程序设计实习（实验班-2024春）
面向对象编程：设计模式选讲

授课教师：姜少峰

助教：冯施源 吴天意

Email: shaofeng.jiang@pku.edu.cn

何谓设计模式?

类比“三十六计”

- 更像是一些best practice/案例的整理，给出一些好的设计“技术”和“思想”

也可以类比“分治”、“动态规划”等

- 这些面向对象方法的好处不光在写一个程序上，更在长远的维护和复用上
  - 例如有新功能需求，如何进行灵活的扩展?
- 不仅是程序设计之前就应考虑的，也是重构时的重要参考

设计模式

面向对象编程的经典

《Design Patterns: Elements of Reusable Object-Oriented Software 》by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, 1994

分为三大类

- 创建型模式：提供创建对象的机制， 提升已有代码的灵活性和可复用性
- 结构型模式：如何将对象和类组装成较大的结构， 并保持结构的灵活和高效
- 行为模式：负责对象间的高效沟通和职责委派

创建型模式

Factory

工厂模式

- 场景：父类/接口定义了一切需要的功能（子类不增加新功能，只是实现接口）
  - 因此具体实现可以对用户忽略，甚至子类的类名也不必暴露
  - 尤其适合具体实现可能比较dirty、经常需要新增具体实现的时候

factory举例：根据地址读文件

```
struct DataReader {
    virtual int* getData() = 0;
};

struct HttpDataReader : public DataReader {
    HttpDataReader(string loc) {}
    int* getData() { /*...*/ }
};

struct FileDataReader : public DataReader {
    FileDataReader(string loc) {}
    int* getData() { /*...*/ }
};

struct SftpDataReader : public DataReader {
    SftpDataReader(string loc) {}
    int* getData() { /*...*/ }
};
```

用户只需要用这个接口

几个具体实现这个接口的类
这几个类用户不需要知道

```
class DataReaderFactory {
    string protocol;
    string location;
    public:
    DataReaderFactory(string uri) {
        auto i = uri.find("://");
        protocol = uri.substr(0, i);
        uri.erase(0, i + 3);
        location = uri;
    }
    DataReader* getReader() {
        if (protocol == "file") return new FileDataReader(location);
        else if (protocol == "http") return new HttpDataReader(location);
        else if (protocol == "sftp") return new SftpDataReader(location);
        return NULL;
    }
};
```

根据“://”来划分协议和地址

根据协议来创建对应的Reader

```
int main() {
    DataReaderFactory fac("file://a.txt");
    DataReader* reader = fac.getReader();
    int* fileData = reader->getData();
    int* httpData = DataReaderFactory("http://xxx.com/file.txt").getReader()->getData();
    return 0;
}
```

直接给地址就可以得到对应的reader来读取数据
具体reader内部实现用户无法（也不需要）看到

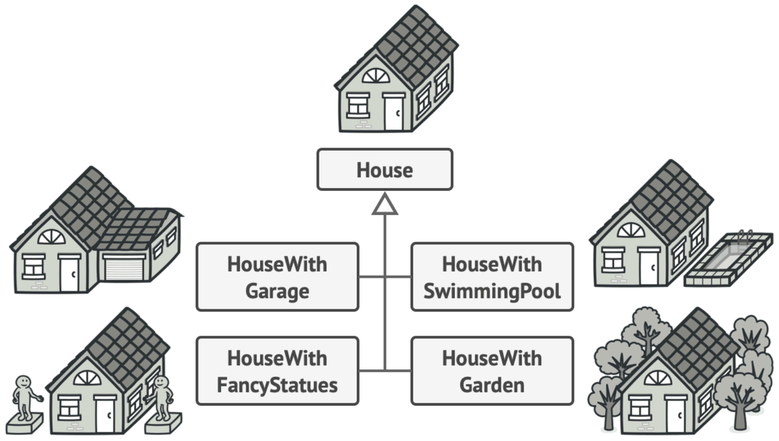
factory举例：跨平台UI库

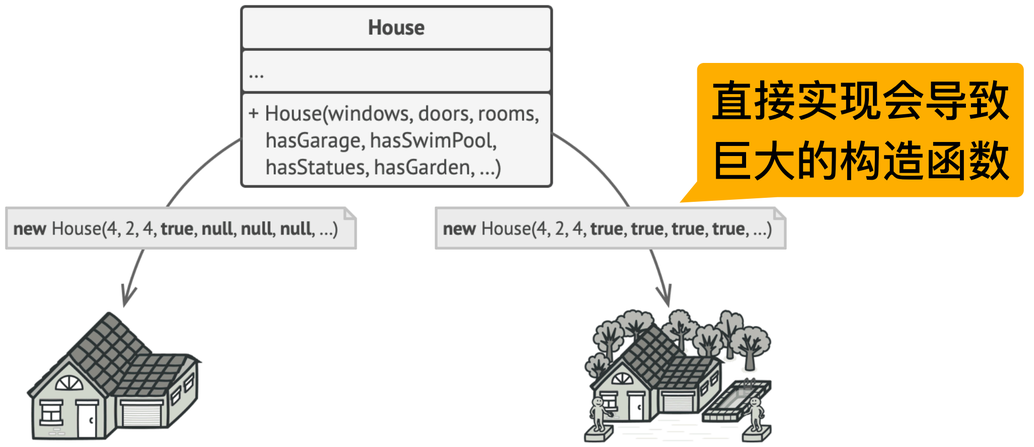
- 考虑实现一个平台无关的Dialog对话框类
  - Win/Linux/Mac的底层实现完全不同，Dialog类具体需要对这三个平台实现
- Dialog定义好一切Dialog需要的函数，并定义成接口，用户之后只要调用这些
  - 对于不同平台，写子类实现这些接口

Builder

生成器模式

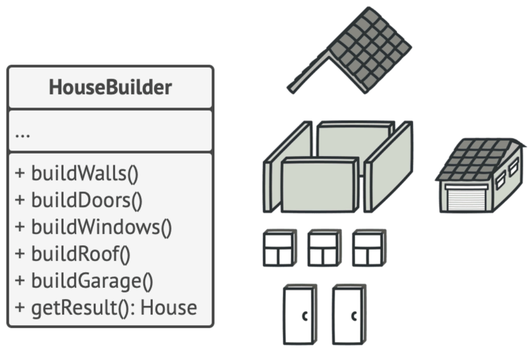
- 考虑一个组成部分非常多的类，如汽车、房屋
- 并且各部分的排列组合有很多可能性
- 构造时需要给出每个部分的定义，潜在会导致巨大的构造函数





Builder

- 这时应该引入额外一层，称为Builder，将房子各个部分分别写成一个build函数
  - 最后调用getResult得到具体的构造后的结果



  - 而且不需要调用所有的buildXXX，只需要调用需要的

Builder

Director

- 还可以实现一个Director类进行额外一层抽象实现复用
  - 例如一般客厅都有若干组可能的基本组成部分，之后根据需要再额外加元素
  - 可以写一个Director专门构造这些基本组成部分

例如可以指定有几扇窗，是否含有开放式厨房

- Director：传入一个builder，代理用户调用builder的函数

```
struct LivingRoomDirector {
    Builder* builder;
    LivingRoomDirector(Builder* b) : builder(b) {}
    void changeBuilder(Builder* b) {builder = b;}
    void makeLivingRoom(string type) {
        if (type == "a") builder->buildStepA();
        else if (type == "b") builder->buildStepB();
        /*...*/
    }
};
```

这里假定builder调用buildXXX会改变自己的状态

Prototype

原型模式：“复制”的活用

- 场景：类的构造/初始化很复杂，但整个程序有若干个典型的初始对象
  - 例如房子的Builder，设有5种基本“模式”房子，一般基于这5种模式继续构造
- 一种方法是可以考虑创建5个典型模式对应的子类
- Prototype：不建立子类，创建5个基本模式的房子对象，之后的都从这5个复制
  - 这5个可以定义成房子builder类的5个static对象

```
struct Builder {
    static map<string, Builder> typicalBuilders;
    static void addTypicalBuilder(string str, const Builder& b) {
        typicalBuilders[str] = b;
    }
    /*...*/
};
```

发现有常用的typical builder就add进去之后可以复用

```
int main() {
    Builder::addTypicalBuilder("type1", new Builder());
    Builder b = Builder::typicalBuilders["type1"];
    return 0;
}
```

此处应深复制，否则可能会导致共用同一份type1

Prototype

- 另外场景：需要复制动态绑定的对象
- Prototype pattern就是所有的类都实现一个clone虚函数
  - 用来返回当前类的一个copy

可以是深复制也可以是浅复制，看需求

- 类似复制构造函数的用途，但是有重要区别：复制构造无法动态绑定
  - 例如Parent* parent = new Child，然后parent->clone()可返回一个child对象

```
struct Clonable {
    virtual Clonable* clone() = 0;
};
```

```
struct Parent : public Clonable {
   Parent* clone() {
        Parent* p = new Parent;
        *p = *this;
        return p;
    }
};
```

具体类中clone可以返回自己类的类型，依然算覆盖

```
struct Child : public Parent {
   Child* clone() {
        Child* p = new Child;
        *p = *this;
        return p;
    }
};
```

```
int main() {
    Parent* p = new Child;
    Parent* q = p->clone();
   return 0;
}
```

Singleton

单例模式

- 提供全局可访问的唯一的对象/实例
  - 读写同一个资源，如文件、网络、数据库资源
  - 另外：例如logger和configuration

```
struct Singleton
{
    private:
    Singleton() {}
    static Singleton* instance;

    public:
    static Singleton* getInstance() {
        if (instance == NULL)
            instance = new Singleton();
        return instance;
    }
};

Singleton* Singleton::instance = NULL;
```

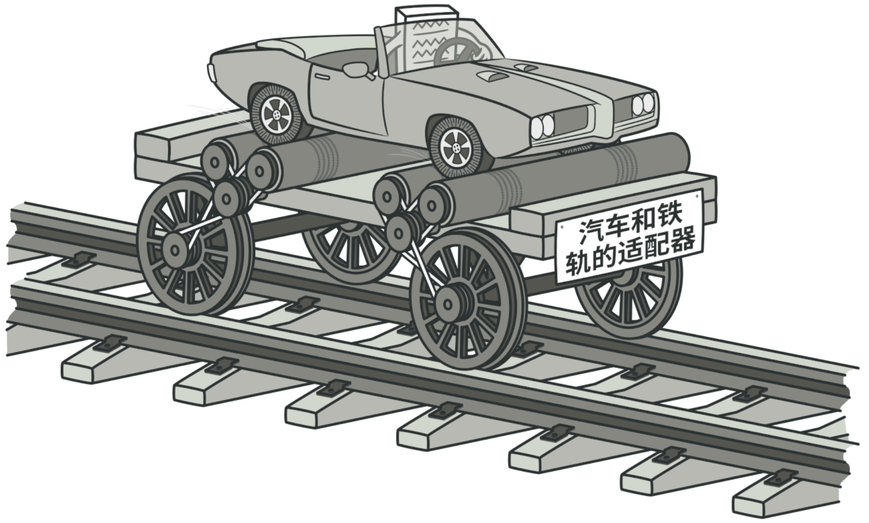
结构型模式

Adapter

适配器模式

- 场景：有一些老/其他地方的代码与现在程序的接口不同，但依然需要用

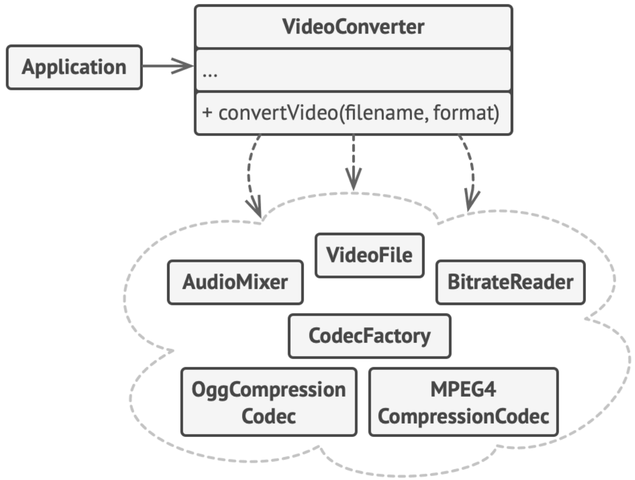
```
struct NewInterface {
    virtual void func(Data*) = 0;
};

struct OldService {
    void oldFunc(SpecialData*);
};

class Adapter : public NewInterface {
    OldService* adaptee;
    public:
    void func(Data* data) {
        SpecialData* specialData = convertToOldServiceFormat(data);
        adaptee->oldFunc(specialData);
    }
}
```



Facade

外观模式

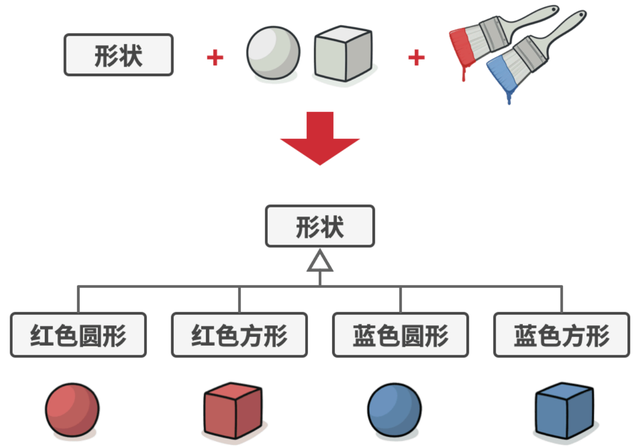
- 场景：你需要实现的功能比较简单，但是需要借助复杂的类库
- 为使不需要直接与复杂类库打交道，可以添加一个“外观”层
  - 例：用比如FFmpeg写一个视频转码的函数encode(filename, format)



Bridge

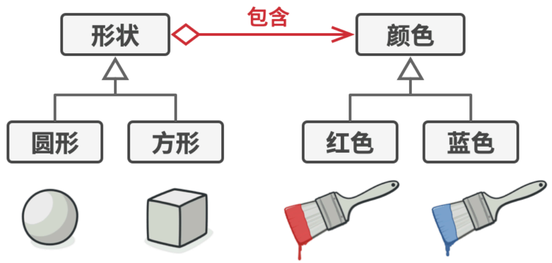
桥接模式

潜在的也可以是多种变化维度，但2种比较典型



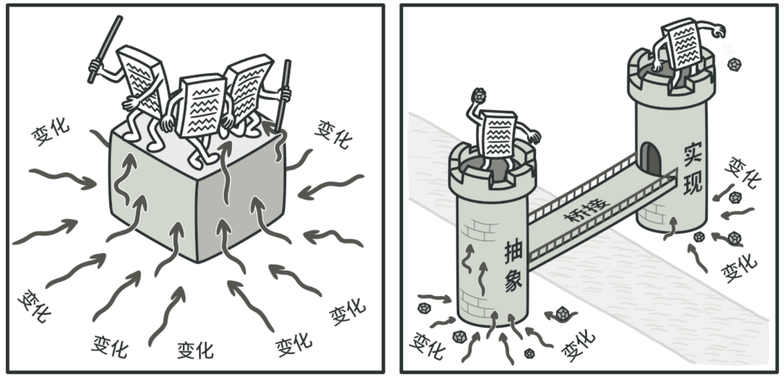
- 场景：类是由2种变化维度的排列组合形成的
  - 如果用继承，就会有很多子类

- Bridge强调应该尽量用复合，而不是继承，从而可以将不同变化维度各个击破



Bridge

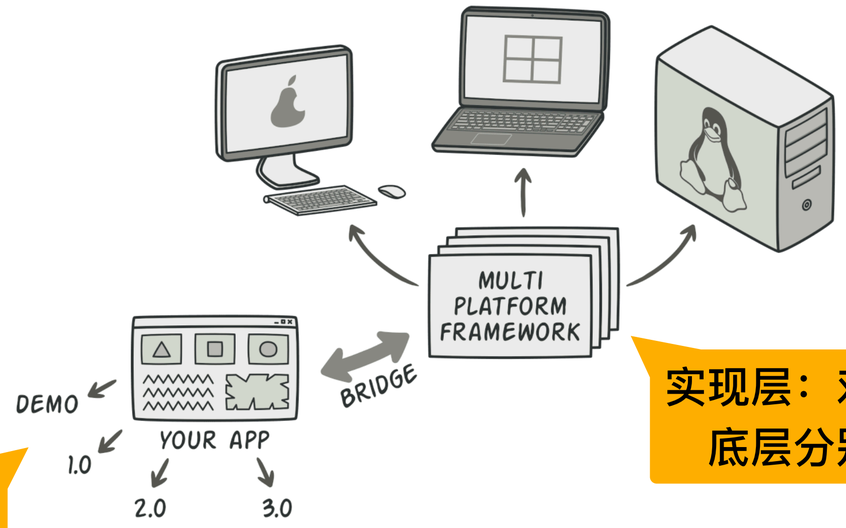
- 例如想写一个GUI图形程序，可以有两个变化维度：
  - 更多的GUI组件：例如button，text box，label等
  - 更多的底层支持：例如支持Win/Linux/Mac，甚至是浏览器、手机系统



Bridge

抽象层与实现层

- GUI对应抽象层，底层实现对应于实现层

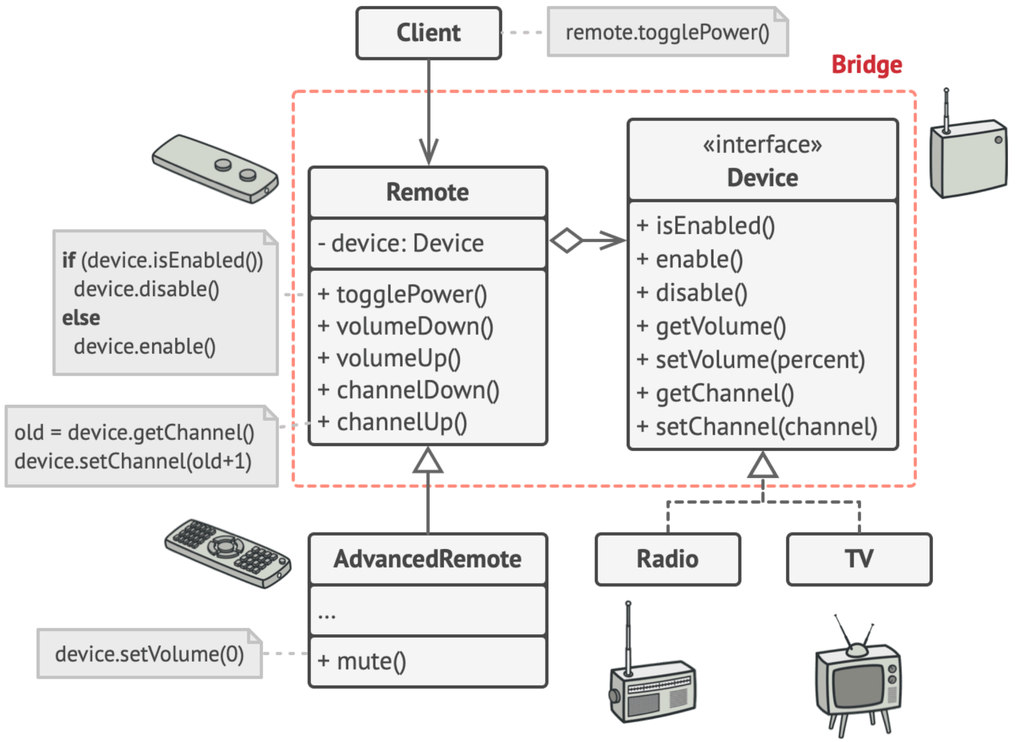


抽象层：控制程序的外观及直接支持的功能

实现层：对应于不同的底层分别进行实现

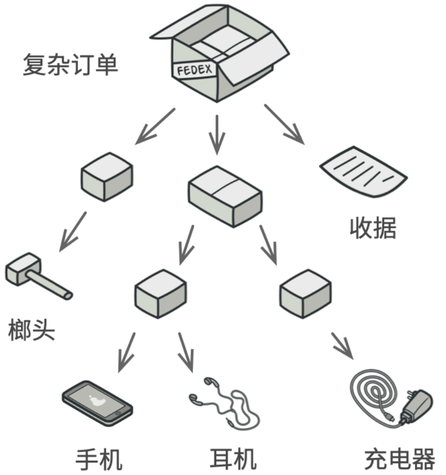
Bridge

- 一般而言：
  - 找出整个任务中独立变化的维度，每个维度对应一个bridge的某一端
  - 对每个维度对应建立一个抽象类，定义这个维度与其他维度的交互手段
    - 每个维度的具体变化都是这个抽象类的继承/实现
  - 只需要使用每个维度的抽象类的接口进行编程，最后将具体实现代入即可


Client
remote.togglePower()
Bridge
«interface»
Device
+ isEnabled()
+ enable()
+ disable()
+ getVolume()
+ setVolume(percent)
+ getChannel()
+ setChannel(channel)
Remote
- device: Device
+ togglePower()
+ volumeDown()
+ volumeUp()
+ channelDown()
+ channelUp()
if (device.isEnabled())
device.disable()
else
device.enable()
old = device.getChannel()
device.setChannel(old+1)
AdvancedRemote
...
+ mute()
device.setVolume(0)
Radio
TV

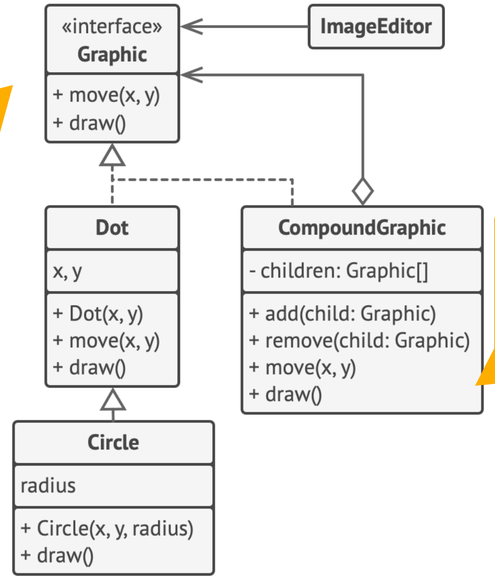
Composite

组合模式



- 场景：对象具有递归/嵌套特性的表达方式
  - 递归进行，一般类似于树
- 例子：ImageEditor

几何元素的接口：都支持move和draw操作



对所有元素的move和draw可以通过递归调用draw/move来进行，类似于树的遍历

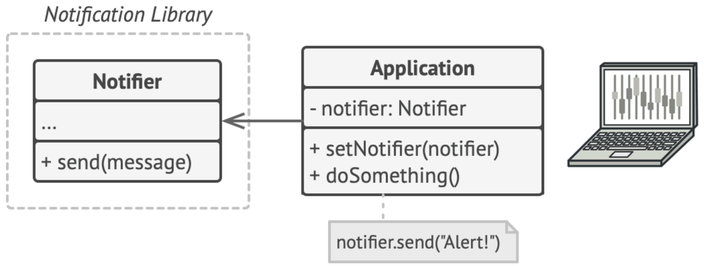
这个复合Graphic的类是若干几何元素的聚合（容器），支持对所含元素的整体move和draw

Dot和Circle是具体的几何元素（树的叶子）

Decorator

装饰模式

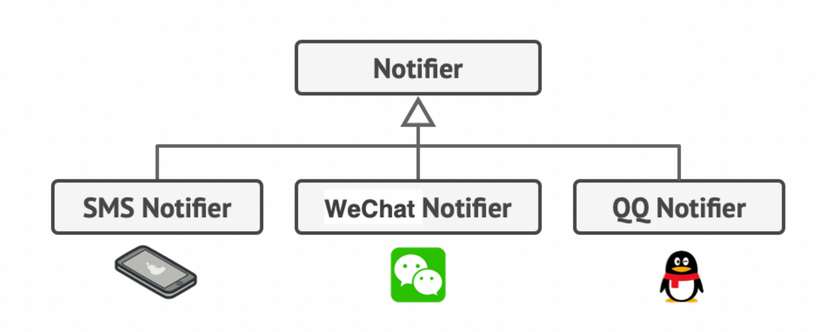
- 我们提供给用户一个Notifier类，内含一个send函数来给特定邮件地址发通知
- 用户的使用方法是一次性初始化这个Notifier类，然后调用send来发通知



- 问题：用户后来又想不光通过邮件发通知，还要支持短信/微信等，怎么办？
  - 特别地，想尽量与现有的Notifier的接口一致，即初始化后调用send

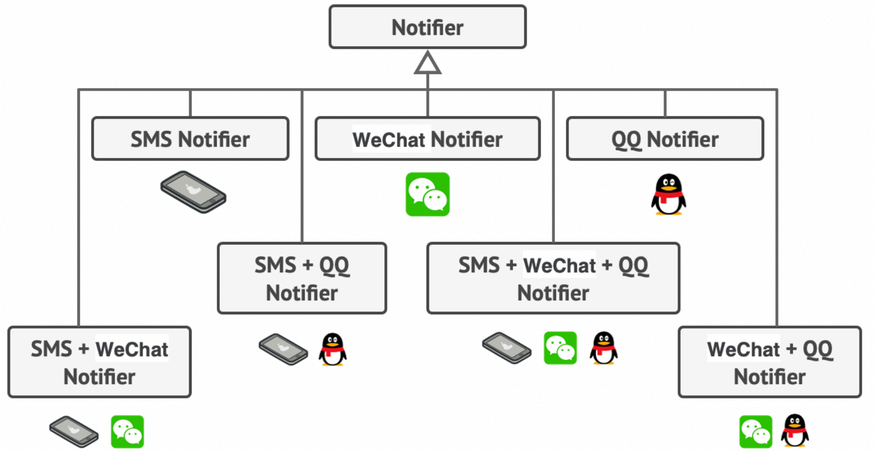
继承?

- 可以通过继承解决：



- 但是突然又有新需求：要求能调用一个send，同时给多个渠道发信息，并且要求在初始化的时候配置这种行为

继续使用继承将非常不灵活

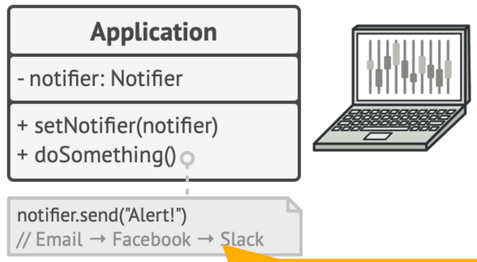


Decorator

- 解决方法：不直接使用继承创建所有组合，而是利用一个Wrapper类实现聚合
- 具体来说：用户想达到如下效果，并且已有的Notifier类不允许改

```
stack = new Notifier()
if (facebookEnabled)
    stack = new FacebookDecorator(stack)
if (slackEnabled)
    stack = new SlackDecorator(stack)

app.setNotifier(stack)
```

用户侧用这种连续初始化的方法来添加需要的渠道，最后setNotifier



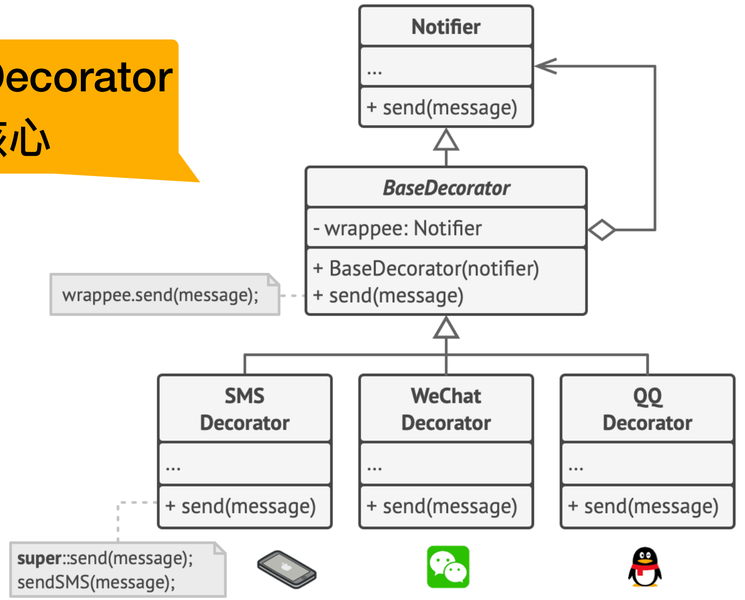

```
notifier.send("Alert!")
// Email → Facebook → Slack
```

最后依然只调用一个send，就通过多个渠道发送了

```
struct Notifier { // 原始的notifier, 只能发email
    virtual void send(const string& mes) {
        cout << "email" << " " << mes << endl;
    }
};

class BaseDec : public Notifier {
    Notifier* wrappee;
    public:
    BaseDec(Notifier* n) : wrappee(n) {}
    void send(const string& mes) {
        wrappee->send(mes);
    }
};

struct SMSDec : public BaseDec {
    SMSDec(Notifier* n) : BaseDec(n) {}
    void send(const string& mes) {
        BaseDec::send(mes);
        cout << "SMS" << " " << mes << endl;
    }
};

struct WCDec : public BaseDec {
    WCDec(Notifier* n) : BaseDec(n) {}
    void send(const string& mes) {
        BaseDec::send(mes);
        cout << "WC" << " " << mes << endl;
    }
};
```

应该定义虚析构，此处为省空间省略

这个中间层是Decorator模式的核心




```
int main() {
    Notifier* note = new Notifier();
    note = new SMSDec(note);
    note = new WCDec(note);
    note->send("mes");
    return 0;
}
```

此处可以加if来判断要不要把功能加上

输出email mes
SMS mes
WC mes

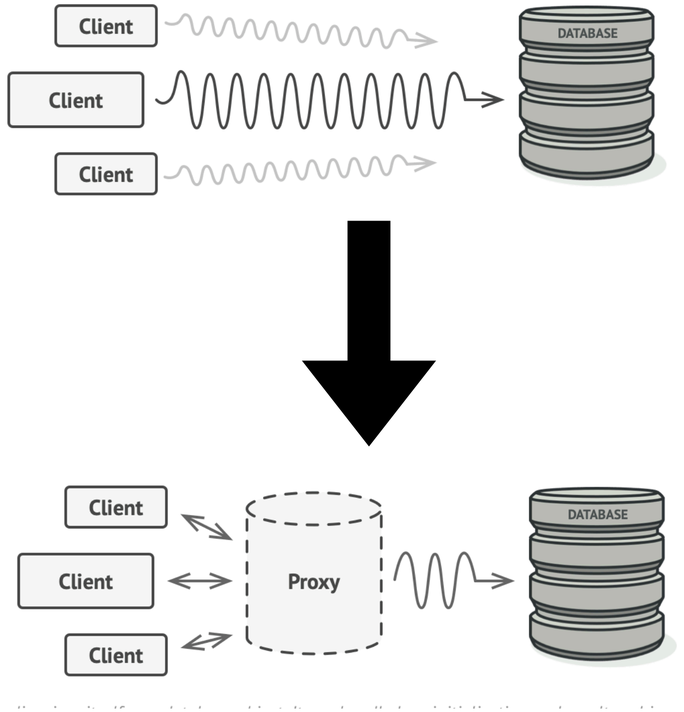
你能看出为什么输出是这个吗?

Proxy

代理模式

- 场景：程序中有一个很消耗资源的对象x，但只有很少的时候会被调用
  - 例如数据库连接
- 解决：可以使用lazy initialization，但是放在哪里最好?
  - 如果放在每次使用的时候，将需要很多重复的代码
- proxy：写一个和原始对象x所属类X接口完全相同的类，但是进行lazy initialize

Proxy


Client
Client
Client
DATABASE
Client
Client
Client
Proxy
DATABASE

Proxy

其他例子

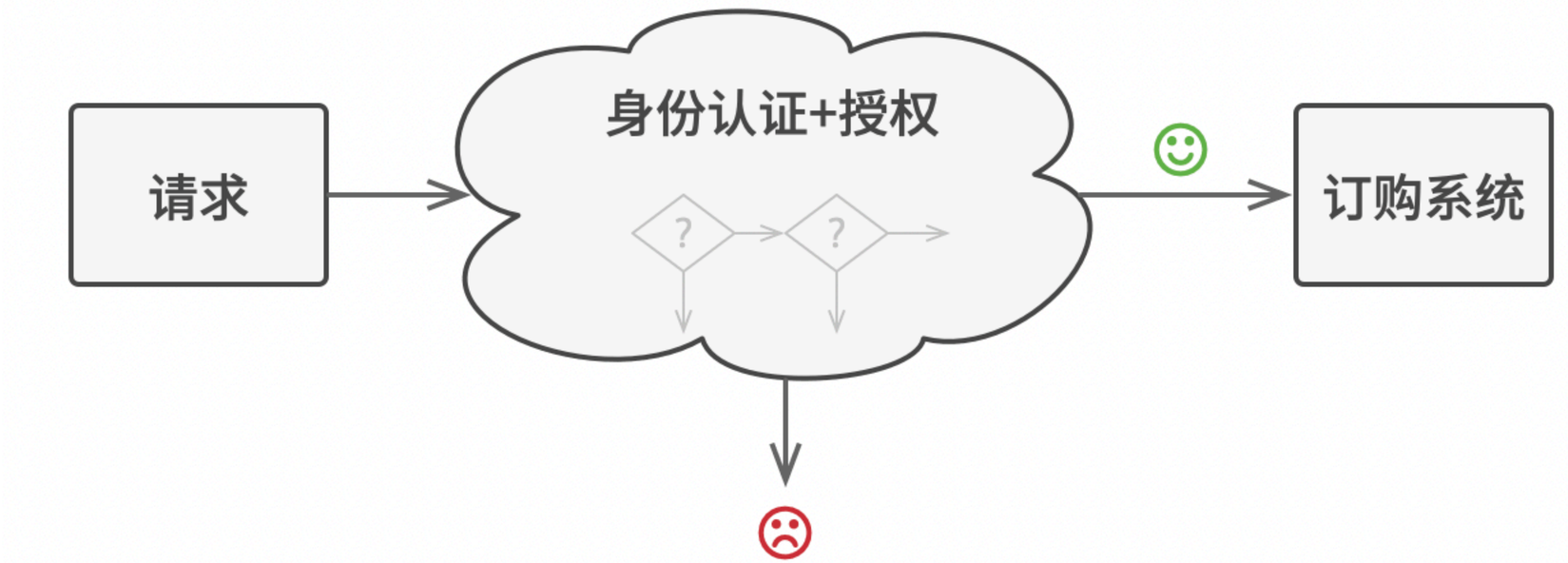
- 缓存：有个Youtube下载器类库，可以通过proxy增加缓存
  - 这样用户的相同请求可以不再重新下载，而是直接取缓存
- 访问控制：有一些敏感资源，套上一层proxy可以只让有授权的用户访问
- 远程proxy：要访问的资源/服务在远端，proxy可以提供一个缓存/转发
- smart reference：跟踪资源被多少client访问，如果没client在用就主动释放

行为模式

Chain of Command

责任链模式

- 场景：开发一个在线购物网站的权限认证模块



Chain of Command

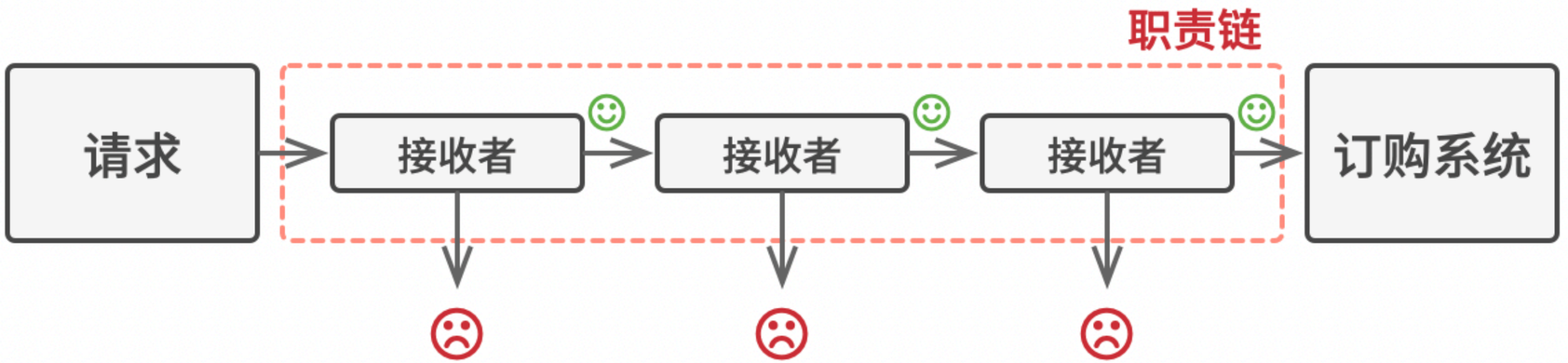
后续需求

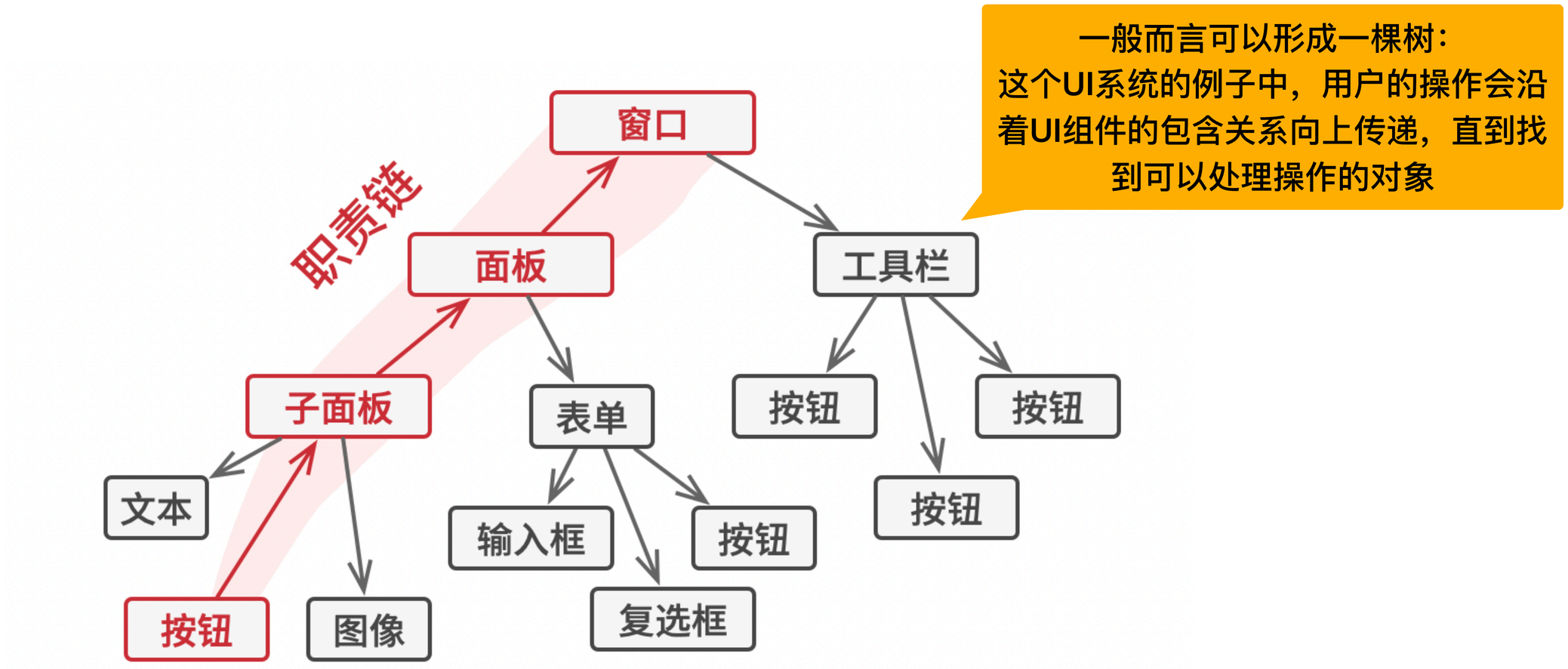
- 有人注意到有暴力穷举密码破译的隐患，加上过滤同一IP多次连续请求
- 又有人说同样的请求应该缓存下来，而不是每次重新验证

- 如何实现这些日益增加的验证需求呢?

写在同一个函数里的问题：
一大堆if-else的逻辑高度耦合，修改一个步骤可能会影响其他步骤
更重要的：以后开发其他系统时很难实现代码复用

实现上类似链表：
每个接收者存储下一个接收者；请求到来时，先判断自己是否可以处理，可以的话就地处理，否则发给下一个




一般而言可以形成一棵树：
这个UI系统的例子中，用户的操作会沿着UI组件的包含关系向上传递，直到找到可以处理操作的对象
职责链
窗口
面板
工具栏
子面板
表单
按钮
按钮
文本
输入框
按钮
按钮
按钮
图像
复选框

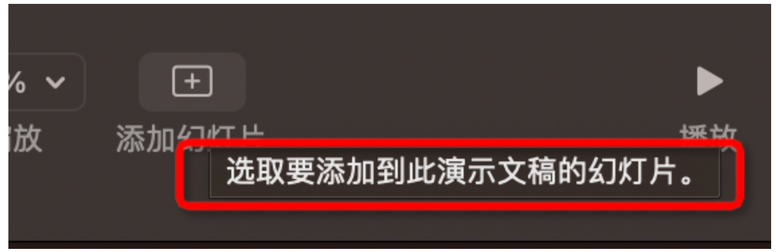
代码实现

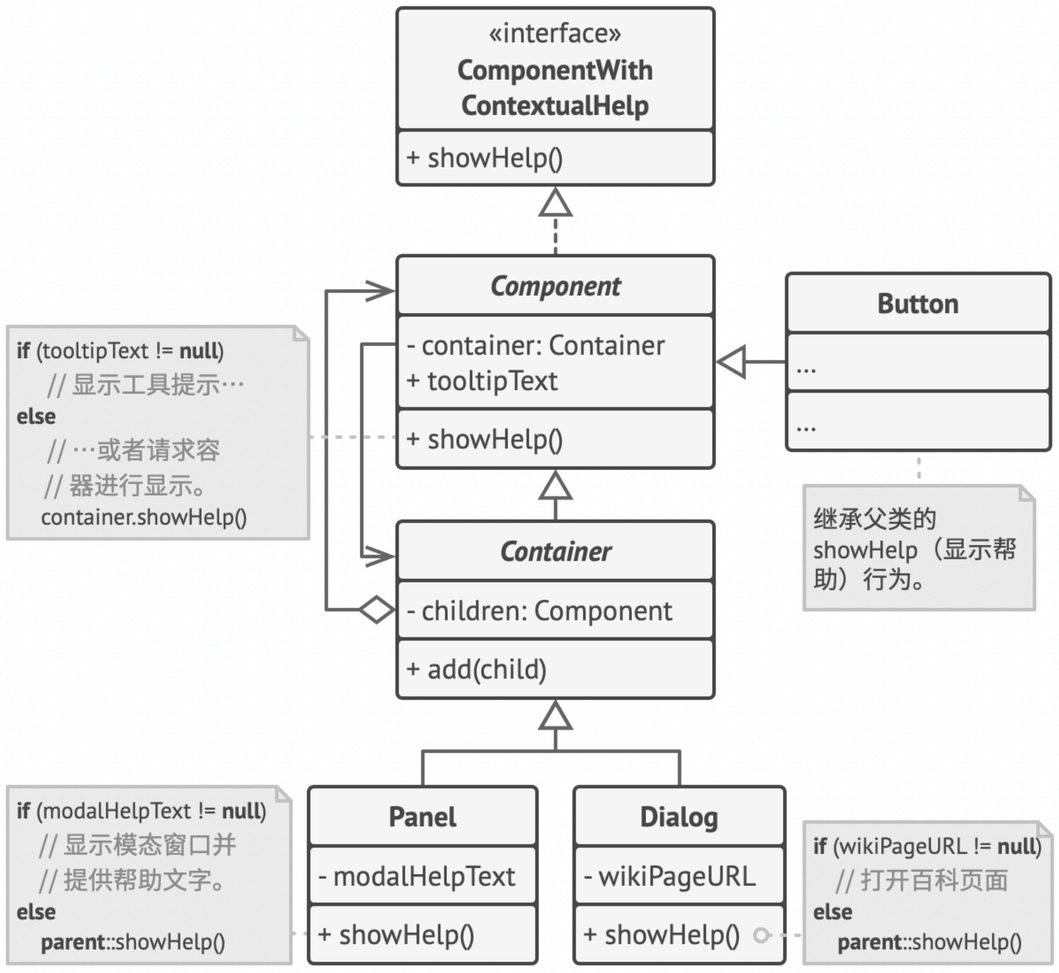
```
int main() {
    Handler* h1 = new Handler1();
    Handler* h2 = new Handler2();
    h1->setNext(h2);
    h1->handle("req");
    return 0;
}
```

```
struct Handler {
    virtual void setNext(Handler* nextHandle) = 0;
    virtual void handle(string request) = 0;
};

class BaseHandler : public Handler{
    Handler* next = NULL;
    public:
    void setNext(Handler* n) {next = n;}
    void handle(string req) {
        if (next) next->handle(req);
    }
};

class Handler1 : public BaseHandler {
    bool canHandle(string req) {return false;}
    public:
    void handle(string req) {
        if (canHandle(req)) cout << "Handler1" << endl;
        else BaseHandler::handle(req);
    }
};

class Handler2 : public BaseHandler {
    bool canHandle(string req) {return true;}
    public:
    void handle(string req) {
        if (canHandle(req)) cout << "Handler2" << endl;
        else BaseHandler::handle(req);
    }
};
```

这里会调用next的handle

一个UI显示帮助提示的例子

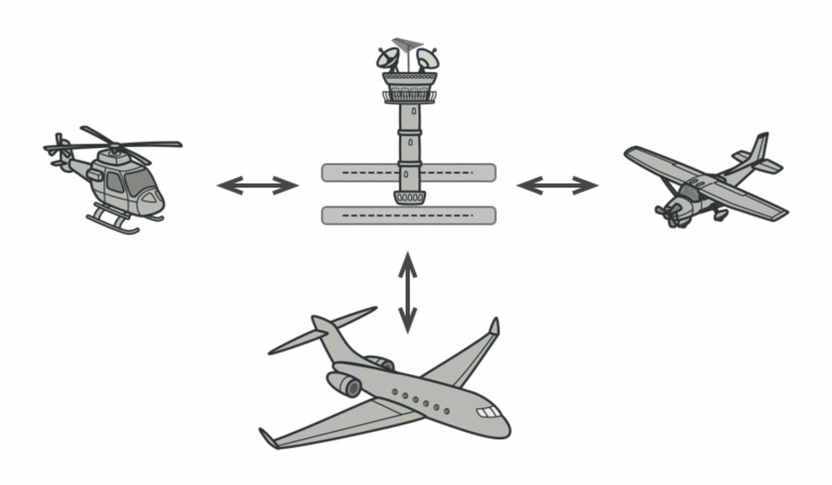




Mediator

中介者模式

- 有很多互相需要传递信息、互相需要调用对方功能的组件
  - 例如UI组件，一个被点按/触发后，要更新整个UI其他所有组件的状态
- 坏的设计：有n个组件，每个组件都和其他n - 1个直接交互



现实中的类比：
飞临同一空域的飞机要经过塔台互相联系而不要直接通信
这个塔台就是中介者

Mediator

举例

- 例如：设计一个简单的文本编辑UI有复制、剪切、粘贴按钮，一个文本区域
- 无文本选择时剪切、复制不可用

- 要实现上述功能，就需要在各个按钮和文本区域触发对应函数后互相更新

如何实现

```
class Component;
struct Mediator {
    virtual void notify(Component* comp, string mes) = 0;
};

class Component {
    // …
    protected:
    Mediator* med;
    public:
    Component(Mediator* m) : med(m) {}
};
```

重点：Mediator接口
notify函数

所有的UI组件都继承自Component
Component里面存储一个Mediator

- 定义一个中介者接口Mediator，里面有一个notify函数
  - 各个UI组件（如按钮、文本区域）都存储自己的Mediator
  - 有事件发生时调用自己mediator的notify函数告知mediator
- mediator接到了notify之后，根据notify的来源、信息来决定该做什么
- 这里mediator类中应该存储所有组件的指针，方便互相调用
- 假定Component会检测输入，并调用selectionChanged()和clicked()

```
class Component;
struct Mediator {
    virtual void notify(Component* comp, string mes) = 0;
};

class Component {
    // …
    protected:
    Mediator* med;
    public:
    Component(Mediator* m) : med(m) {}
};
```

```
struct Button : public Component {
    Button(Mediator* m) : Component(m) {}
    void setEnabled(bool e) {/*...*/}
    void clicked() {
        med->notify(this, "clicked");
    }
};

class TextField : public Component {
    string text;
    int curPos, selLen;
    public:
    TextField(Mediator* m) : Component(m) {}
    void selectionChanged() {
        med->notify(this, "selection changed");
    }
    void setSelLen(int len) {
        if (len != selLen) {
            selLen = len;
            selectionChanged();
        }
    }
    void delSelected() {text.erase(curPos, selLen); setSelLen(0);}
    string getSelected() {return text.substr(curPos, selLen);}
    void insert(string str) {text.insert(curPos, str);}
};
```

具体UI组件都是Component子类

当Button被点按后，需要notify mediator

当选中区域改变后，需要notify mediator

修改选中区域需要主动调用notify

若干工具函数用来查询/修改TextField的内容

总的App类就是一个Mediator

```
class App : public Mediator {
    Button *cpBtn, *pasteBtn, *cutBtn;
    TextField* tf;
    string clipBoard;

    void handleCopy() {clipBoard = tf->getSelected();}
    void handlePaste() {
        tf->delSelected(); tf->insert(clipBoard);
    }
    void handleCut() {handleCopy(); tf->delSelected();}
    void handleSelectionChange() {
        if(tf->getSelected().size()) {
            cpBtn->setEnabled(true); cutBtn->setEnabled(true);
        }
        else {
            cpBtn->setEnabled(false); cutBtn->setEnabled(false);
        }
    }

    public:
    App() {
        cpBtn = new Button(this);
        pasteBtn = new Button(this);
        cutBtn = new Button(this);
        tf = new TextField(this);
    }
    void notify(Component* comp, string event) {
        if (comp == cpBtn) {handleCopy();}
        if (comp == pasteBtn) {handlePaste();}
        if (comp == cutBtn) {handleCut();}
        if (comp == tf) {handleSelectionChange();}
    }
};
```

总管各个UI组件的创建与维护

这几个handleXXX是主要的业务逻辑

这里根据notify的信息来源判断应该执行什么操作；有时需要利用event的信息（这里不需利用）

Memento

备忘录模式

- 场景：文本编辑器想支持存储历史记录和撤销
  - 希望编辑器生成快照，以放入某个容器中
  - 不希望容器可以看到任何文本编辑器的数据，只有编辑器可以访问这些数据
- 如何实现?
  - 又希望把数据快照放在别人那里，又不希望别人看到?

如何实现

- 创建一个备忘录类，所有函数、变量都是private的，但将编辑器设置成友元
- 编辑器用自己的内部状态构造备忘录
- 备忘录成为了编辑器数据的实体，同时外界又不可访问

编程实现

```cpp
struct State {
    int x = 0;
};

class Originator;
class Memento {
    State state;
    Memento(const State& s) : state(s) {}
    State getState() {return state;}
    friend class Originator;
};

class Originator {
    State state;
    public:
    Originator() {}
    void changeState() {
        state.x++;
        cout << "changed to " << state.x << endl;
    }
    Memento* save() {return new Memento(state);}
    void restore(Memento* m) {
        state = m->state;
        cout << "restored " << state.x << endl;
    }
};
```

Memento包含Originator的一个快照，所有函数私有，尤其不能构造

这个friend只允许Originator创建/修改Memento，其他类均不能进行任何访问

Originator对应刚才提到的编辑器，即包含敏感数据的原始类

Caretaker主要负责针对orig的快照创建和恢复

```cpp
class Caretaker {
    Originator* orig;
    vector<Memento*> history;
    public:
    Caretaker(Originator* o) : orig(o) {}
    void snapshot() {
        history.push_back(orig->save());
    }
    void undo() {
        orig->restore(history.back());
        history.pop_back();
    }
};

int main() {
    Originator* orig = new Originator();
    Caretaker ck(orig);
    orig->changeState(); //输出 changed to 1
    ck.snapshot();
    orig->changeState(); //输出 changed to 2
    ck.undo(); // 输出 restored 1
    return 0;
}
```

由于Memento全员私有，Caretaker无法查看/复制

类似但是相反的一种操作

- 有时候希望将某些数据作为只读查询结果返回给用户
  - 设DataBase类返回查询结果Result类
  - Result所存数据为公开，并且不再公开其他成员
  - Result类只有DataBase能构造、不可在其他地方创建/复制

```
class DataBase;
struct Result {
    const int a;
    const string b;
    const State s;
    private:
    Result(const int& _a, const string& _b, const State& _s) :
        a(_a), b(_b), s(_s) {}
    friend class DataBase;
};
```

所有数据成员都是public const

构造函数私有，除了友元DataBase
谁也无法构造/复制

```
struct DataBase {
    Result* query() {
        new Result(5, "abc", State());
    }
};
```

DataBase可以方便地返回只读
Result结果集

Observer

观察者模式

```
struct Subscriber {
    virtual void update(const string& mes) = 0;
};

struct SubscriberX {
    void update(const string& mes) {
        cout << mes << endl;
    }
};
```

接口

具体Subscriber类（可有很多）

- 场景：有一个信息发布源，很多其他类都想在信息发布时得到通知

```
class Publisher {
    set<Subscriber*> subscribers;
    public:
    void subscribe(Subscriber* sub) {
        subscribers.insert(sub);
    }
    void unsub(Subscriber* sub) {
        subscribers.erase(sub);
    }
    void notifySubscribers() {
        for (auto s = subscribers.begin(); s != subscribers.end(); s++) {
            (*s)->update("mes");
        }
    }
};
```

Publisher在合适的时候调用notifySubscribers来调用所有subscribers的update函数发送信息

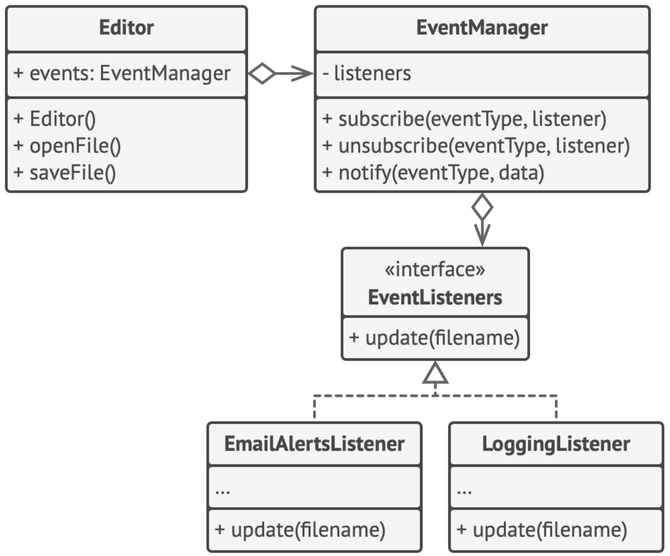
UI交互事件的例子

- Editor会将用户对自己的交互式更改发送给events
- 各种listener是事件的具体处理者

events是一个EventManager，类似刚刚的Publisher

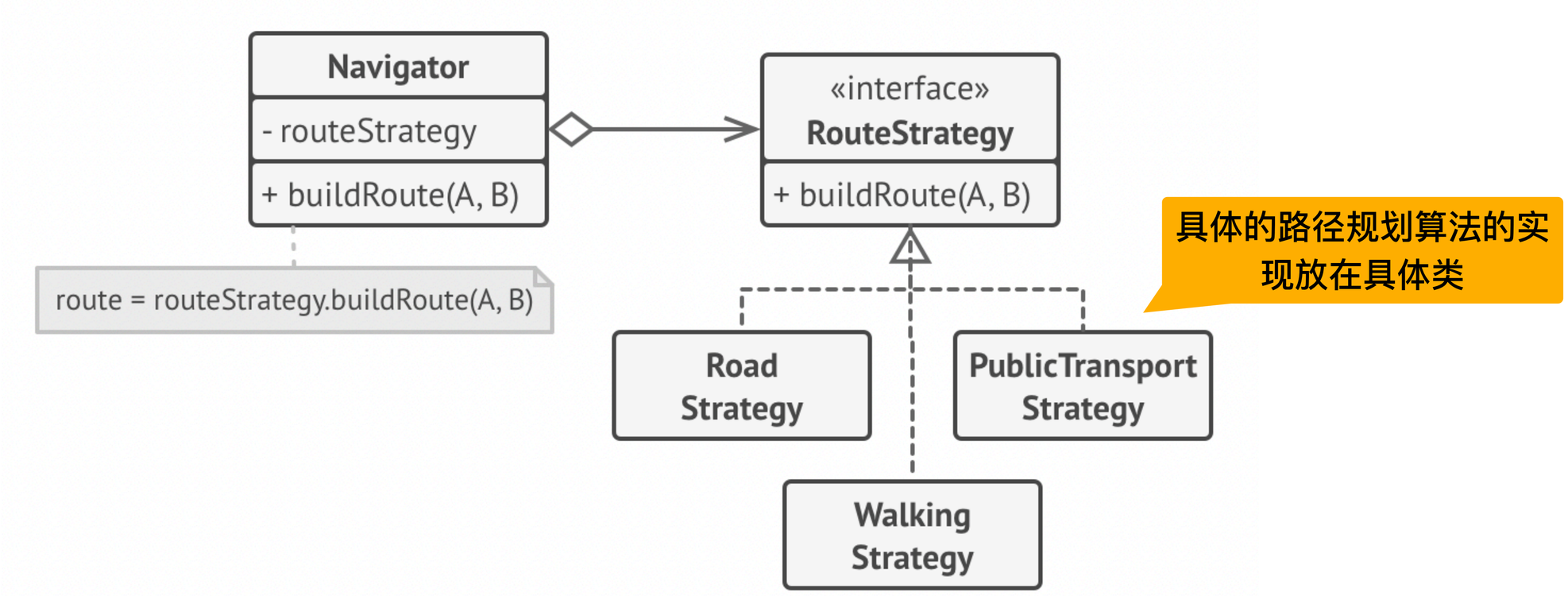
通过调用events的notify函数把时间类型evenType和数据data发送给若干listener

在Event Manager里将自己注册上，这样就实现了基于事件的UI编程



策略模式

- 场景：一个核心task可以有若干种实现方法，如何将这些方法作为模块加载
  - 例如，有一个地图App，一开始只有开车的路线规划功能
  - 后来又想加上自行车、步行、公交的规划功能
  - 比较差的做法是每次都把算法写在主要功能类里面，debug和复用都困难


Navigator
- routeStrategy
+ buildRoute(A, B)
«interface»
RouteStrategy
+ buildRoute(A, B)
route = routeStrategy.buildRoute(A, B)
Road Strategy
PublicTransport Strategy
Walking Strategy
具体的路径规划算法的实现放在具体类

Visitor

访问者模式

- 场景：有一个地图软件，维护一个巨大的图
  - 每个节点代表一个实体（例如城市），可以包含若干子节点
  - 节点之间的连边代表公路
- 新需求：将该图导出成XML
  - 需要每个节点实现一个export函数，然后利用动态绑定递归调用下去

然而……

- 要求对已有类改动尽量小，不允许加入export函数
- 并且在具体的节点类里面加入export XML的函数不一定有意义
  - 节点类主要就为了处理地理数据，export XML并不是主要功能
  - 再另外，以后要求导出其他格式，难道继续在节点类加函数?

访问者

- 将新行为，也就是export行为，放入到一个新的Visitor类里面
- 需要将执行操作的原始对象，也就是地图节点，传入Visitor
  - 之后Visitor根据传入的对象类型实现对应export函数
- 这样实现会遇到怎样的问题?

如何避免类型判断?

```
struct ExportVisitor {
    void doForCity(City c) {/*...*/}
    void doForHouse(House h) {/*...*/}
    void doForIndustry(Industry ind) {/*...*/}
};
```

ExportVisitor为每种节点定义一个完成export动作的函数

```
for (auto i = graph.begin(); i != graph.end(); i++) {
    if (typeof(*i) == City) visitor.doForCity(*i);
    if (typeof(*i) == House) visitor.doForHouse(*i);
    if (typeof(*i) == Industry) visitor.doForIndustry(*i);
}
```

但是具体使用的时候，迭代整个图节点的时候未必知道每个节点的类型，如何进行类型判断？？

双分派技巧

每支持一个新类XX，只需要加一个doForXX
且新类只需要v->doForXX(this)
不同功能可以通过不同具体Visitor具体类实现

```
struct Visitor {
    virtual void doForCity(City*) = 0;
    virtual void doForHouse(House*) = 0;
    virtual void doForIndustry(Industry*) = 0;
};
struct Node {
    virtual void accept(Visitor*) = 0;
};
```

- 解决方案：让具体的节点类“主动告诉”Visitor自己的类型
- 所有节点Node实现一个accept(Visitor)函数，调用Visitor里面对应的类型的函数

```
struct City : public Node {
    void accept(Visitor* v) {
        v->doForCity(this);
    }
}

struct House : public Node {
    void accept(Visitor* v) {
        v->doForHouse(this);
    }
};
struct Industry : public Node {
    void accept(Visitor* v) {
        v->doForIndustry(this);
    }
};
```

```
int main() {
    set<Node*> graph;
    Visitor* visitor;
    for (auto i = graph.begin(); i != graph.end(); i++) {
        (*i)->accept(visitor);
    }
    return 0;
}
```

针对每个类型的计算代码放Visitor里
*i依然是抽象类指针，但动态绑定到具体类
具体类accept了Visitor后调用visitor内自己类型的函数