结构型模式

(OOP)

黄民烈

aihuang@tsinghua.edu.cn

http://coai.cs.tsinghua.edu.cn/hml

课程团队：刘知远 姚海龙 黄民烈

本讲内容提要

15.1 适配器（Adapter）模式

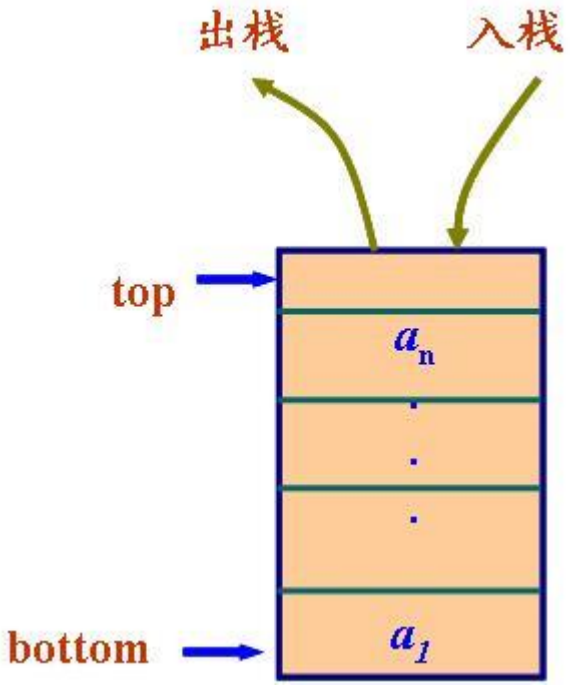
15.2 代理/委托（Proxy）模式

15.3 装饰器（Decorator）模式

15.4 设计模式总结

一个简单例子—栈

- 功能类似数组
- 元素访问规则有所不同，是“后进先出”（Last-In-First-Out）
- 简单起见，只支持int类型的元素



代码实现

```
#include <cstring>
#include <cstdio>
#include <vector>
#include <iostream>

//堆栈基类
class Stack {
public:
    virtual ~Stack() { }
    virtual   bool full() = 0;
    virtual   bool empty() = 0;
    virtual   void push(int i) = 0;
    virtual   void pop() = 0;
    virtual   int size() = 0;
    virtual   int top() = 0;
}
```

简单实现

```
class MyStack : public Stack{

private:
    int *m_data; const int m_size; int m_top;

public:
    //构造函数
    MyStack(size) : m_size(size), m_top(-1), m_data(NULL) {
        if (m_size > 0) m_data = new int[m_size];
    }
    //析构函数
    virtual ~MyStack() {
        if (m_data) delete [] m_data;
    }
    //满栈检测
    bool full() {
        return m_size <= 0 || (m_top+1) == m_size;
    }
```

简单实现

```
    //空栈检测
    bool empty() {
        return m_top < 0;
    }
    //入栈
    void push(int i) {
        if (m_top+1 < m_size) m_data[++ m_top] = i;
    }
    //出栈
    void pop() { if (!empty()) --m_top;  }
    //获取堆栈已用空间
    int size() { return m_top+1; }
    //获取栈头内容
    int top() {
        if (!empty())
            return m_data[m_top];
        else
            return INT_MIN;
    }
};
```

简单实现

```
int main() {

    //创建一个最多放置10个元素的栈
    MyStack stack(10);

    //压入1,2,3,4
    for (int i = 1; i < 5; i++)
        stack.push(i);

    //逐个弹出
    for (int i = 0; i < 4; i++) {
        cout << stack.top() << "\n";
        stack.pop();
    }

    return 0;
}
```

```
4
3
2
1
```

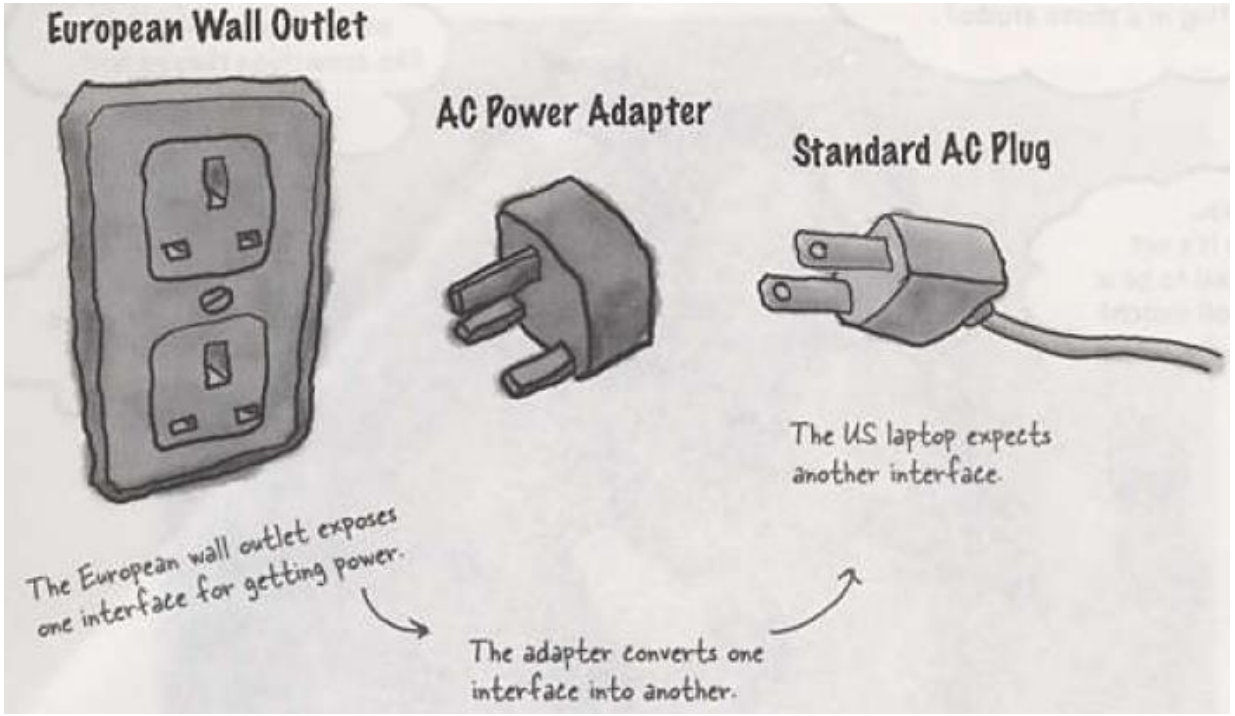
有没有更简单的实现方式?

STL vector

- 工作量太大(OOP思想之一：复用)

- STL中已经有vector这个容器
- vector提供了如下方法:
  - push_back()
  - size()
  - back()
  - pop_back()

分析



- Vector
  - 功能上满足要求（内存管理，元素插入弹出）
  - 但是接口不一致
- 需要进行接口的“转换”

适配器

- 考虑生活中一种常见的情况：
  - 有手机、手机充电线，要给手机充电。
  - 充电线只能插在USB接口上进行充电。
  - 但是现在只有220V的插座可以供电。
  - 所以需要用一个转接头将220V插座和USB接口衔接。

- 这里的转接头实际上就是一种现实中的适配器

适配器
Adapter

适配器

概述

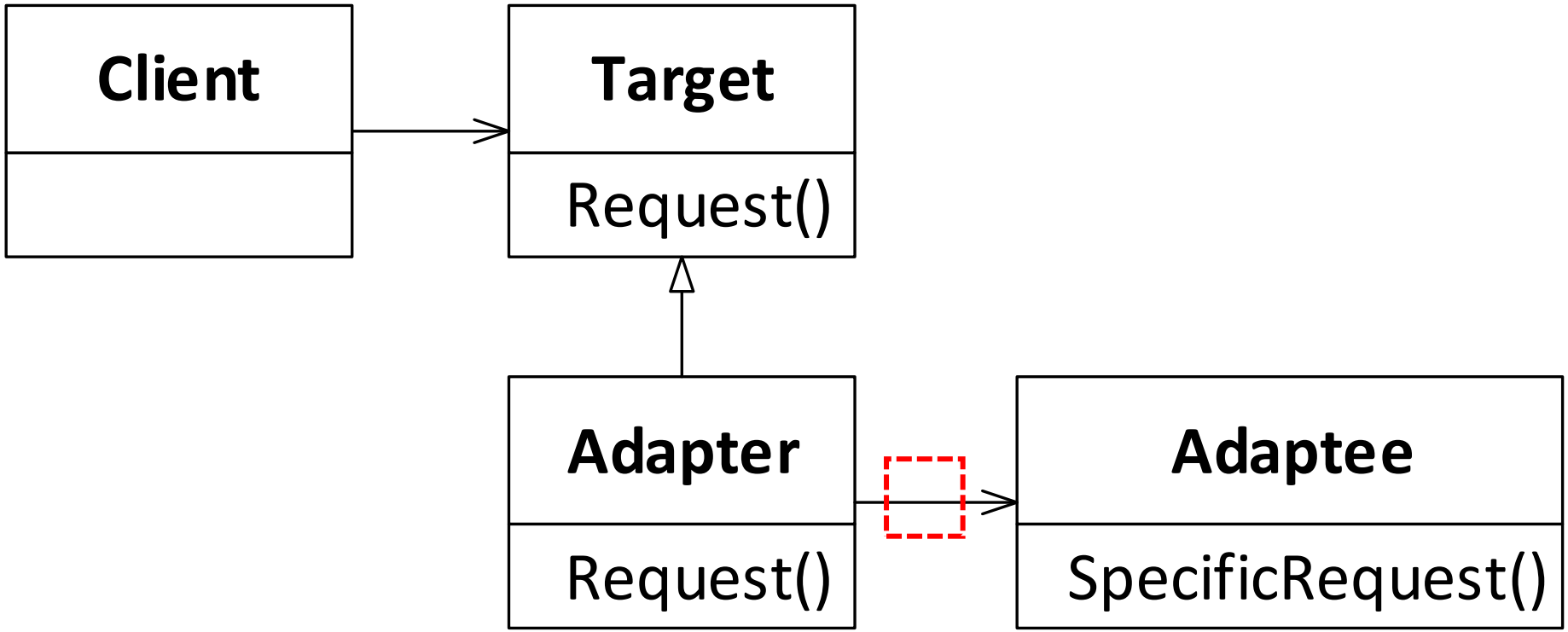
- 适配器模式将一个类的接口转换成客户希望的另一个接口，从而使得原本由于接口不兼容而不能一起工作的类可以在统一的接口环境下工作。

从adapter到adaptee的转换

结构

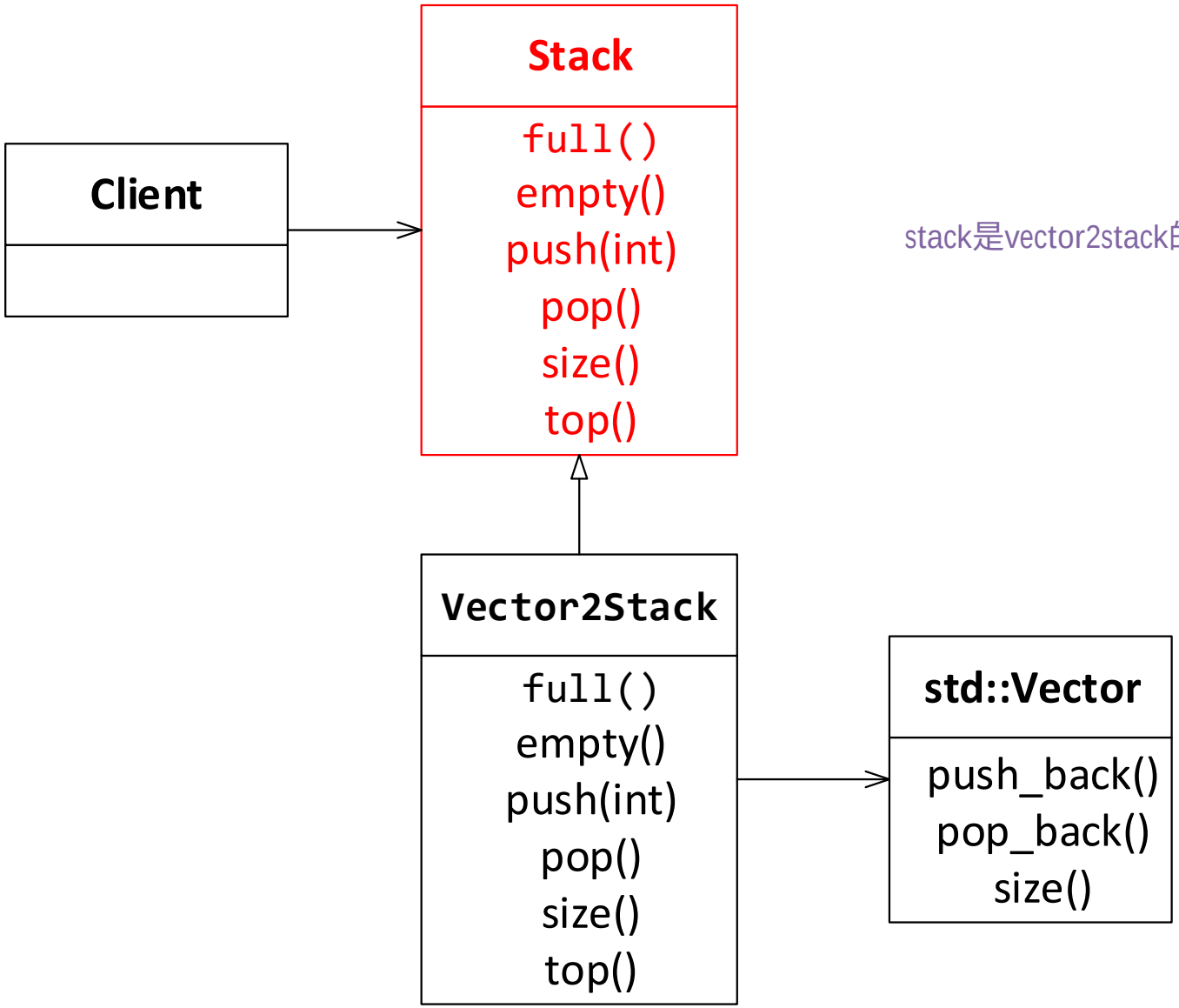
- 目标（Target）：客户所期待的接口。
- 需要适配的类（Adaptee）：需要适配的类。
- 适配器（Adapter）：通过包装一个需要适配的类，把原接口转换成目标接口。

适配器——实现一



使用组合实现适配，称作对象适配器模式

适配器基类定义

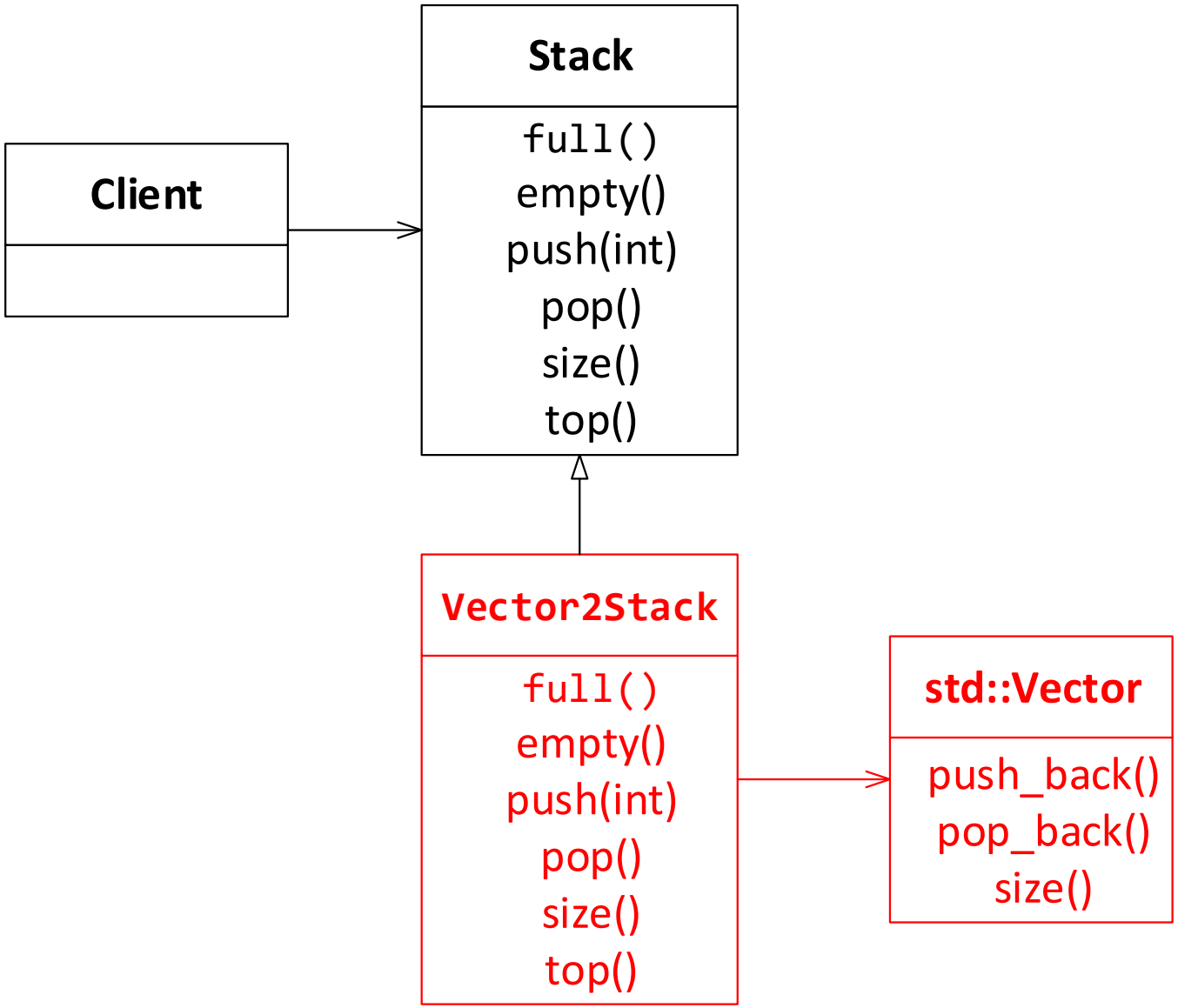


stack是vector2stack的派生类

适配器——实现一

```
#include <cstring>
#include <cstdio>
#include <vector>
#include <iostream>

//堆栈基类
class Stack {
public:
    virtual ~Stack() { }
    virtual   bool full() = 0;
    virtual   bool empty() = 0;
    virtual   void push(int i) = 0;
    virtual   void pop() = 0;
    virtual   int size() = 0;
    virtual   int top() = 0;
}
```

组合方式实现适配器模式



适配器——实现一

```
class Vector2Stack : public Stack{
private:
    std::vector<int> m_data; //将vector的接口组合进来实现具体功能
    const int m_size;
public:
    Vector2Stack(int size) : m_size(size) { }
    bool full() { return (int)m_data.size() >= m_size; }   //满栈检测
    bool empty() { return (int)m_data.size() == 0; }   //空栈检测
    void push(int i) { m_data.push_back(i); }       //入栈
    void pop() { if (!empty()) m_data.pop_back(); }//出栈
    int size() { return m_data.size(); }            //获取堆栈已用空间
    int top() {                                     //获取栈头内容
        if (!empty())
            return m_data[m_data.size()-1];
        else
            return INT_MIN;
    }
};
```

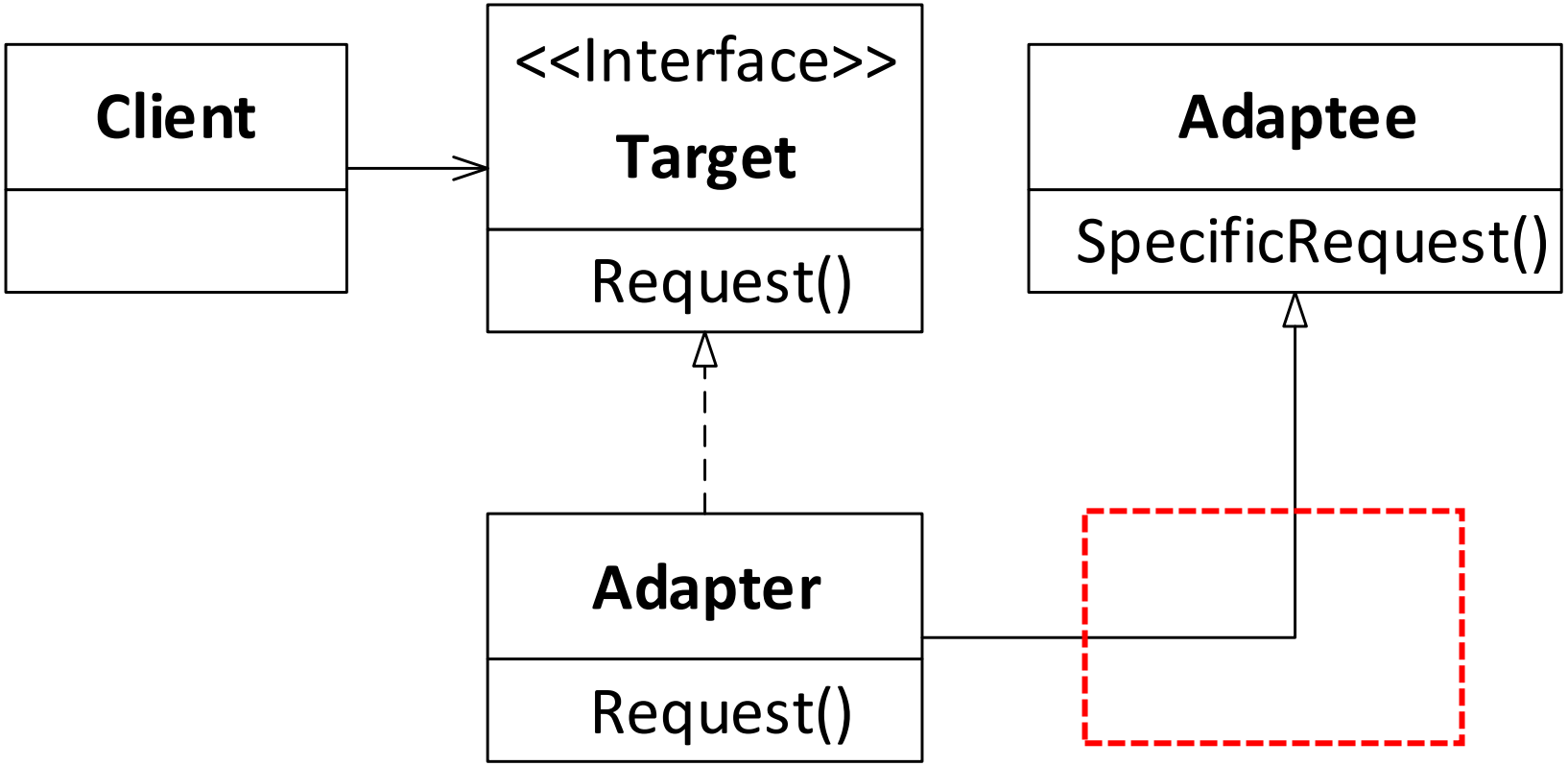
m_size是stack的最大长度

适配器——实现一

```
int main() {

    Vector2Stack stack(10);

    //压入1,2,3,4
    for (int i = 1; i < 5; i++)
        stack.push(i);

    //逐个弹出
    for (int i = 0; i < 4; i++) {
        std:: cout << stack.top() << "\n";
        stack.pop();
    }

    return 0;
}
```

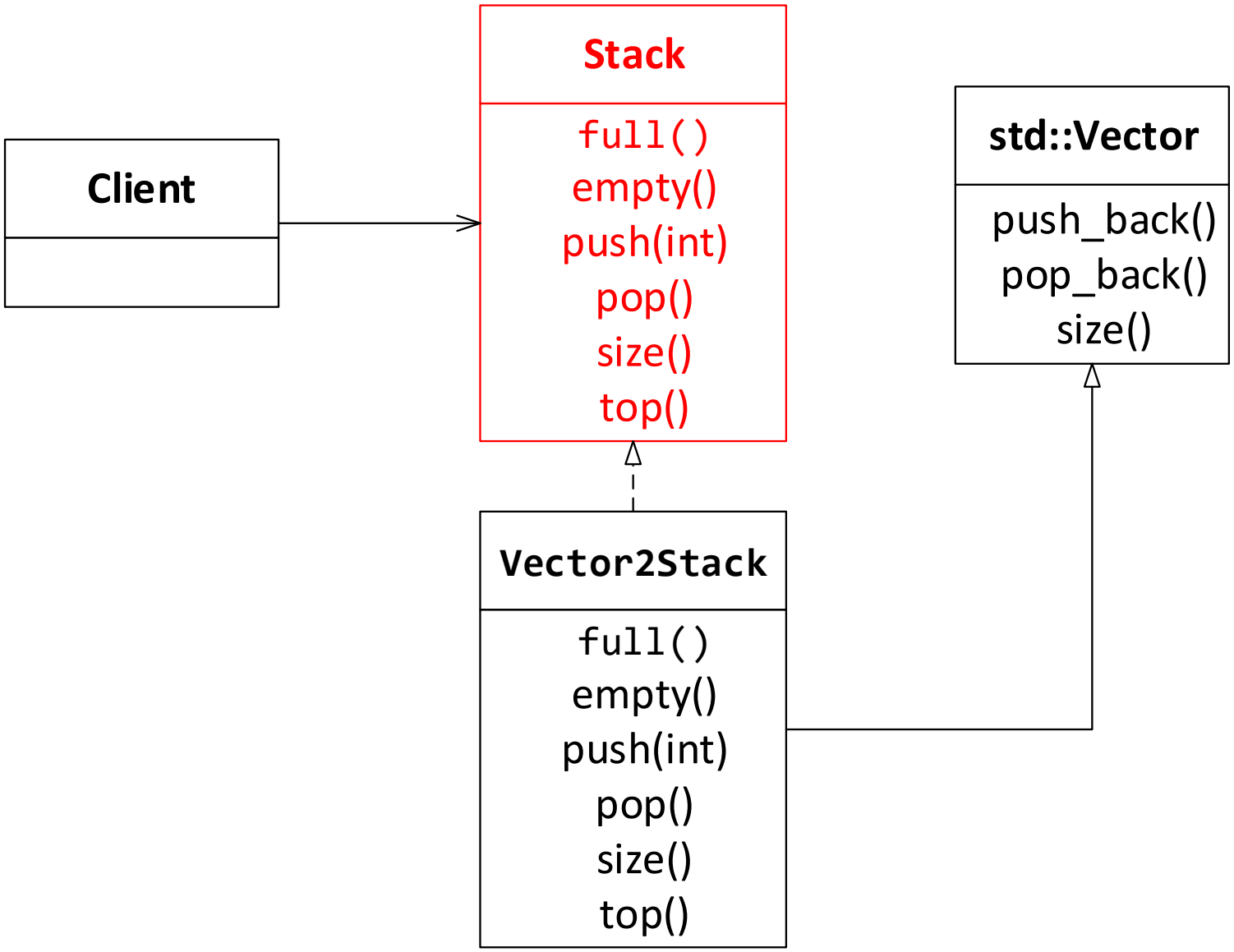
```
4
3
2
1
```

适配器——实现二



使用继承实现适配，称作类适配器模式

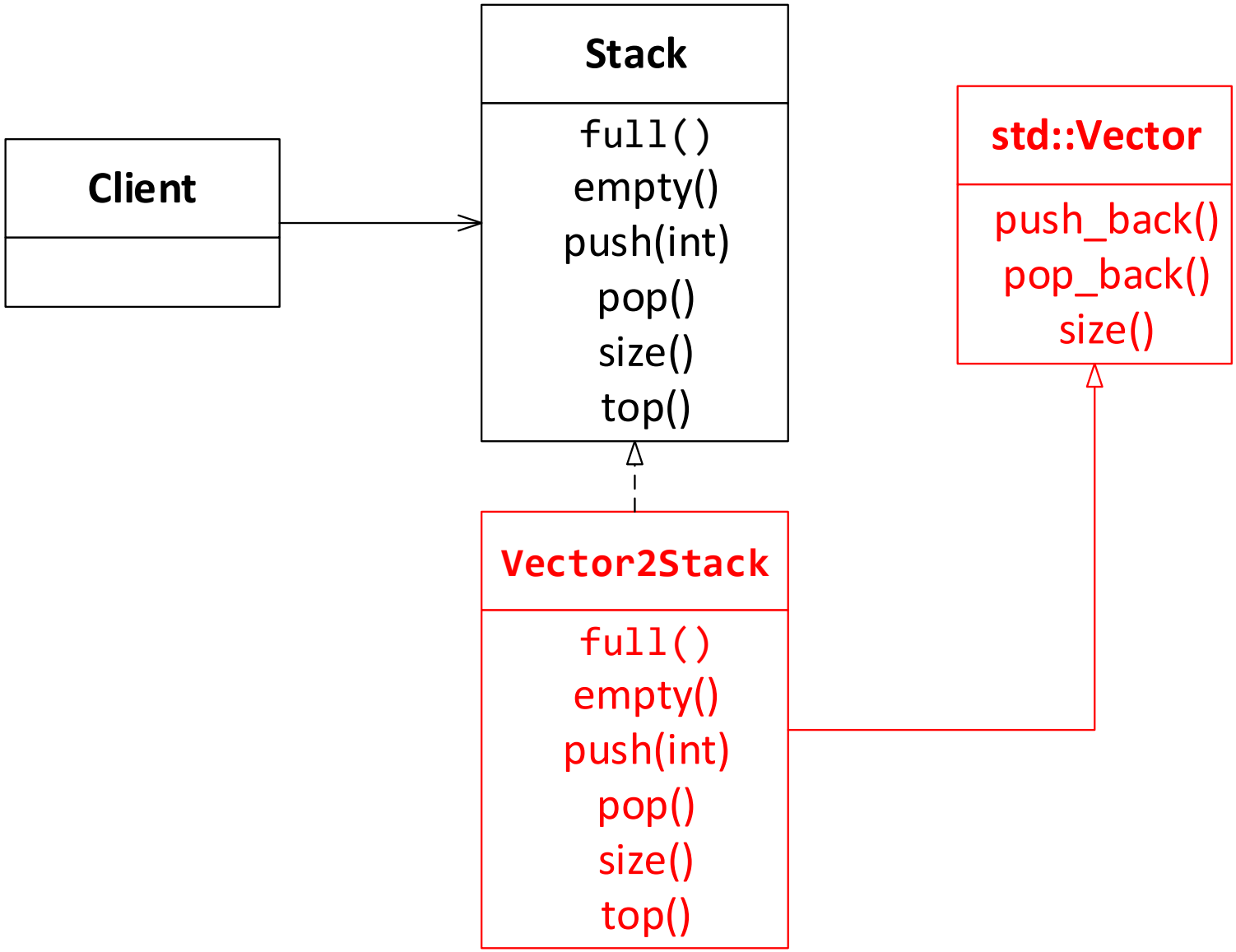
适配器接口定义


Client
Stack
full()
empty()
push(int)
pop()
size()
top()
std::Vector
push_back()
pop_back()
size()
Vector2Stack
full()
empty()
push(int)
pop()
size()
top()

适配器——实现二

```
#include <cstring>
#include <cstdio>
#include <vector>
#include <iostream>

//堆栈基类
class Stack {
public:
    virtual ~Stack() { }
    virtual   bool full() = 0;
    virtual   bool empty() = 0;
    virtual   void push(int i) = 0;
    virtual   void pop() = 0;
    virtual   int size() = 0;
    virtual   int top() = 0;
}
```

继承方式实现适配器模式



适配器——实现二

```
//直接继承vector并改造接口，采用私有继承可以使得外界只能接触到Vector2Stack中的接口
class Vector2Stack : private std::vector<int>, public Stack {
public:
    Vector2Stack(int size) : vector<int>(size) { }
    bool full() { return false; }
    bool empty() { return vector<int>::empty(); }
    void push(int i) { push_back(i); }
    void pop() { pop_back(); }
    int size() { return vector<int>::size(); }
    int top() { return back(); }
};
```

vector是私有继承，使得vector的接口在外部无法调用，只能由stack提供外部可用接口

适配器——实现二

```
int main(int argc, char *argv[]) {

    Vector2Stack stack(10);

    //压入1,2,3,4
    for (int i = 1; i < 5; i++)
        stack.push(i);

    //逐个弹出
    for (int i = 0; i < 4; i++) {
        std::cout << stack.top() << "\n";
        stack.pop();
    }

    return 0;
}
```

```
4
3
2
1
```

适配器

优点

oop是面向接口的编程，而不是面向实现的编程；方便多人协作

- 通过适配器，客户端可以用统一接口调用各种复杂的底层工作类
- 复用了现有的类，提高代码复用率
- 将目标类和适配者类解耦，通过引入一个适配器类包装现有的适配者类以满足新接口需求，无需修改原有代码

适用场景举例

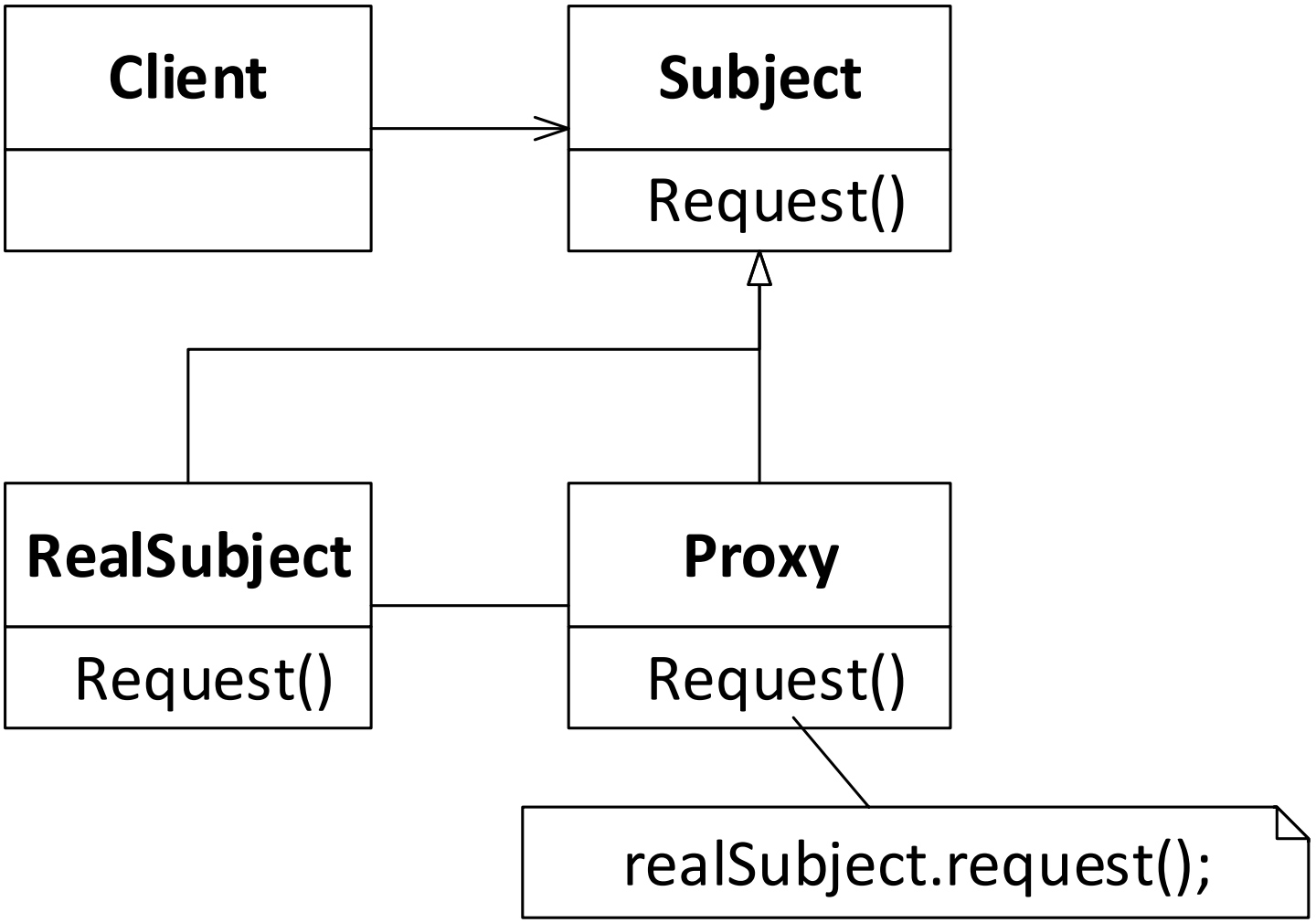
- 系统需要复用已有的类，但这些类的接口不符合系统的接口
- 接入第三方组件，但组件接口定义与自身定义不同
- 旧系统开发的类已经实现了一些功能，但是客户端只能以新接口的形式访问，且我们不希望手动更改原有类

代理/委托
Proxy

代理/委托

- 在一些应用中，直接访问对象往往会带来诸多问题
  - 对象访问前需要先预处理：
    - 对象的数据在远程机器上，访问前需要进行数据传输
  - 对象访问后需要后处理：
    - 对象访问后需要销毁
    - 访问后需要将对象数据储存到硬盘上
  - 对象访问需要更多控制：
    - 需要检查访问者权限
    - 需要记录访问过程
- 我们可以在被访问对象上加上一个访问层，在访问层上增加新的控制操作，访问层的接口保持不变，这就是代理/委托模式

代理/委托


Client
Subject
Request()
RealSubject
Request()
Proxy
Request()
realSubject.request();

代理/委托

Proxy类里面储存的指针是real subject类（而不是subject）

```
class Subject{
public:
    virtual void Request() = 0;
};
```

提供接口

```
class RealSubject:public Subject{
public:
    virtual void Request() {
        ... // request
    }
};
```

真正实现request的地方

```
class Proxy:public Subject{
private:
     RealSubject* m_realSub;

public:
    void Request() {
        ... // do something
        m_realSub ->Request();
        ... // do something
    }
};
```

override了

- **Proxy与Subject有相同的接口**
  - 调用代理类的Request接口，在调用实际接口时增加额外功能

例子：房屋中介

Proxy里面保存了一个引用，使得代理可以访问实体（也就是HouseOwner),并提供一个与HouseOwner的接口相同的接口，这样就可以用来代替实体了。
Proxy的作用就是增加一些操作！！！！！！

```
class IRentHouse {
public:
    virtual void rentHouse();
};
class HouseOwner: public IRentHouse {
public:
    void rentHouse(){
        cout << “房东:收取租金5000元” << endl;
    }
};
class IntermediaryProxy: public IRentHouse {
private:
    HouseOwner* owner;
public:
    IntermediaryProxy(HouseOwner* owner_): owner(owner_){}
    void renthouse(){
        check();
        cout << “中介:收取中介费1000元” << endl;
        owner->rentHouse();
        cout << “中介:负责维修管理” << endl;
    }
    void check() { cout << “中介:检查家具完整性” << endl; }
};
```

例子：房屋中介

```
int main(int argc, char *argv[]) {
    //声明指针
    HouseOwner* owner = new HouseOwner();
    //使用代理来包裹指针
    IntermediaryProxy proxy(owner);
    //之后的操作均通过代理进行
    proxy.rentHouse();
    delete owner;
    return 0;
}
```

中介:检查家具完整性

中介:收取中介费1000元

房东:收取租金5000元

中介:负责维修管理

“变”与“不变”

这两个类都继承了IRentHouse，都重写覆盖了rentHouse函数

- IntermediaryProxy与HouseOwner有相同的接口
  - rentHouse()
  - 对于租户来说，添加的控制层是透明的
- IntermediaryProxy比HouseOwner增加了一些控制操作
  - 预处理：检查家具完成性，先收取中介费
  - 后处理：负责维修管理
- “代理”模式
  - 接口不变，增加控制操作
  - 用于对被代理对象进行控制，如引用计数控制、权限控制、远程代理、延迟初始化等等
  - 代理类就好比被代理类的“经纪人”，一方面提供被代理类所有接口的功能，另一方面可以同时进行额外的控制操作。

代理/委托 与 适配器

- **相似：**
  - 均是在被访问对象之上进行封装
  - 均提供被封装对象的功能接口供外部使用

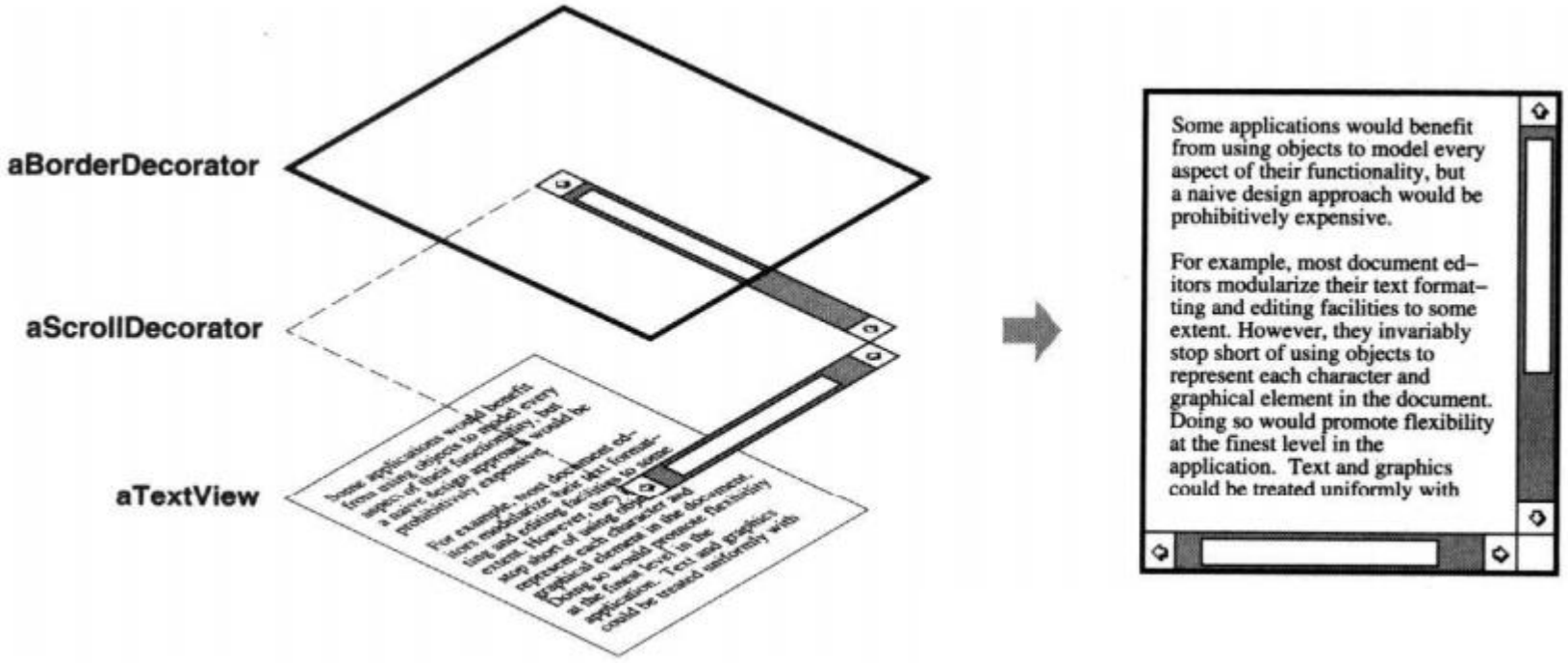
适配器的目标就是改变接口，但是适配器不会增加其他的操作（控制）

- **不同：**
  - 代理不会改变接口，但适配器可能会
  - 适配器不会增加控制，代理可能会
  - 适配器的核心要素是变换接口，代理的核心要素是分割访问对象与被访问对象以减少耦合，并能在中间增加各种控制功能。

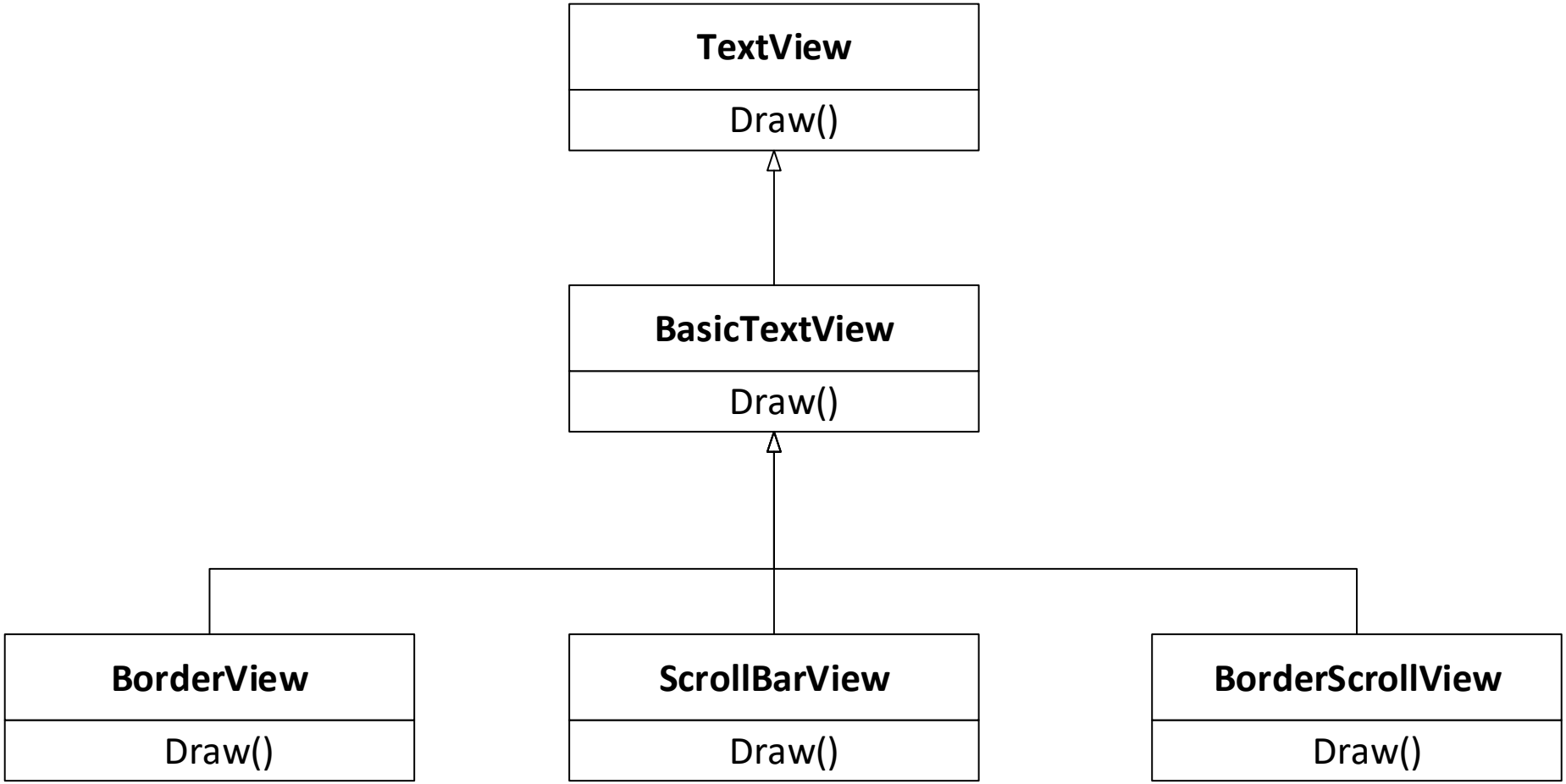
装饰器
Decorator

例子

- 有一个对象TextView，在窗口中显示文本
- 希望接口不变，增加滚动条、边框、……



继承

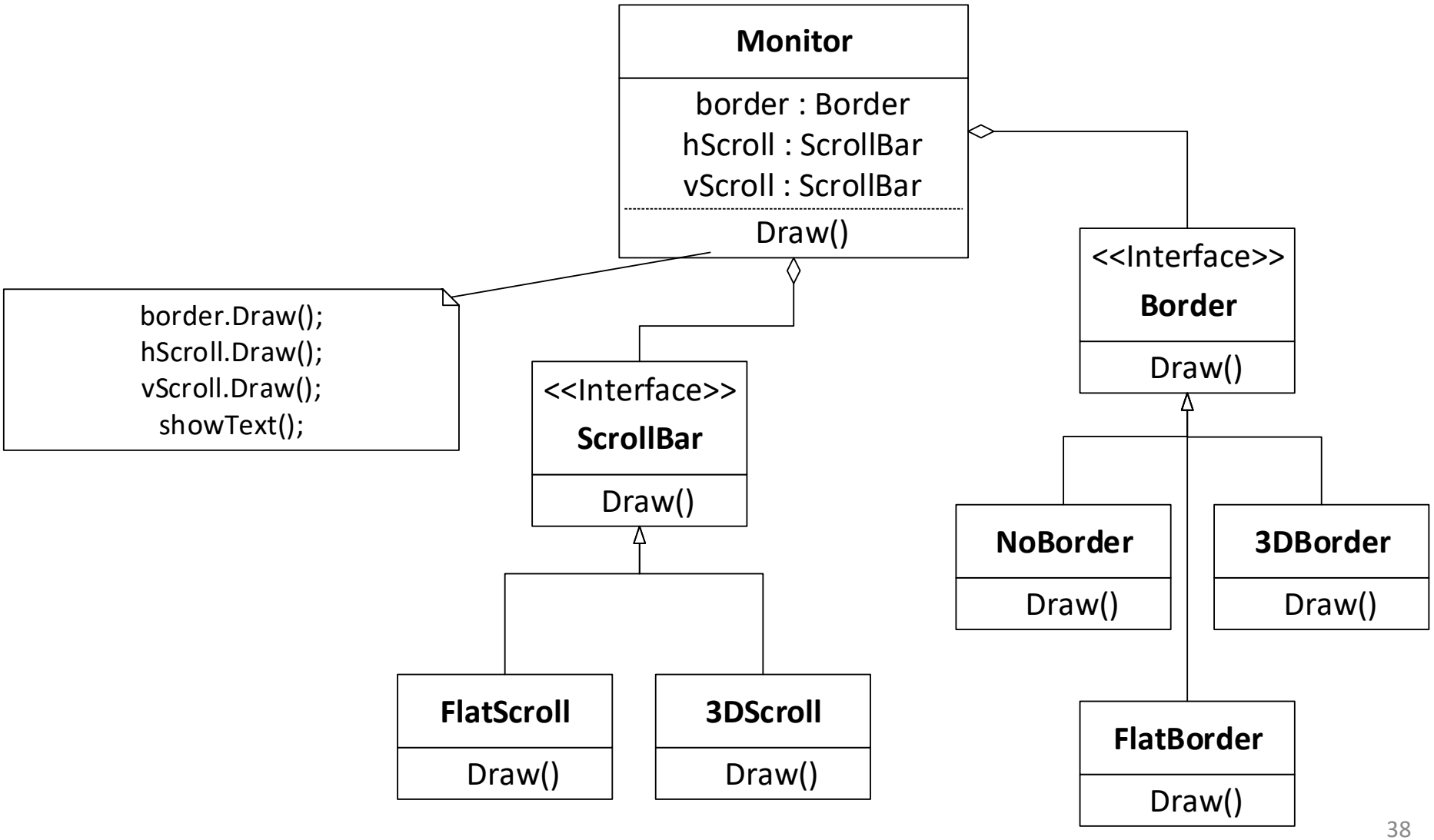

TextView
Draw()
BasicTextView
Draw()
BorderView
Draw()
ScrollBarView
Draw()
BorderScrollView
Draw()

继承

- 使用继承
- 依靠多态实现功能的变化
- 问题
  - 随着功能的变多，继承类的数量急剧膨胀，其最大派生类的数目可以是所有功能的组合数
  - 如果TextView的基类增加新的接口，那么所有的派生类都需要进行修改

策略

啥叫策略模式？就是每种策略有一个基类和若干派生类，然后我真正实现的对象里面是策略基类指针

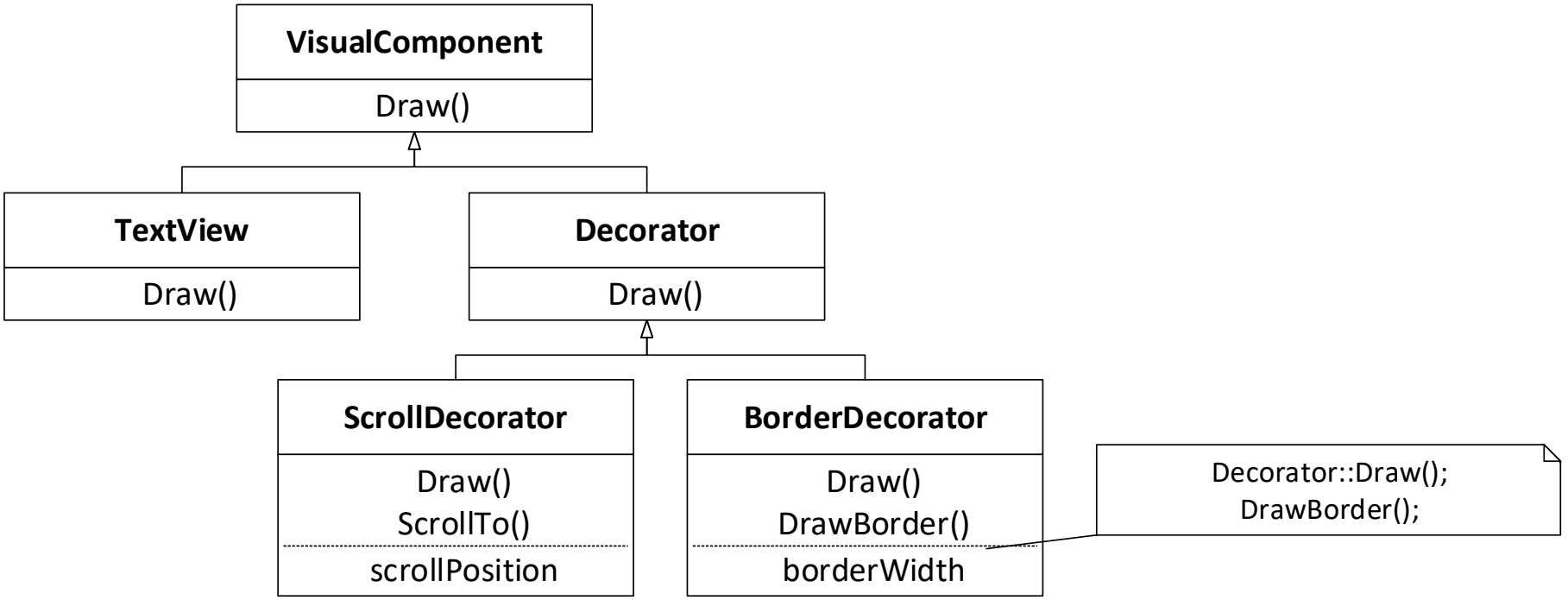


策略

- 用组合替代继承
- 问题
  - 策略的个数是基类中预先定义好的，如基类中定义了边框和滑动条，那么策略模式只能实现不同的边框与滑动条功能的组合。
  - 如果我要再增加一个滚动条和边框之外的新功能，那么就要修改基类，在基类中增加策略个数和新的方法。这样对整体框架的改动是我们不乐意见到的。

装饰器

- 创建了一个装饰类，用来包装原有的类，并在保持类方法完整性的前提下，提供了额外的功能。
- 且装饰类与被包装的类继承于同一基类，这样装饰之后的类可以被再次包装并赋予更多功能。



装饰器示例

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

//所有View的基类
class Component {
public:
    virtual ~Component() { }
    virtual void draw() = 0;
};

//一个基本的TextView类
class TextView : public Component {
public:
    void draw() {
        cout << "TextView." << endl;
    }
};
```

接口类，decorator和被包装的类都是它的派生类

装饰器示例

```
//装饰器的核心内涵在于用装饰器类整体包裹改动之前的类，以保留原来的全部接口
//在原来接口保留的基础上进行新功能扩充

class Decorator : public Component {
    //这里一个基类指针可以让Decorator能够以递归的形式不断增加新功能
    Component* _component;
public:
    Decorator(Component* component) : _component(component) {
    }
    virtual void addon() = 0;
    void draw() {
        addon();
        _component -> draw();
    }
};
```

Component是要包装的抽象类或接口；
Textview是Component的实现类，是最后装饰的实际对象；
Decorator是一个抽象类，继承或实现了Component的接口，同时它持有一个对Component实例对象的引用，也可以有自己的方法。
具体装饰(ConcreteDecorator)角色：是Decorator的实现类，是具体的装饰者对象，负责给ConcreteComponent附加责任。

代码

真正实现功能的是decorator的各种派生类，而不是decorator
通过override虚函数addon,得到不同的功能

```
//包裹原Component并扩充边框
class Border : public Decorator {
public:
    Border(Component* component) : Decorator(component) { }
    void addon() { cout << "Bordered "; }
};

//包裹原Component并扩充水平滚动条
class HScroll : public Decorator {
public:
    HScroll(Component* component): Decorator(component) { }
    void addon() { cout << "HScrolled "; }
};

//包裹原Component并扩充垂直滚动条
class VScroll : public Decorator {
public:
    VScroll(Component* component): Decorator(component) { }
    void addon() { cout << "VScrolled "; }
};
```

代码

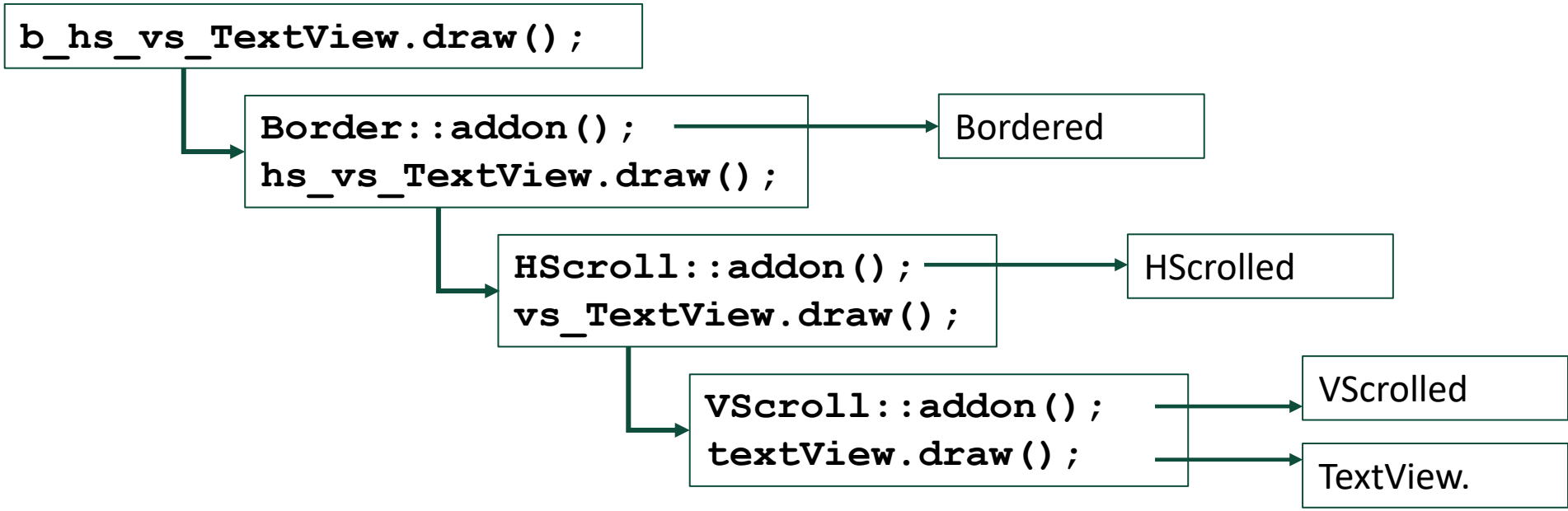
每次将新增功能后的对象作为参数传入，这时新对象调用draw（）时，调用的component->draw()各不相同，是递归调用

不断进行包装得到的仍然是decorator的派生类，只不过每多包装一次，其内部储存的component指针的实际类型就变得更复杂一些，之前包装的类不断成为之后包装的类中储存的指针，当调用该指针指向的draw函数时，draw函数也变得越来越复杂

```
int main(int argc, char** argv) {
    //基础的textView
    TextView textView;
    //在基础textView上增加滚动条
    VScroll vs_TextView(&textView);
    //在增加垂直滚动条的基础上增加滚动横条
    HScroll hs_vs_TextView(&vs_TextView);
    //在增加水平与垂直滚动条之后增加边框
    Border b_hs_vs_TextView(&hs_vs_TextView);
    b_hs_vs_TextView.draw();
    return 0;
}
```

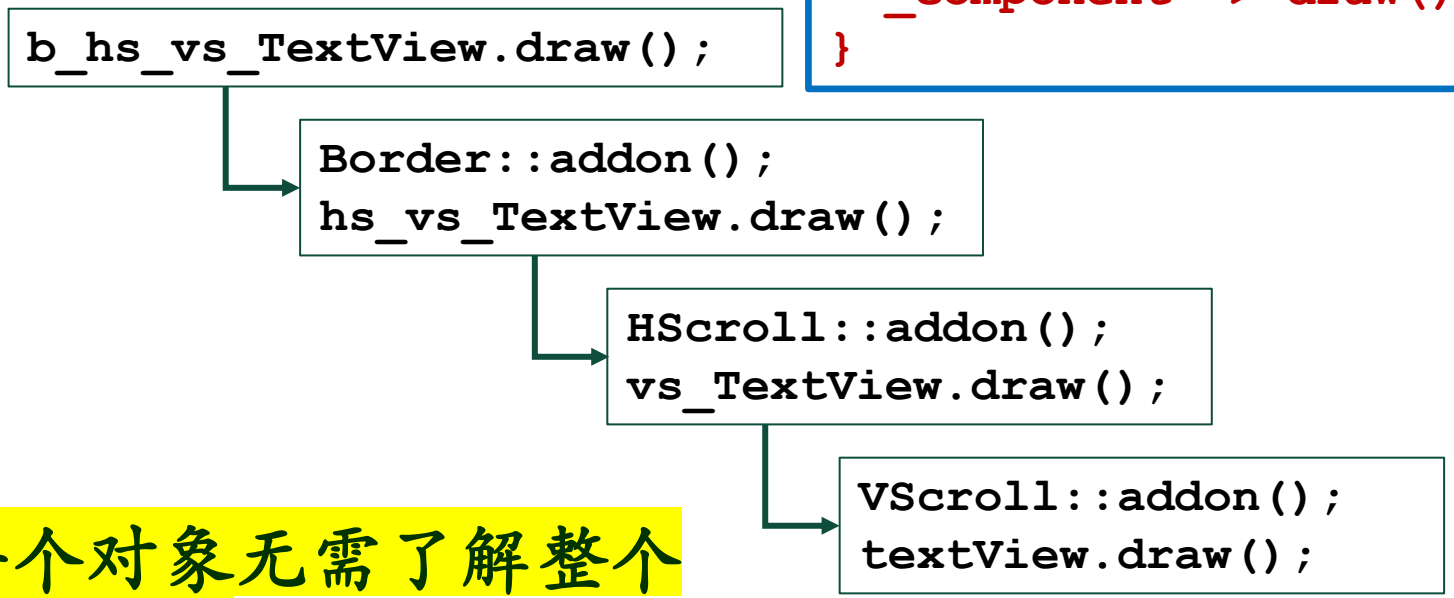
运行过程与结果

```
Bordered HScrolled VScrolled TextView.
```



调用的链式关系

```
void Decorator::draw() {
  addon();
  _component -> draw();
}
```



- 每个对象无需了解整个链的全貌
- 每一次都是将之前的版本完全包裹住，再增加新的功能。换句话说，有多少个新功能就包裹几次

装饰与策略

- 相同点
  - 通过对象的组合修改对象的功能
  - 以组合替代简单继承，更加灵活，减少冗余
- 不同点

策略	装饰
• 修改对象功能的内核（行为） • 组件必须了解有哪些需要选择的策略，侧重于功能选择	• 修改对象功能的外壳（结构） • 组件无需了解有哪些可以装饰的内容，侧重于功能组装

装饰与代理

- 都用来改变对象的行为
- 可以把“装饰”看成是一连串的“代理”

装饰	代理
• 为被装饰对象增加额外的行为 • 不影响被装饰对象的原有功能 • 不创建被装饰对象，只是将新功能添加到已有对象上 • 经常多重嵌套装饰 传入的指针越来越复杂	• 常用来对被代理对象进行更精细的控制 • 被代理对象不存在时常创建被代理对象 • 少见多重嵌套 只传进一次指针

创建型模式

- 结构型设计模式关心对象组成结构上的抽象，包括接口，层次，对象组合等。
  - 适配器模式在类与类之间进行转接，能够了类的复用度与灵活性
  - 代理/委托模式减少了类与类层次间的耦合，使得类各自的职责清晰
  - 装饰器模式可以动态扩展被装饰类的功能，并留有接口进行持续扩展
- 核心就在于抽象结构层次上的不变量，尽可能减少类与类之间的联系与耦合，从而能够以最小的代价支持新功能的增加。

设计模式总结

设计模式回顾

■行为型模式（Behavioral Patterns）

- 关注对象行为功能上的抽象，提升对象在行为功能上的可拓展性，能以最少的代码变动完成功能的增减
- 常用于描述对类和对象的交互与职责分配

行为型模式

- **模板方法模式：** 定义算法骨架，将具体步骤的实现放到子类中实现。可以在不改变算法流程的情况下，自定义某些步骤。
- **策略模式：** 定义一类算法，将每个算法分别封装，不同算法可以相互替换。
- **迭代器模式：** 用于遍历数据集合（数组、链表、树、图等），解耦算法与数据访问。

设计模式回顾

■行为型模式（Behavioral Patterns）

- 关注对象行为功能上的抽象，提升对象在行为功能上的可拓展性，能以最少的代码变动完成功能的增减
- 常用于描述对类和对象的交互与职责分配

行为型模式

- **观察者模式：** 将事件观察者与被观察者解耦
- **职责链模式：** 多个处理器处理按职责处理同一请求
- **解释器模式：** 某个语言定义它的语法（或者叫文法）表示，并定义一个解释器用来处理这个语法
- **备忘录模式：** 捕捉并存储对象内部状态，以便后续恢复
- **访问者模式：** 允许多个操作应用到一组对象上，解耦操作和对象本身

设计模式回顾

■结构型模式（Structural Patterns）

- 关注对象之间结构关系上的抽象，从而提升对象结构的可维护性、代码的健壮性，能在结构层面上尽可能的解耦合
- 常用于处理类和对象的组合关系

结构型模式

- **适配器模式：**将不兼容的接口转换为可兼容的接口
- **代理/委托模式：**在不改变原始类接口的条件下，为原始类定义一个代理类，增加控制访问
- **装饰模式：**用组合来替代继承，给原始类添加增强功能

设计模式回顾

■结构型模式（Structural Patterns）

- 关注对象之间结构关系上的抽象，从而提升对象结构的可维护性、代码的健壮性，能在结构层面上尽可能的解耦合
- 常用于处理类和对象的组合关系

结构型模式

- **组合模式：**将一组对象组织成树形结构，将单个对象和组合对象都看作树中的节点，以统一处理逻辑
- **外观模式：**它通过封装细粒度的接口，提供组合各个细粒度接口的高层次接口，来提高接口的易用性
- **享元模式：**复用不可变对象，节省内存

设计模式回顾

■创建型模式（Creational Patterns）

- 将对象的创建与使用进行划分，从而规避复杂对象创建带来的资源消耗，能以简短的代码完成对象的高效创建
- 用于对象的创建

创建型模式

- **抽象工厂模式：**提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类
- **建造者模式：**建造者模式用来创建复杂对象，可以通过设置不同的可选参数，“定制化”地创建不同的对象。
- **工厂方法模式：**用来创建不同但是相关类型的对象，由给定的参数来决定创建哪种类型的对象

设计模式回顾

■创建型模式（Creational Patterns）

- 将对象的创建与使用进行划分，从而规避复杂对象创建带来的资源消耗，能以简短的代码完成对象的高效创建
- 用于对象的创建

创建型模式

- **原型模式：** 利用对已有对象（原型）进行复制（或者叫拷贝）的方式，来创建新对象，以节省时间
- **单例模式：** 用来创建全局唯一的对象

设计原则

开闭原则

- 一个软件实体，比如类，模块，函数应该**对扩展开放**，**对修改关闭**
- 最基础的设计原则

单一职责原则

- 每个类应该只有一个职责，只有一个原因可以引起它的改变
- 例如：迭代器模式使得数据结构与算法分离；可视化程序设计中页面与逻辑分离

里氏代换原则

- 只要父类出现的地方子类就可以出现，即子类尽量不修改父类的数据与方法，实现基类代码的充分复用

设计原则

依赖倒转原则

- 要依赖于抽象，不要依赖于具体。针对接口编程，而不是针对实现编程。具体而言就是上层模块不应该依赖底层模块，使用接口和抽象类指定好规范，剩下的具体细节由实现类来完成
- 例如：策略模式/模板方法模式不依赖于具体的策略实现，只依赖于抽象

接口隔离原则

- 不要建立臃肿庞大的接口。即接口尽量细化的同时接口中的方法尽量少
- 功能拆分粒度太小，将使得类、接口的数量过多；功能拆分粒度太大，将使得类之间耦合度高，程序不灵活

设计原则

- **迪米特原则**
  - 最少知道原则，一个对象应该对其他对象有最少的了解，使得功能模块相对独立
- **合成复用原则**
  - 合成复用原则就是指在一个新的对象里通过关联关系（包括组合关系）来使用一些已有的对象，使之成为新对象的一部分；新对象通过委派调用已有对象的方法达到复用其已有功能的目的
  - 即在实现扩展类功能时，优先考虑使用组合而不是继承；如需要使用继承，则遵守里氏代换原则

- **在程序设计中尽量遵循七大原则，但也需根据实际情况调整，切勿滥用设计模式使得代码过度冗余**

结束

拓展阅读：智能指针中的设计模式

- C++中，指针是一个使用起来需要格外注意的东西，尤其是class中有指针，那么析构与释放就会是一个及其棘手的事情，稍有不慎就会程序错误或者内存泄漏。
- 我们要实现智能指针类，能够包裹指针，具有指针的各项功能，并能够进行引用计数，在计数为0时自动释放指针空间。

拓展阅读：智能指针中的设计模式

- 使用适配器模式可以进行指针的封装，并对外提供指针各项功能的接口。
- 但是，适配器模式仅仅只是接口的转换，其本身无法在提供接口的同时进行计数这样的功能控制。
- 我们如何在提供功能的同时进行计数控制呢?

拓展阅读：智能指针中的设计模式

- 代理模式：智能指针引用计数

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

template <typename T>
//提前声明智能指针模板类
class SmartPtr;
//辅助指针，用于存储指针计数以及封装实际指针地址
template <typename T>
class U_Ptr {
private:
    friend class SmartPtr<T>;
    U_Ptr(T *ptr) :p(ptr), count(1) { }
    ~U_Ptr() { delete p; }

    int count;
    T *p; //数据存放地址
};
```

拓展阅读：智能指针中的设计模式

- 代理模式：智能指针引用计数

```
template <typename T>
class SmartPtr {//智能指针
private:
    U_Ptr<T> *rp;    //进行实际指针操作的辅助指针
public:
    SmartPtr(T *ptr) :rp(new U_Ptr<T>(ptr)) { }
    //调动拷贝构造即增加引用计数
    SmartPtr(const SmartPtr<T> &sp) :rp(sp.rp) { ++rp->count; }
    SmartPtr& operator=(const SmartPtr<T>& rhs) {
        ++rhs.rp->count; //赋值号后的指针引用加1
        if (--rp->count == 0) delete rp; //原内部指针引用减1
        rp = rhs.rp;     //代理新的指针
        return *this;
    }
    ~SmartPtr() { //只有引用次数为0才会释放
        if (--rp->count == 0) delete rp;
    }
    //对智能指针操作等同于对内部辅助指针操作
    T & operator *() { return *(rp->p); }
    T* operator ->() { return rp->p; }
};
```

拓展阅读：智能指针中的设计模式

- 代理模式：智能指针引用计数

```
int main(int argc, char *argv[]) {
    //声明指针
    int *i = new int(2);
    //使用代理来包裹指针
    SmartPtr<int> ptr1(i);
    SmartPtr<int> ptr2(ptr1);
    SmartPtr<int> ptr3 = ptr2;
    //之后的操作均通过代理进行
    cout << *ptr1 << endl;
    *ptr1 = 20;
    cout << *ptr2 << endl;
    return 0;
}
```

```
2
20
```

拓展阅读：智能指针中的设计模式

- “变”与“不变”
  - SmartPtr<int>与 int* 有相同的接口
    - 操作符：*和->
    - 赋值操作符与初始化（拷贝构造）
    - 释放（析构）
  - SmartPtr<int>比 int* 增加了一些控制操作
    - 拷贝构造时引用计数加一
    - 析构时引用计数减一，直到引用计数为0时释放
    - 赋值时对当前引用计数和参数引用计数分别处理

扩展阅读：单例模式

- 所谓单例，就是只能构造一份实例的类

```
class Counter {
    // 显式删除拷贝构造函数与赋值操作符
    Counter(const Counter &) = delete;
    void operator =(const Counter &) = delete;

    int count;
    Counter() { count = 0; }
    ~Counter() {}

    static Counter _instance; // 全局唯一的实例
public:
    static Counter &instance() {
        return _instance;
    }
    // 成员函数而非静态方法
    void addCount() { count += 1; }
    int getCount() { return count; }
};
```

扩展阅读：单例模式

- 调用单例

```
class Counter { ... };
  // 定义类中的静态成员，单例在此被初始化
  Counter Counter::_instance;

int main() {
      // 由于删去了拷贝构造函数，必须存为引用
      Counter &c = Counter::instance();
      c.addCount();
      cout << c.getCount() << endl;
      return 0;
  }
```

扩展阅读：惰性初始化（Lazy Initialization）

- 能否在使用时再构造单例实例?

```
class Counter {
    // ...
public:
    static Counter &instance() {
        static Counter _instance;
        return _instance;
    }
    // ...
};
```

- 在第一次调用instance方法时才会构造单例

扩展阅读：单例模式

- 需要避免的情况：
  - 实例被重复构造
    - 由于构造函数为private，且拷贝构造函数、赋值操作符被显式删除，故无法重复构造。
  - 实例被意外删除
    - 由于析构函数为private，故无法被意外删除。

扩展阅读：单例模式

- 单例模式是存在争议的一种设计模式
- 优点：
  - 以相对安全的形式提供可供全局访问的数据
  - 实现似乎比较简单
- 缺点：
  - 难以完全正确地实现
  - 违反单一职责原则
  - 过度使用这一方法会使得实际的依赖关系变得隐蔽