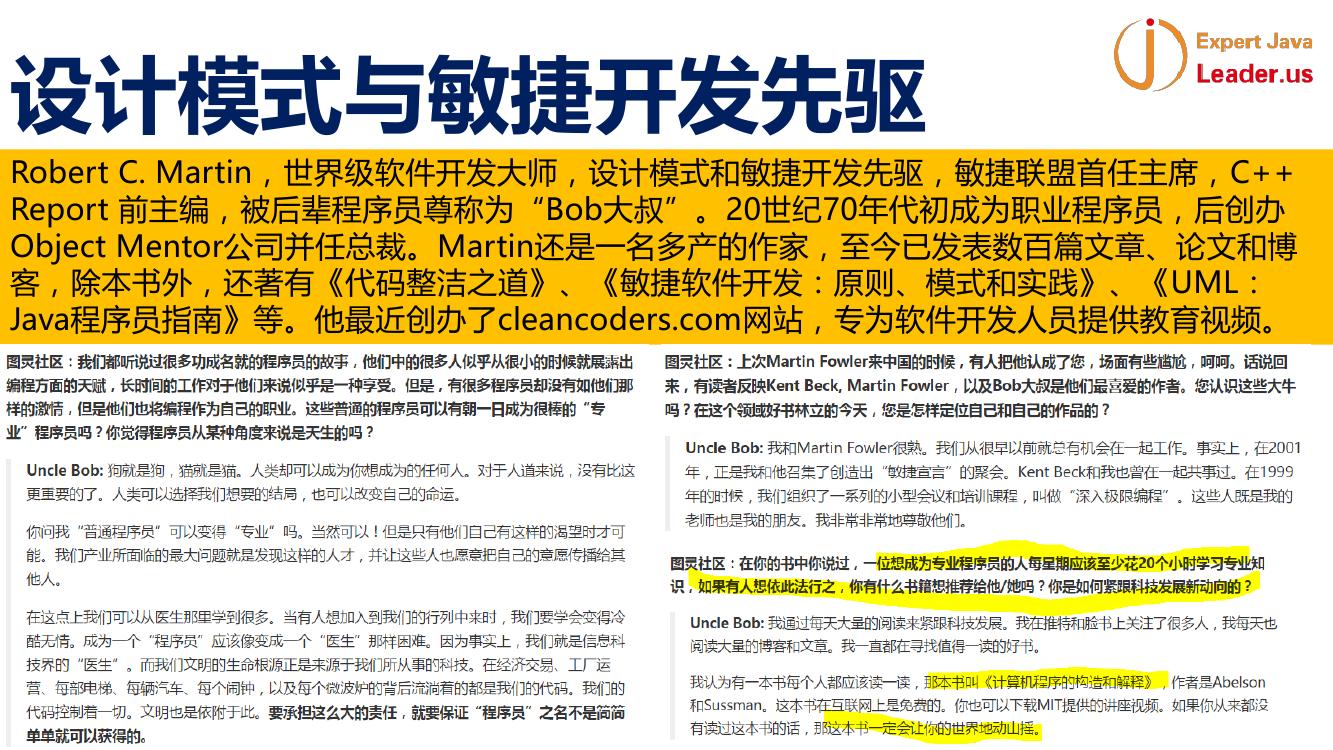




Spring是架构相关的一部分 相关的设计模式和设计原则都会一并介绍



Solid是几个人提出来的 一个人提出来的Solid的

这个人就是Martin ------

一个程序员不能只会编码 是一个很多经验和技能累积的

专业的程序员 每星期20个小时 ---- 每天大概3小时

很多大师都推荐持续学习 推荐了一本书 ---- **计算机程序的构造与解释**



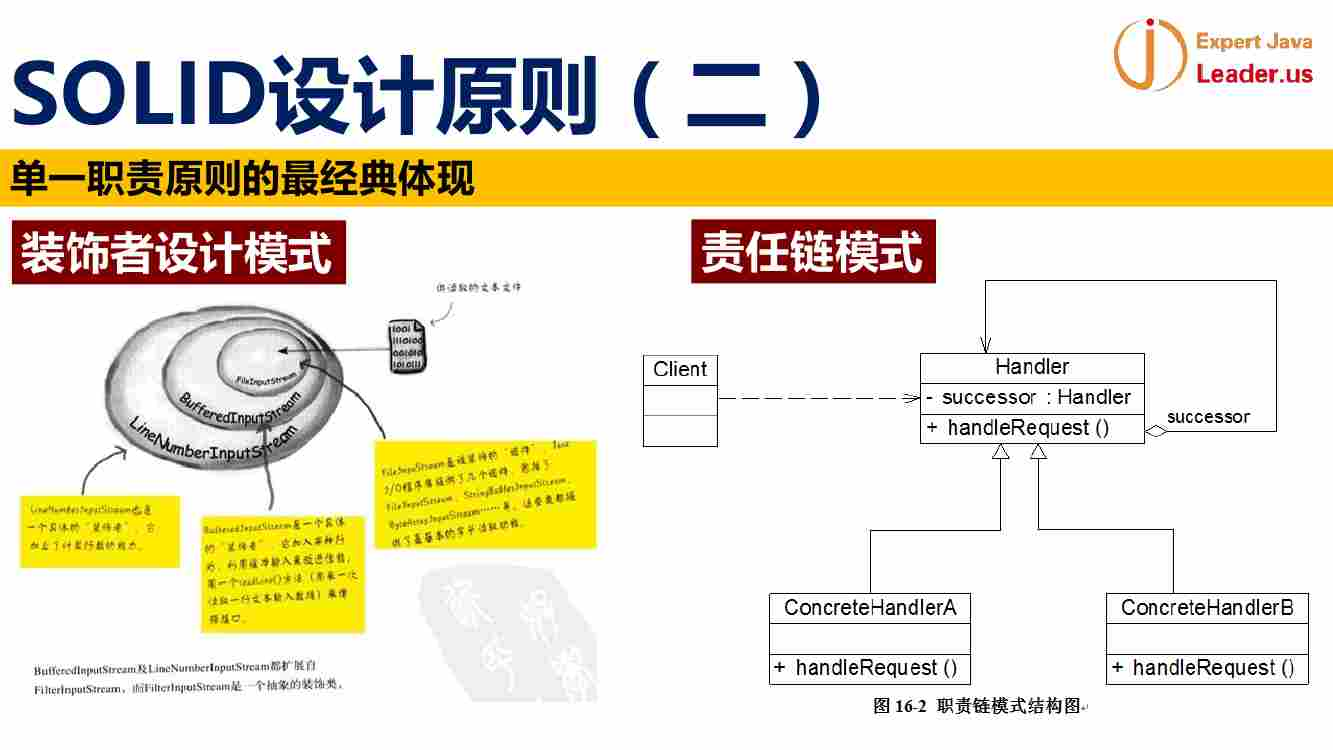
这5个设计原则是密切



单一职责原则

每个类 每个方法只做一件事 不能什么都做





装饰者模式 IO系列就是装饰者模式的典型表现

最底层的IO读写是有堵塞的 ---- 希望能有一次buffer

另外很多时候 读的时候 需要文本的解析 再往上 就是压缩 然后就是加密

---- 可以看出 从filteInputStream -> BufferedInputStream -> LineNumberInputStream 是每一个类都做一件事

BufferedInputStream就负责提供一个缓冲数组 去批量的读数据

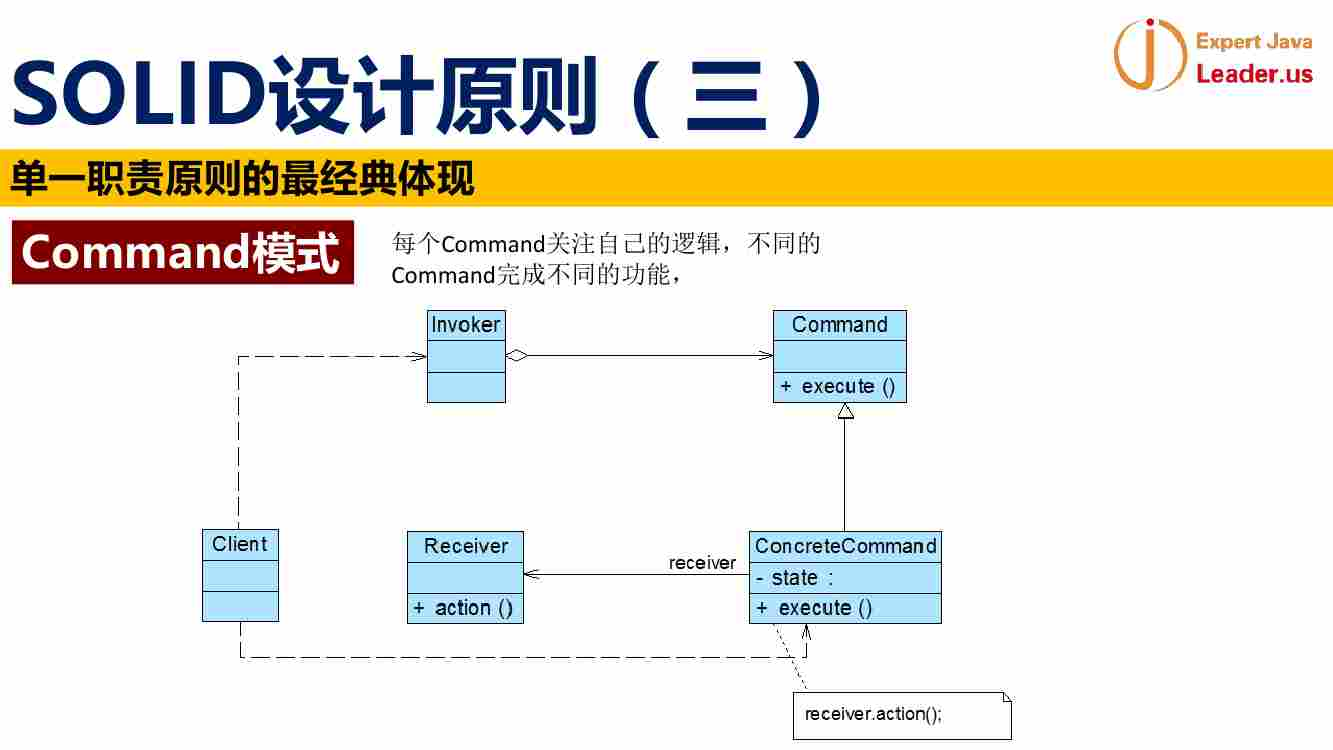
LineNumberInputStream就负责把一个一个的回车换行符解析出来

每一个只做一件事

跟装饰者模式类似的是责任链模式

有一个handle的接口 大家都去实现这个handle接口 每一个实现handle接口的都做不同的事情 就结束 ---- 这样就穿出一个链条 类似于装饰者模式 ---- 链条上面的每一个handle都要处理

关于这两个的表现区别 大家查查

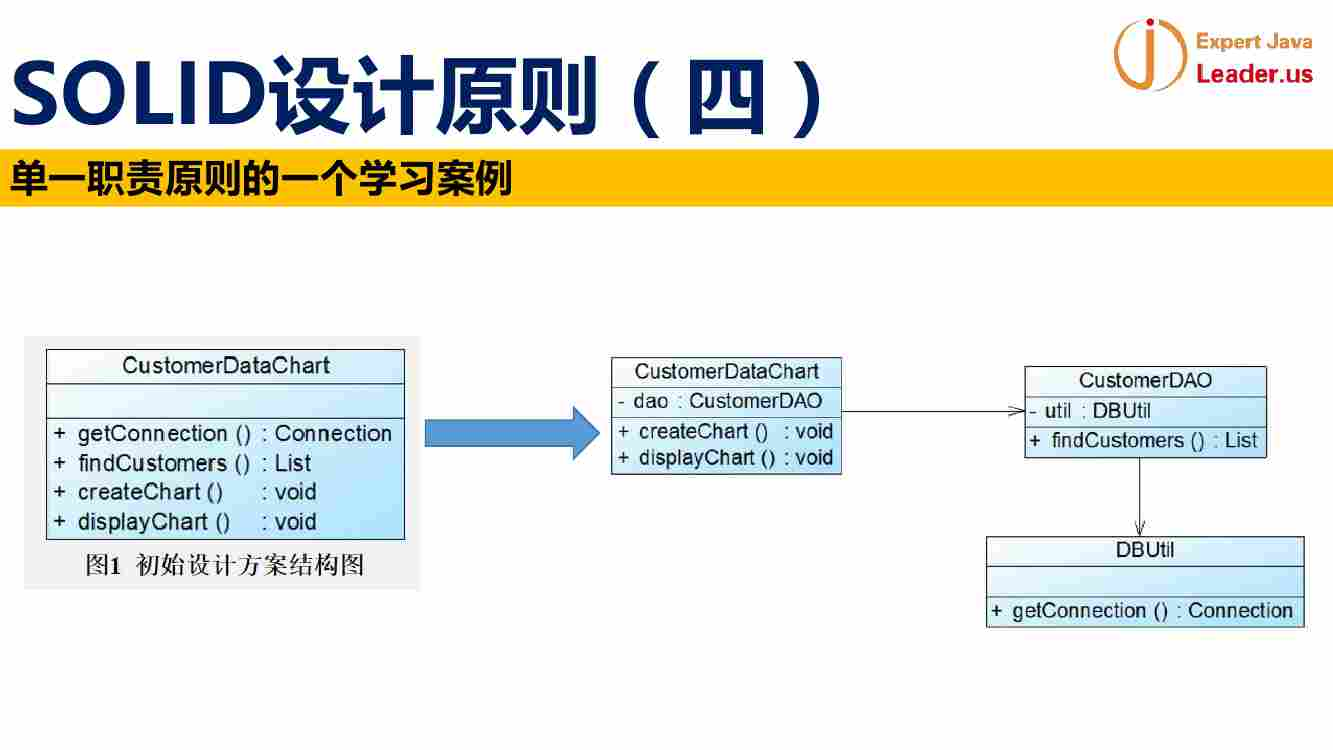


和责任链模式很类似 在网络编程中用的非常多

每一个报文是一个不同的请求 不同的请求被不同的command处理 处理完成返回一个数据流 ----- 那么客户端调用invoker这样的一个类 这个里面调用很多不同的Command ---- Invoker中会标记出来哪个报文请求应该分派到哪一个具体的Command上面

也就是说 这个模式有点复杂 ---- 大多数 就是invoker分发到ConcreteCommand 这些具体的Command执行完就返回给Client

这个模式建议仔细看看 能够自己写一个例子去验证 Command模式在协议处理用的非常多 每一个Command做单一职责



这是违反单一职责原则的案例

开始就是CustomerDataChart这个类 发现把很多领域的东西都融合到一起 ---- 关注的核心点 创建图表和展示图标 和其他的分裂出来

【实际上可以一下

这里面有获取数据库连接getConnection

查找客户信息 findCustomers

生成图表 createChart

绘制图标 displayChart

从最外面的展示来看：因为是单一职责 按照职责的不同 先剥离出图标的创建和展示是一类 这样 createChart & displayChart + [getConnection, findCustomers]----- 单一职责 那就和图表相关的两个方法 变成了最外面的一个类的两个方法 ----- 单一职责

那么这个类仅仅有这两个方法是不行的 需要有其他的来支持 但是 因为是单一职责 所以 辅助的两个东西不能以这个类的方法直接存在=🡺 另一种存在的形式 就是 被调用 ==》类的成员 这两个东西先整体打包 实际上就是UserDao ===》 这个CustomerDataChart内部就聚合了一个dao 被createChart和displayChart所调用 ==== 为什么不是继承？因为单一职责 你的具体子类都不是做这件事的 你的父类干嘛做这件事 ----- 所以 把这个从继承转到了聚合 ===== 一个CustomerDataChart里面 就聚合了一个dao

====== 下面 这个dao的类就是customerDao类

====== customerDao目前还是大杂烩 有两个方法 一个是findCustomers 另一个是getConnection 如果你认为这两个方法做的是一件事 那就没关系了 仿造CustomerDao里面 否则 看一下 应该是两件事 一个是获取数据 一个是为获取数据提供的connection

这样 CustomerDao只有一个findCustomers方法 这个类中需要聚合一个提供getConnection的对象 ---- 这就定义成util:DBUtil ----- 这样就出现一个新的类DBUtil 提供getConnection的方法

===== 核心思想 通过这个图的演变 就是 我们开始分不清的时候 可以把所有的功能都放到一个类图中 然后 分清楚 谁是终端的方法 ----- 利用单一职责分成一个类 ---- 这样 从外向内一层一层 就变成了一系列的类图了 ==== 这就是如何利用单一职责原则

】

----- CustomerDataChart：创建图标createChart()和displayChart()

很多设计都是从这里面开始 把一个大杂烩分清楚 分成多个 每一个最后都可以扩展

DButil可以分成不同的数据库的扩展

出发点就是单一职责



单一职责之后 后面涉及到的是修改 --- 修改的时候 怎么把握？因为很多时候需求是变化的 ---- 那么就要区分哪些可以改 哪些不可以改【因为单一职责原则已经把一个大杂烩分成了多个类 那么出现一个新的需求 总不能哪个地方都要修改吧！ 所以 单一职责原则之后 需要开闭原则进行完善后续任务的升级】

这个图就是如果你给一个布娃娃传一个衣服的时候 是不需要给这个布娃娃开膛手术的



Jacobson 提出了Rational的统一过程

上面那句英文的话：所有的系统都会在生命周期变化。Borne 由…携带的 运载的

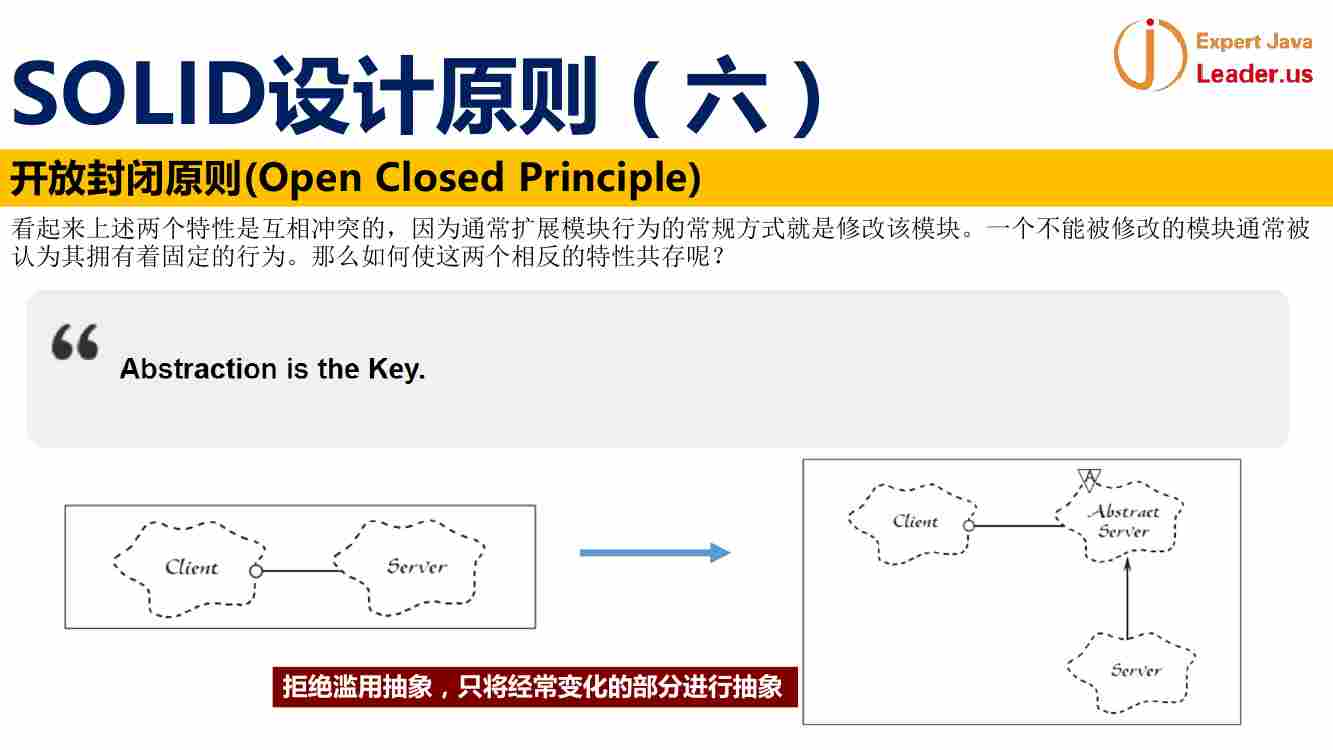
就是开发出来的系统都期望持续的时间比第一版长 ---- 意译成后面的翻译

开发一个软件 只要又一次迭代 必须有改变

有些软件几个月不符合要求 有些很长 ---- 也就是如何做一个设计稳定面对变化 让软件的生命周期更长 ----- 不是乱改 是稳定的变化

法国那位 --- 所有的实体都应该对扩展是开放的 对修改是封闭

------ 成员变量都设置为private ------ 这就是不open的



开放封闭原则 就是 一个类设计出来 来了新的需求 不是修改这个类的代码 而是扩展这个类。那么也是普遍被违反的规则

修改了 原来的测试都会有问题

开放封闭看起来是冲突的

要求设计方面 有一定的抽象能力 有一定的前瞻性 让这个模块具备一定的扩展能力

上面的图 --- 我们的设计就体现在这方面

通过抽象 让基层的东西不变 就是一开始 代码不变 ---- 后面通过扩展 实现延展

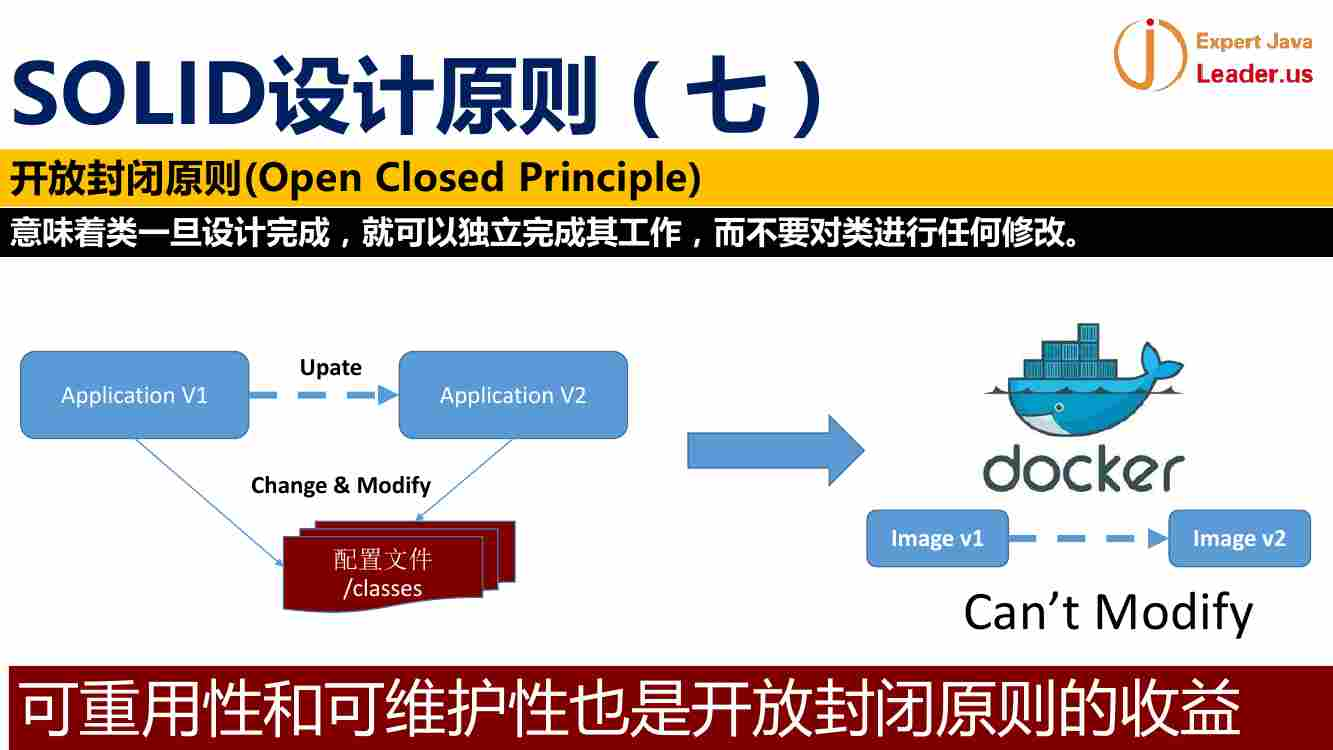
关键在于抽象 ----- 开闭原则给出了 我们要用设计模式 ---- 抽象是设计模式的指导

前面那句应为 “Abstraction is key”

解释一下上面的这个图：就是 这是典型的CS模式 开始的时候 Client和Server打交道 但是 随着时间推移 这样 Server有非常多的变化 这样 Client就有非常多的修改

但是 如果把这个Server抽象出来 客户端Client只和AbstractServer打交道 这样 修改仅仅是Server本身 后面的变化看不到 ---- 变化就是对Server进行扩展 Client和AbstractServer都不改变

开放封闭原则的特点 就是 避免通过修改源代码来实现扩展 真正的扩展要通过集成来实现



类一旦设计完 就可以独立工作 不要对类进行任何修改

左边的是传统的

版本升级的过程是一个update过程 这个update体现在部署这个过程 --- 修改的是配置文件 class 包等等 ---- 修改的过程不是标准控制过程 是一个人工过程 --- v1到v2不是一个清晰的变化 是一个渐进的过程 还附带了一个人工的 就是修改原来的配置文件 + 参数

右边的是docker部署的过程

Docker 做成两个镜像 image v1和image v2 ---- 然后这两个镜像是完全独立的 发新版就是替换成新的镜像即可

【我们的root项目 配置的修改等等 都是在jekenis完成的 最后的war都是独立的 替换就好】

开放封闭原则就可以理解为一种替换 理解为延展一种替换 升级一种替换 不是改代码 开放封闭原则 提高了可重用性和可维护性

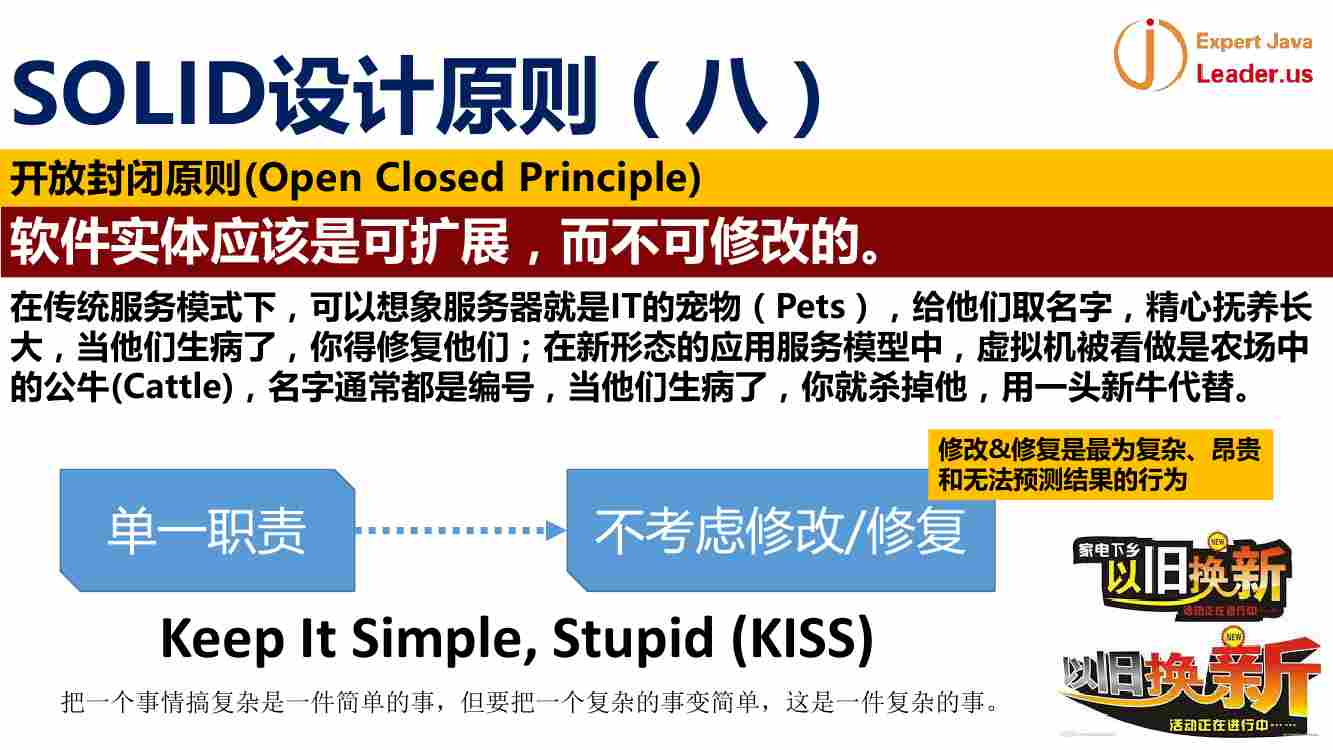
----- 可重用性 原来抽象的东西没有被改变 --- 可以继续使用（比如上面的client abstractserver）

----- 可维护性 因为不改动已有的客户端的代码 维护提升了

改代码是最差的维护 只有改的人知道内幕

开闭原则最重要的思想

如何设计一个框架 就是技能稳定 又能可扩展



传统模式下：服务器 软件 PETS 都由不同的IP 域名 ----- 上面的 大量消耗了人力 物理 扩展增加丰富的特性

新的就是替换模式 可就是把原来有问题的原件进行升级

单一职责也体现了 就是 不管别的东西 我只关心这件事 ----- 开闭原则进行了强化 如果要求改 就是 一旦要增加新的功能 不是在已有的东西上面打补丁 改源码

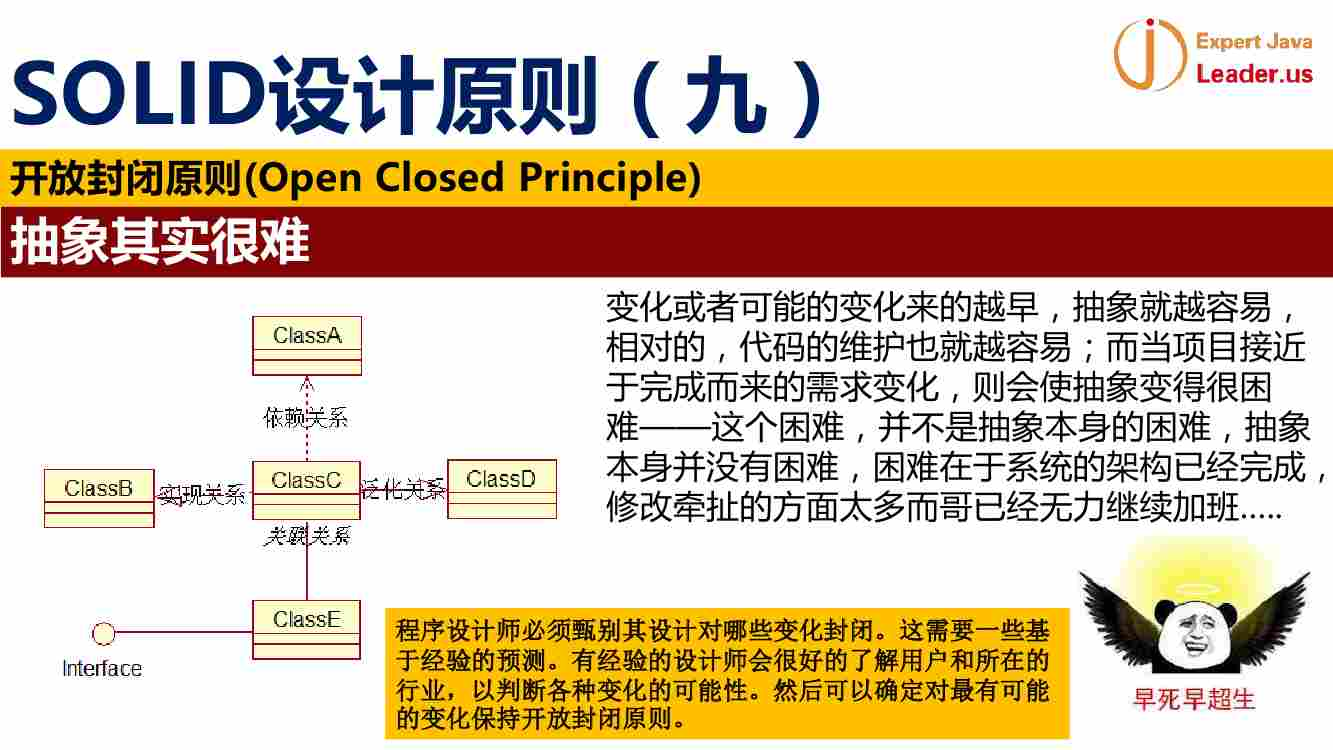
单一+开闭 让我们每一次的新变化都是一个独立的实体

【单一 确保功能模块是一个独立完整的 开闭 确保了这次的是一个全新的 而不是翻新老的出来的 】

到现在为止 自动化的框架在分布式越来越受欢迎 就是因为这种源码的修改 修复 和 装配 都是最复杂 最不可靠的 需要很专业的人 对全局任何细节都了解 才能修复 这个就是最不可靠的 ----- 我们就是替换他

原来方式下 考虑功能性 可靠性 这个吧很多功能柔和在一起 很不靠谱

Keep it simple, Stupid



抽象是一个思考过程 把变和不变分开 变得过程 进行扩展 不变的保证框架稳定

变得越早 --- 抽象越容易 ----- 这样可以把抽象做好

但是 快完事了 再来新的变化 抽象困难的 这时候 团队承担很麻烦

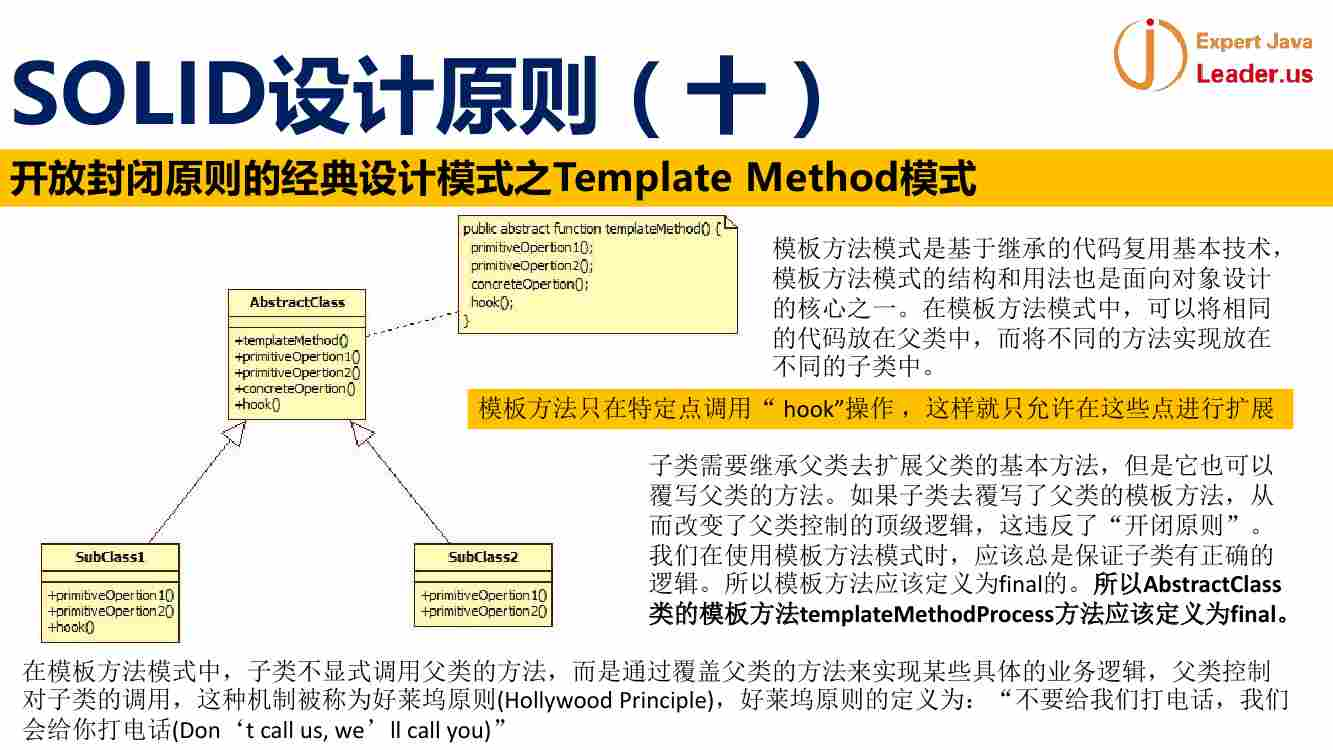
抽象要对这个领域很多认识 比如Google这方面的知识 + 熟练的架构能力 否则仅仅抽象出来的是这个项目的抽象 不是这个领域的抽象 就会以偏概全

抽象的关键点就是 哪些不变 哪些改变

到新的领域 要求查这个领域的实体对象 流程

建议掌握一定的UML 掌握一下 基本的UML要能看懂

UML建议看懂基本的类



这个模式就是把一些算法用模板的固定下来 某些点可以被调整 这个模式用得少 属于算法级别的东西 ---- 把复杂的算法步骤化 某些步骤子类可以扩展出来

不变的放到一个模板里面去 然后增加一个钩子的操作 ---- 这样子类可以在钩子上面进行扩展 变化

这就是一个模式 这个模式有些地方也用到了

Hook一般是protected的 可以扩展 并不一定要扩展

建议看看模板方法模式

这里面有一点提到的 子类其实可以覆盖父类 ----- 如果子类覆盖了父类 可能改变了流程控制 这样违反了开放原则的 ------ 子类不可能 也不应该修改父类的方法 --- 不是父类中的任意方法都能被改变 ---- 如果要覆盖 一定要明白覆盖要产生的结果

策略模式也是一个实现 策略模式在框架中用的多

缓存的策略 FIFO LRU LFU 这个是缓存中间件考虑的 ------ 这个由客户端来选择 ----- 这部分是可以变化 扩展 和自定义的 ---- 把这部分可以变化的变成一个策略

Policy通常代表一组数据 一组模式

策略模式通常包装算法 ---- 算法是被抽象的框架调用的

图中 数据是固定的 对于不同的算法 就是使用这个数据的策略是不一样的 ---- 不同策略会产生不同的影响 ---- 数据是通用的 处理逻辑是通用的 但是对数据特定的影响是变化的 ----- 那么这个就是策略算法 ---- 多个类在不同行为上面稍有不同 可以动态切换

对使用者是不需要知道的 是框架为了提高自身灵活性提供的设计模式

现在有三种方式可以选择 任何一个都可以被替换



Context封装了策略需要的上下文和数据

Context通常使用哈希表来传递数据 --- 传给Strategy

策略模式没有明显说调用参数的问题 ---- 参数都在Context中封装起来的 ---- 这里面有一个问题 就是策略比较多的时候 膨胀了 这时候可以使用组合模式   
还有 和工厂模式挂钩 如何创建策略对象

还有就是策略模式违反了迪米特法则 策略相对框架是底层的 ---- 上层要知道哪些策略可用 --- 这样就增加了使用者的负担 ----- 现在很多软件都非常智能 就是根据运行的数据 自适应调整 而不是让用户去选择 ---- 这就是违反了迪米特原则的坏处

仍然有很多策略 可以在运行的过程中动态的调整 自适应



【多看看PPT上面的内容】

34：51