适应设计模式

接口，实现，委托 组合（作为属性的对象） 消息传递

交给子类

生成实例

分开考虑

一致性

访问数据结构

简单化

管理状态

避免浪费

用类来表现

类图： 类 接口 实例 不仅仅是类

空心箭头 继承 子到父 super sub

类名 字段名 方法名

抽象类 方法是斜体 静态字段方法带下划线

接口 空心箭头 虚线 接口 实现 接口中都是抽象

聚合 一个类中有另一个类的对象实例 ：作为属性/字段 可以有多个实例 数组或向量

空心菱形

访问控制 +public -private #protect ~同一包

**关联、组合，聚合都是成员变量，但是意义不同 部分能否离开整体**

**依赖则是一种调用关系，在方法内使用其他类的对象 当调用使用静态数据或者方法，对象参数，局部变量创建对象在方法内时，就产生了类依赖**

**依赖本身就是关联？**

类之间的关联 黑色三角： 使用 生成 发送消息（提交者和观察者） 调用函数就是发送消息

时序图：运行关系 用的是实例 :类名

虚线是生命线，表示生命周期 方块是指活动（某个方法执行） 黑色实线箭头就是方法的调用，调用的是下一个实例的方法 虚线箭头指方法返回 （可能有返回值）

在方法执行块中进行调用其他实例方法

类库中使用了设计模式 遍历器 观察者 工厂方法 getInstance

Component 组合模式

设计模式是为了提高复用性

设计模式中要理解角色，每个类代表什么角色，如何互动

接口实际上就是函数，要被使用的函数称为接口,接口可以向上转型使用，但是因为接口一般是多继承，建议还是用父类转型

但是面向对象语言的核心是对象，我们主要还是关注对象的行为

所以

**分析例子的方法**

一。、迭代器

书架实现了Aggrate接口，该接口有iterator()方法（未实现），该方法创建迭代器对象

继承Iterator接口，实现了书架遍历器类

书架遍历器 中组合了书架对象 hasNext() next()

书架中组合了书本对象 books[] 具有获取书getBook 添加书 遍历等方法

书是单纯的数据类（对象） 只有name和 getname

在书架类中实现iterator()方法， new BookshelfIterator(this) 生成的遍历器中将自身组合进去 这样才可以遍历

hasNext next方法都和bookshelf对象相关 为了封装性不直接访问对方的书 ，用get方法

二、 适配器 现有的程序接口不对 简单来说函数不对，使其他打算调用某个通用函数的无法使用

类适配器

Banner类 showWithAser() showWithParen

Print 接口 printWeak() printStrong() 功能是一样的 但是名字不同

适配器 PrintBanner 继承Banner 实现Print 然后实现 printWeak(){showWithAser() }

对象适配器(委托)

把Banner对象给组合到PrintBanner中 printWeak(){banner.showWithAser() }

角色 目标 （目标接口方法） 请求者（cilent） 被适配 适配

三、模板方法，具体交给子类

铅笔临摹：完全一致

父类含有抽象方法 ，**定义了如何调用处理的流程框架，但是具体方法没实现**

循环显示五次字符串

AbstractDisplay open print close display 只有display实现了 display(){ open(); for(i<5)print(); close(); } 其他方法没实现

字符显示类 继承 设置属性 char 构造中获取输入参数给char 重写open print close

串显示

可以使逻辑流程通用化

类的层次：

子类角度：可以使用父类的方法 可以增加新的方法 可以覆盖重写父类的方法

父类：希望子类实现抽象方法，要求子类一定要实现

抽象类用来做模板方法很好，一部分实现一部分不实现，不能生成实例

四、工厂方法、 子类来决定生成具体哪个实例（与子类无关）

父类工厂决定生成的方式，但是对象的产生 交给子类，不同子类工厂决定如何创建对象，创建什么对象

使用了模板方法

制作身份证 ：根据输入的人名

产品类（抽象） use()

工厂类（抽象） create createProduct() resigisterProduct()

只实现了create (String onwer ) { createProduct(); registerProduct() return product; }

ID卡 owner

ID卡工厂 实现 createProduct new IDCard(owner)

工厂可以根据条件的不同生产不同对象， 类型是不一定的，工厂类和对象类没有任何关联

子类实现的意义，不同子类创建不同对象 但是一个工厂子类可以创建不同类的对象吗？用switch? 是好想法吗？

五、 单例 生成一个实例 任何情况都只会有一个实例（不怕不小心）

什么时候会不小心？ 并发编程 网络同时访问？

程序上表现出只有一个实例

Private Singleton() 构造函数禁止访问 静态方法.getInstance()

Private single = new Singleton() 初始化,内部可以访问 getInstance(){return single; }

类的初始化

可以全局的对象，然后 if(exsist) return?

六、 原型模式 复制生成实例

对象太多

保存和复制

不依赖具体的类

可克隆接口Cloneable

使用接口：就是调用接口的函数

在消息框中显示或者下划线显示字符串

产品接口 use createClone 复制实例 接口继承了Cloneable

管理器 类 create register注册,传参名字和product p 把name和p实例（向上转型）保存到一个结构中 create 从结构name获取到p 实例， 然后p.createClone()

所以最终实现的是无论创建消息框对象，还是下划线对象，都可以产生克隆

消息框类 实现产品接口 use 输出字符串 createClone 调用clone()方法

下划线 产品接口

原型接口，实现的原型 复制原型

Clone（）方法在java.lang.Object包中 ，Cloneable接口本身没实现任何方法，他只是个**标记接口**

属于浅复制

可以重写成深复制

七、 建造者模式 组装复杂实例

编写文档 标题，字符串，条目项目

Director 编剧,导演，使用不同类写不同文档 builder 组合了建造器 （使用了）

在构造函数中调用builder的方法 builder.makeTitle() –>builder.makeString() ->builder.makeItems()

Builder 建造器 抽象 makeTitle makeString makeItems close

TextBuilder String对象 添加各字段 返回字符串

HTMLBuilder 文件对象 PrintWriter 返回文件

建造中把各部分分开建造，再组装 不同类建造器建造的是不同内容（多态）

**谁知道什么（定义中） Main只知道 new Director() 方法 main中才传入了builder的真正类型**

**Director知道Builder,不知道是哪个子类**

可替换性

谁决定方法的调用顺序？ 模板方法中，父类实方法决定了子类方法的调用顺序

建造者模式，则是导演类的方法决定了如何调用建造方法（如何建造）

八、抽象工厂 关联零件组装产品

抽象工厂是把抽象零件组装成抽象产品，不关心实现，只关心接口

子类则是具体工厂

制作带链接页面

Factory 抽象工厂 getFactory通过子类类名反射得到对象 其他是抽象方法createLink createTray createPage

Item （项） 抽象零件 makeHtml link和tray的父类，用来多态。生成Link还是tray

Tray （托盘，或者是嵌套项） makeHtml add 含有多个link或者tray的容器（托盘） 有个数组tray

Add() 把对象加入数组

Link 只有链接的项 抽象零件 makeHtml 输入url

Page 由很多项组成页面 抽象零件 makeHtml add 产品 add向其中加入item（tray或者link） content数组

整体流程，工厂生成不同的link tray page ,tray中加入link和tray 形成多层项 ， page中加入link项或者tray项，形成最终的页面

ListFactory

ListTray

ListLink

ListPage

不同工厂生成不同样的零件

如何生成实例 new clone newInstance调用无参构造，需要try catch

九、桥接模式 ——类层次和功能实现层次分离

类的功能层次结构？

如何增加功能： 写个子类，然后增加新方法？ 如果想继续增加，子类的子类？**形成多个功能层**

类的实现层次结够？

不是增加功能，而是实现功能，重写实现方法。。。如果想修改功能，创建一个新类，重写实现 **形成多个实现层**

增加功能还是增加实现？ 是新增还是修改？

功能层和实现层可能混杂，因为都是子类，可以同时重写和新增

桥接模式，分成两个平行层， 然后连接

用组合代替继承

最上层是个抽象类 里面的方法是”抽象”的， 然后组合一个其他对象（该对象是抽象的，具体方法由子类实现） ，把自己的方法都委托给该对象

该抽象类的子类负责 添加功能，作为功能层

组合对象的子类负责实现，实现功能，作为实现层

继承时类之间是强关联，委托是弱关联

十、策略模式 整体替换算法

对象委托，多态

猜拳策略：这局赢了，下局一样的手势;上局手势算概率

Player 玩家 strategy 策略 nextHand win lose even

Hand 手势 三个实例 hand1 hand2 hand3 h.handvalue

Strategy nexthand 下一局手势 study

WinningStrategy

ProbStrategy

核心：使用委托来替换算法(语句) 调用同样的接口，使用不同的策略对象——function() {strategy.method() }

用多态实现整个算法的替换 也就是函数内部语句的替换

运行中切换策略

十一、组合模式 容器和内容一致

容器的递归结构 文件，目录 把容器和内容看作同一种东西,通用方法，递归实现

多个和单个的一致性，嵌套结构

方法是相同的

Entry(目录条目) 是file 和Directory的父类 getName getSize PrintList add

File 没有add ,调用add引发异常 name size new File(“name”) 创建文件

Directory

New Directory（） 创建文件夹

没有size 不过也可以实现吧 getSize是遍历所有文件大小相加 directory数组，保存目录条目（文件或目录）

getSize使用递归实现，遍历，加上下一个文件大小getSize( for(entry of directory)size+=entry.getSize() )

printList也是递归实现 printList( for(entry of directory)entry.printList() )

Exception

十二、 装饰的边框 和被装饰的

委托和组合

给对象增加装饰，增加功能

Display 抽象类 显示字符串 只实现了show() 模板方法

StringDisplay 实现的子类 getColumns getRows 获取行数 列数 show显示所有行字符串{ }

Border 继承display 同时组合了display对象 边框装饰的抽象

所有**方法委托给display对象进行**,同时做一些处理getConlumns(){ display.getColumns()+1 } 最后仍然使用show()模板方法显示

SideBoder 左右边框 实现

FullBoder 四方向边况

**装饰对象有些像是代理对象**欸 **委托**了其他对象的方法，然后进行了方法扩展

使用的是同样的接口

继承和委托的一致性：父子的一致， 自己和委托对象的一致，**接口一致**

十三、访问者模式 访问数据结构并处理

分离数据结构和 处理的类 实现不同的访问方式，避免访问方式太固定

Vistor 抽象 visit(file) visit(directory) 重载方法

ListVistor 实现

Element 数据结构接口 accept(vistor)

Entry 都实现了element接口

File accept(vistor){ vistor.visit(this) } 调用的是visit(File)

Directory 调用的是 visit(Directory)

这里比较复杂的是递归调用 visit会调用accept accept会调用visit 这是由于访问的嵌套

如果是文件直接读出来， 如果是文件夹就往下读

开闭原则

开放扩展，关闭修改 不修改现有类，扩展新类

十四、责任链模式

无法解决的处理交给下个对象 ，需要有个条件判断

不同人解决不同问题

Support 抽象 support(){if (! Resolve() ) next.support() } next setNext resolve 尝试解决

设置不同子类 写不同的解决方法 resolve()

所有对象都实例化，然后设置彼此的链条关系，setNext() .support(question) 开始执行

每个对象专注自己的工作

十五、外观模式facade 窗口模式

类越来越多，关联越来越复杂 调用顺序不好理清

提供高层接口， 调用下层接口 最后减少接口

十六、中介者模式

多个对象的关系，一个对象的操作影响多个其他对象，其他对象的操作又可以影响多个其他对象

需要用到其他对象的属性作为条件 需要调用对方的get

需要改变其他对象的属性 就需要调用对方的set

**对其他对象通信，先要对自己通信，先使用自己的一个方法，再调用对方的方法或者属性**

**其他对象往往是用属性，或者new，或者全局传入的，或者参数传入的**

**如果多对多的话，说明每个对象都要处理多个对象的引用**

可以让中介包含控制各种对象的方法。让本对象调用

中介一对多 其他对象一对一，不需要和多个对象通信（**不需要引入外部对象**）

由中介者来控制，收到一个

中介的api

对象的api

十七、观察者模式

观察者Obsever 这里观察数字的生成

DigitObser 观察的数字用数字显示（观察者）show()

GraphObsever 用图像显示 事件处理方法不同，多态 show()

数字生成器 （发送者） addObsever() notifyObsever() getnumber

可以发现这是个事件模式 发送者负责的是生成 发送的数据number，这个数据也可以是一个具有数据属性和方法的对象,称为事件event notifyObserver()发送事件，触发事件处理 { obsever[i].show(number ) }

每个监听对象都具有一个处理函数， 这里命名为show(number)

这是一种类似委托的机制 不过调用的是完全不相关的函数 ，而且传入了参数

灵活性：1、发送者往往会 维护一个监听者列表（数组），可以增加，删除监听者对象

2、监听对象的事件处理方法，有时候是可以设计成动态改变的，可以自定义使用哪个方法（使用高阶函数函数对象， 或者是策略模式修改内部算法等）

addEventLisener

3、 使用event对象，完全可以自定义event类型 ，在处理方法中定义如何解析event的数据，如何使用

十八、备忘录模式 memoto

实现撤销undo,重做redo,历史记录history，快照snapshot

记录某时间的实例属性，然后可恢复 ：定时保存，每次操作都保存。。。可以保存当前数据，也可以保存每一步的数据，形成数组 ，也可以保存每一次的操作，形成操作数组，保存最后的数据，作为存档

PS：游戏中，用结构或者对象数组保存状态很常见，比如所有任务状态数组，所有怪物的数组，所有物品的数组，所有地图事件的数组，所有npc状态的数组。。。相似对象就应该保存在一起，所有的飞机数组

游戏对象，只要是重复的就需要用数据结构组织起来，最好不要孤立。

某类：createMoment 创建一个备忘录对象 restoreMemento(mem)

备忘录对象：拥有和使用者同样的属性（只保存需要被记录的属性） add() get()

注意两个对象之间毫无关系，不是属性也不是内部new的，需要由被保存对象亲手创建，又用参数的方式保存进去

2、我们可以实现更加丰富的对备忘录操作的方法，比如从备忘中恢复

3、备忘录是直接访问数据 还是用get set方法？

4、备忘录完全可以认为是单纯的数据类 get set 属性

5、可以创建多个备忘录，或者备忘数组，实现更加复杂的恢复，多步撤销

6、**序列化，把对象直接保存为文件，方便恢复**

十九、状态模式

不使用状态模式的时候 状态用数字或者字符表示

然后不同方法内部对不同状态进行判断，执行不同的代码

使用状态模式，把条件判断的代码提取出来

不同状态对象 拥有同一接口 然后不同状态方法代码不同

用方法判断状态，还是用对象表示状态 这是一个多态的案例

方法本身也可以切换当前状态，这就形成了多态

**为什么要使用单例,这里要涉及的是对象生成形式，一个类如何生成和获取其他对象?在方法内new? 方法内传入参数（在外部new）如b.setA(a) ,初始化时new**

**外部new还算好管理 如果使用方法new的话，如果要多次执行方法，或者不同方法需要new同一对象？ 这就需要条件判断 当然如果生成对象的方法确保只执行一次，其他的方法都只引用这个对象那也未必需要单例**

状态迁移表，程序生成状态代码

二十、享元模式 ——共享对象

组合

共享一组同样的属性，避免浪费内存 每次new都要给各属性开空间

其中一个属性是对象（对象） 对象只是指针而已，占几字节空间 然后用方法来访问共享对象的属性

减少new次数的方法？ 可以继续用其他模板对象作为自己的属性，可以共用方法

问题来了，方法就不占用空间了吗？ 方法在执行时保存对象的属性到临时变量，使用的是栈空间，执行完就释放了。 最终返回的只是一个处理结果，不一定要占多少 空间

下一个问题，这样程序会更高效吗？ 内存占用少未必程序就会更高效，需要加载和处理的数据还是那么多。 函数语句没变少，重复执行次数没变少。。。不过内存不会撑爆了，也算提高性能了

还有其他资源：

代替的方法：每处理一个对象就把他删掉（除非以后不再用，而且在此之前你的内存没撑爆），这方法没啥意义，只用来清除

二十一、代理模式——必要时生成实例？

和装饰模式比较象 使用代理对象接口，委托了原对象的方法，但是同时又增加了自己的方法。。。。但是代理对象不一定需要原本的对象才能解决，他可能自己就能解决

组合

此时可以动态地new 或者获取对象（不创建对象可以省空间）

代理提速， 把耗时处理放在实际对象上，有需要再执行

http代理，代理服务器页面缓存，存在缓存就读缓存，没有缓存才读服务器

代理进行了哪些处理？

二十二、命令模式 命令就是事件event

进行操作—》生成命令对象 ->execute

命令可以做个命令集合 数组，对象等 添加删除命令，顺序执行

执行方法中调用，自己的方法语句 或者 其他对象的方法（委托），一般是要和其他对象通信的

发送者接受者

命令对象包含的信息

二十三模式 解释器模式

用迷你语言表述问题 需要翻译这种语言 修改问题，只需要修改迷你程序

Ebnf 定义树就是语法树

Lex获取标记token parse推导语法树

skipToken

node.parse 递归将子节点加入父节点 最后结点是终结符