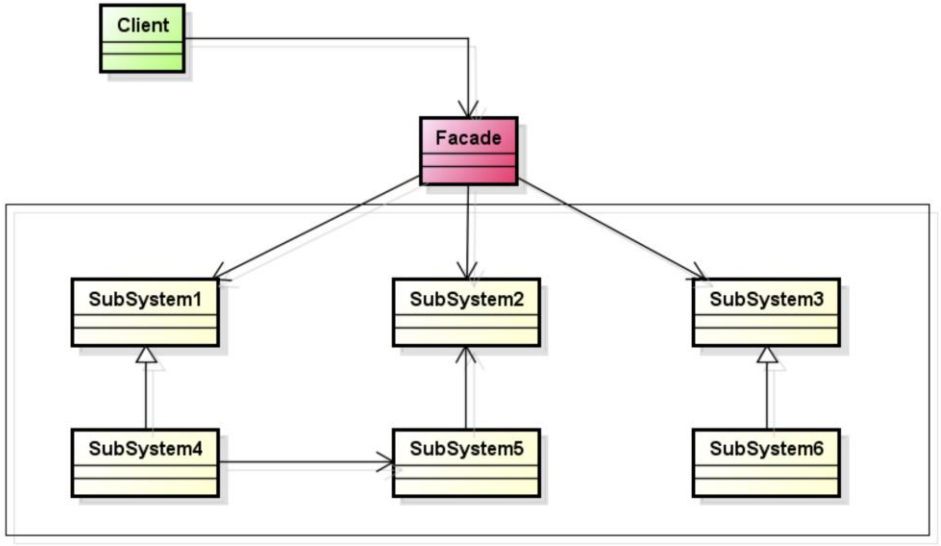
外观模式

- 思考：毕业生办毕业手续
  - 简化办法：通过一个代办处统一处理



外观模式

- 外观（Facade）模式提供一个统一的接口，用来访问子系统中的一群接口
  - 外观定义了一个高层接口，让子系统更容易使用
  - 解除客户程序与抽象类具体实现部分的依赖性，有利于移植和更改
  - 外观模式的本质是让接口变得更简单



Façade
- 知道哪些子系统类负责处理请求
- 将客户的请求代理给适当的子系统对象

Subsystem Classes
- 实现子系统的功能
- 处理由Façade对象指派的任务
- 没有Façade的任何相关信息

例：

- 假如有4个子系统的类，在不同场景下需要按不同的顺序调用其中几个类的成员函数，如何简化？

```
class SubSystemOne {
public:
    void MethodOne() {
        cout << "子系统方法一" << endl;
    }
};

class SubSystemTwo {
public:
    void MethodTwo() {
        cout << "子系统方法二" << endl;
    }
};

class SubSystemThree {
public:
    void MethodThree() {
        cout << "子系统方法三" << endl;
    }
};

class SubSystemFour {
public:
    void MethodFour() {
        cout << "子系统方法四" << endl;
    }
};
```

```
class Facade {
    SubSystemOne* one;
    SubSystemTwo* two;
    SubSystemThree* three;
    SubSystemFour* four;
public:
    void MathodA() {
        cout << "方法组A()----" << endl;
        one->MethodOne();
        two->MethodTwo();
        four->MethodFour();
    }
    void MathodB() {
        cout << "方法组B()----" << endl;
        two->MethodTwo();
        three->MethodThree();
    }
};
```

```
Facade* facade = new Facade;
facade->MathodA();
facade->MathodB();
```

外观模式

使用效果

- 对客户端屏蔽子系统组件，减少客户端使用对象数目
- 实现了子系统与客户之间松耦合的关系，使得子系统组件的变化不会影响到客户
- 不限制客户应用子系统类

策略模式

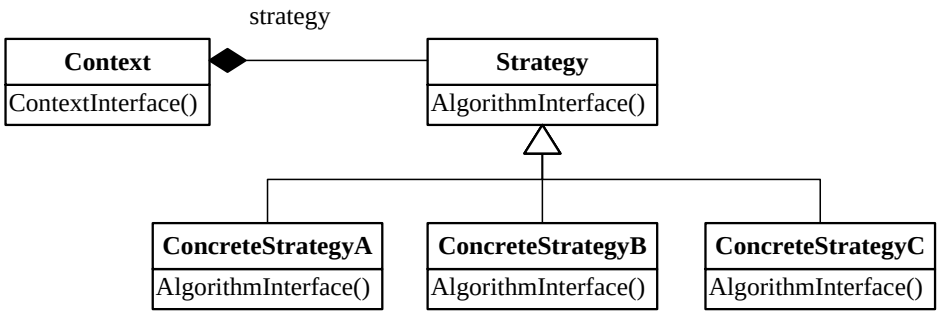
- 思考：有多种不同的算法来实现同一个任务，但需要算法使用者根据需要动态切换算法，而不是写死在代码里，如何实现灵活替换？
  - 画一个图形，可选用红色笔来画，还是绿色笔来画，或者蓝色笔来画

```
class Context {
public:
    void executeDraw(int radius, int x, int y, int color) {
        if (color == 1)
            cout << "用红色笔画图 radius:" << radius << ", x:" << x << ", y:" << y << endl;
        else if (color == 2)
            cout << "用绿色笔画图 radius:" << radius << ", x:" << x << ", y:" << y << endl;
        else if (color == 3)
            cout << "用蓝色笔画图 radius:" << radius << ", x:" << x << ", y:" << y << endl;
    }
};
```

- 使用算法的类复杂而难于维护，尤其当需要支持多种算法且每种算法都很复杂时问题会更加严重
- 算法的实现和使用算法的对象紧紧耦合在一起，使新增算法或修改算法变得十分困难，系统应对变化的能力很差

策略模式

- 策略（Strategy）模式是指定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换
  - 使得算法可独立于使用它的客户而变化
  - 适用场景：某些对象使用的算法可能多种多样，经常改变，如果将这些算法都编码到对象中，将会使得对象变得异常复杂；而且有时候支持不同的算法也是一个性能负担



Strategy
- 定义所有支持算法的公共接口
- Context使用该接口调用某ConcreteStrategy定义的算法

ConcreteStrategy
- 以Strategy接口实现某具体算法

Context
- 用一个ConcreteStrategy对象来配置
- 维护一个对Strategy对象的引用
- 可定义一个接口来让Strategy访问它的数据

❑ 例：不同颜色的画笔

```
class Strategy {
public:
    virtual void draw(int radius, int x, int y) = 0;
};

class RedPen : public Strategy {
public:
    void draw(int radius, int x, int y) {
        std::cout << "用红色笔画图 radius:" << radius
<< ", x:" << x << ", y:" << y << std::endl;
    }
};

class GreenPen : public Strategy {
public:
    void draw(int radius, int x, int y) {
        std::cout << "用绿色笔画图 radius:" << radius
<< ", x:" << x << ", y:" << y << std::endl;
    }
};

class BluePen : public Strategy {
public:
    void draw(int radius, int x, int y) {
        std::cout << "用蓝色笔画图 radius:" << radius
<< ", x:" << x << ", y:" << y << std::endl;
    }
};
```

```
class Context {
private:
    Strategy* strategy;
public:
    Context(Strategy* strategy) :
strategy(strategy) {}

    void setStrategy(Strategy* strategy) {
        this->strategy = strategy;
    }

    void executeDraw(int radius, int x, int y) {
        strategy->draw(radius, x, y);
    }
};
```

```
int main() {
    Context context(new BluePen());
    context.executeDraw(10, 0, 0);

    return 0;
}
```

优点

- 算法和使用算法的对象相互分离，客户程序可以在运行时动态选择算法，代码复用性好，便于修改和维护
- 消除了冗长的条件语句序列
- 提供相同行为的不同实现，客户可以根据不同的上下文从不同的策略中选择算法

缺点

- 客户必须了解不同的Strategy
- Strategy和Context之间的通信开销加大
- 增加了类和对象数目，在算法较多时更加严重

❑ 回顾：求和

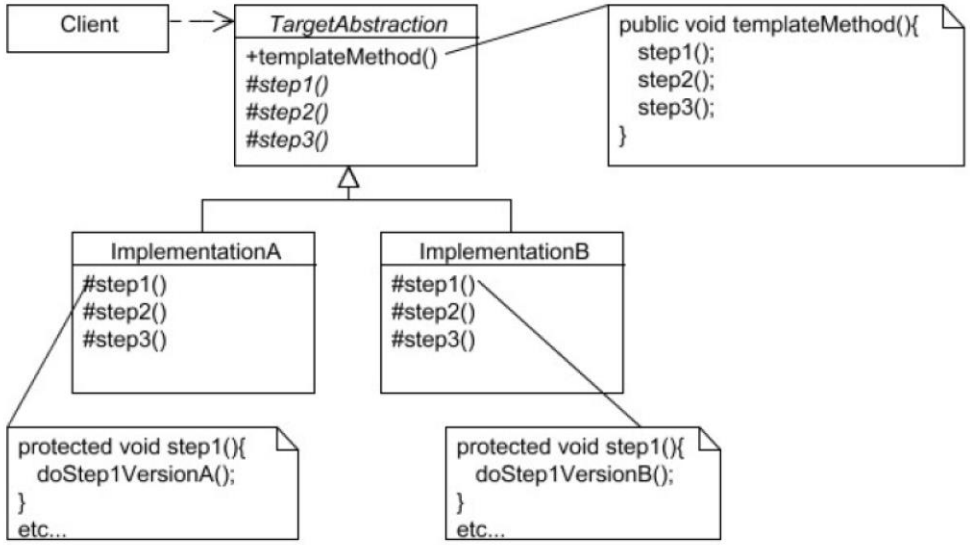
$$\sum_{k=1}^{n} k \qquad \sum_{k=1}^{n} k^3 \qquad \sum_{k=1}^{n} (4k-3)(2k+5)$$

```
int sum_naturals(int n) {
    int total = 0, k = 1;
    while (k <= n) {
        total = total + k; k++;
    }
    return total;
}
int sum_cubes(int n) {
    int total = 0, k = 1;
    while (k <= n) {
        total = total + k*k*k; k++;
    }
    return total;
}
int sum_poly(int n) {
    int total = 0, k = 1;
    while (k <= n) {
        total = total + (4*k-3)*(2*k+5); k++;
    }
    return total;
}
```

```
int summation(int n, int (*term)(int)) {
    int total = 0, k = 1;
    while (k <= n) {
        total = total + term(k); k++;
    }
    return total;
}

int natural(int x)
{
    return x;
}

int sum_naturals(int n) {
    return summation(n,
natural);
}

int cube(int x) {
    return x*x*x;
}

int sum_cubes(int n) {
    return summation(n, cube);
}

int poly(int x) {
    return (4*x-
3)*(2*x+5);
}

int sum_poly(int n) {
    return summation(n, poly);
}
```

模板方法模式

- 模板方法（Template Method）模式定义操作中算法的骨架，将一些步骤的执行延迟到其子类中
  - 子类不需要改变算法结构即可重定义算法的某些步骤
  - 适用场景：具有统一的操作步骤或操作过程，具有不同的操作细节，即存在多个具有同样操作步骤的应用场景，但某些具体的操作细节却各不相同



TargetAbstraction

- 定义一个“模板”(或算法骨架)。子类(使用该模板的客户)重定义模板中的操作(相当于根据模板填写内容)

Implementation

- 实现模板定义的操作以完成算法中与特定子类相关的步骤

模板方法模式

- 例：求和

```
class Summation  {
protected:
    virtual int getTerm(int) = 0;
public:
    int sum(int n) {
        int total = 0, k = 1;
        while (k <= n) {
            total = total + getTerm(k); k++;
        }
        return total;
    }
};

class Summation_Naturals: public Summation {
protected:
    int getTerm(int x) {return x;}
};

class Summation_Cube: public Summation {
protected:
    int getTerm(int x) {return x*x*x;}
};

class Summation_Poly: public Summation {
protected:
    int getTerm(int x) {return (4*x-3)*(2*x+5);}
};
```

使用效果

- 模板方法是一种代码复用技术，模板提取了子类的公共行为
- 模板方法模式形成一种反向的控制结构（依赖倒置原则）
  - 父类调用子类的操作（高层模块调用低层模块的操作），低层模块实现高层模块声明的接口
  - 这样控制权在父类（高层模块），低层模块反而要依赖高层模块
- 可通过在抽象模板定义模板方法给出成熟算法步骤，同时又不限制步骤细节，具体模板实现算法细节不会改变整个算法骨架
- 在抽象模板模式中，可以通过钩子方法对某些步骤进行挂钩，具体模板通过钩子可以选择算法骨架中的某些步骤

谢谢

欢迎在课程群里填写问卷反馈