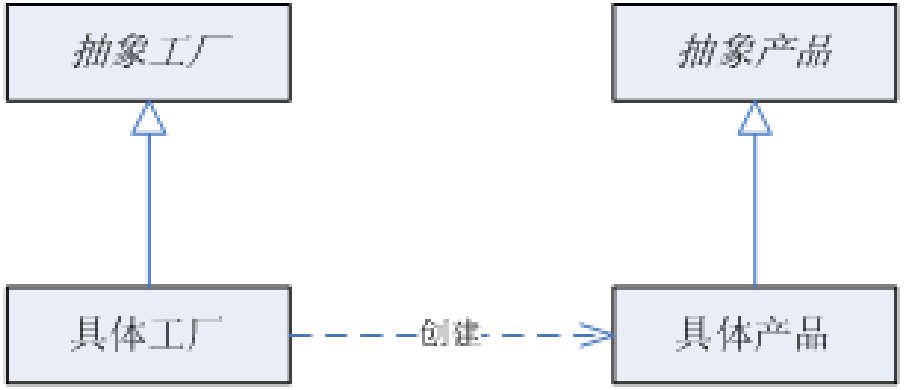
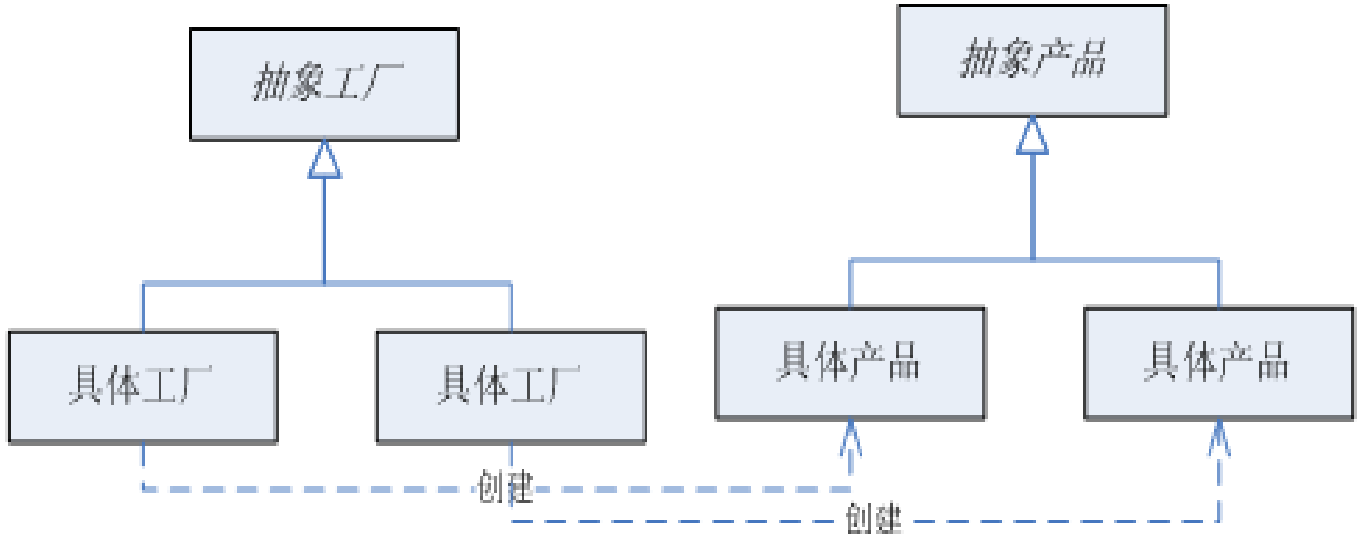
1）工厂方法 Factory Method

- 举例：请朋友去麦当劳吃汉堡，不同朋友有不同的口味，要每个都记住是一件烦人的事情，我一般采用Factory Method模式，带着朋友到服务员那儿，说“要一个汉堡”，具体要什么样的汉堡呢，让朋友直接跟服务员说就行了。



- Factory Method：抽象工厂类仅负责给出具体工厂类必须实现的接口，不负责产品的创建，将具体创建的工作交给子类去做，而不接触哪一个产品类应当被实例化这种细节。

- 优点：这种抽象的结果，使Factory Method模式可以用来允许系统不修改具体工厂角色的情况下引进新产品

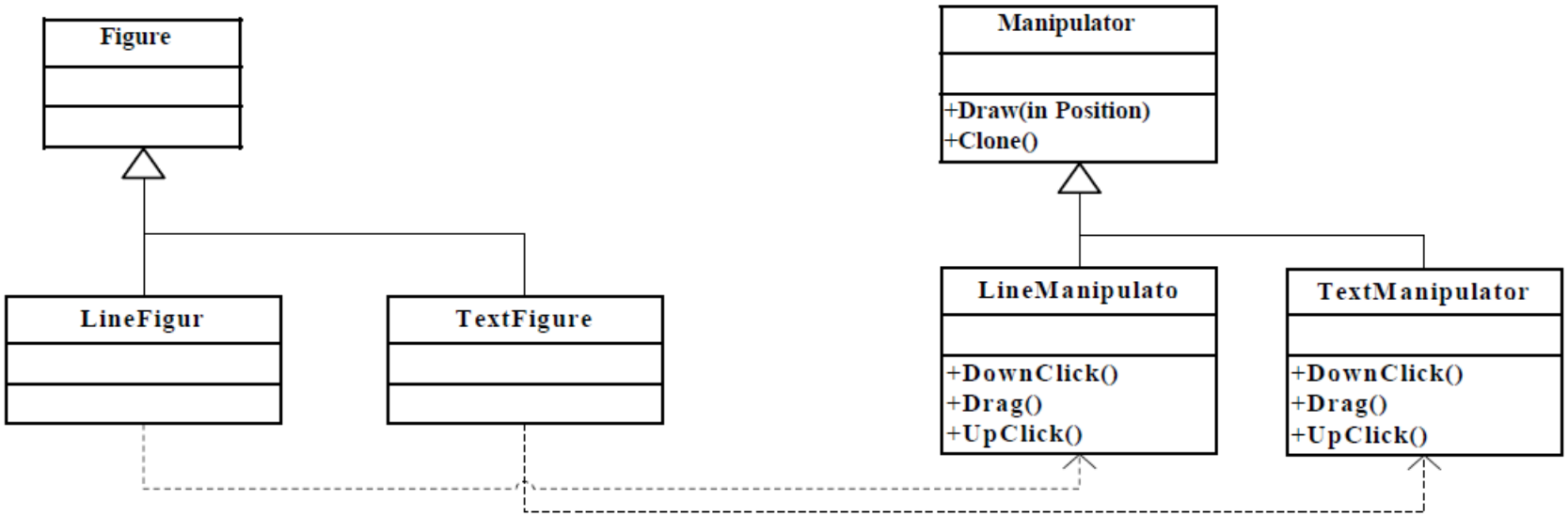


- 在Factory Method模式中，一般都有一个平行的等级结构，也就是说工厂和产品是对应的。抽象工厂对应抽象产品，具体工厂对应具体产品。

Factory Method模式解决的问题

- 两类问题：
  - 第一类问题：基于信息隐藏的原理，面向对象的软件框架通常使用抽象类来维护对象之间的关系。基于这样的框架，具体的应用系统如何扩展，在不修改框架的基础上，实现具体应用的功能?
  - 第一类问题的例子：
    - 背景：一个应用框架（例如：MFC），可以向用户显示多个文档。在这个框架中，两个主要的类Application和Document都是抽象类。客户基于这两个抽象类进行扩展，以实现具体应用所需的功能。
    - 问题：基于框架的应用程序开发人员基于这个框架开发一个绘图应用，我们定义DrawingApplication和DrawingDocument类。这个应用使用框架的功能——Application负责Document的创建。例如，当用户从菜单中选择Open或New菜单，Application则创建一个新的Document。问题是：这里需要创建的是一个具体的Document——DrawingDocument，但是，框架的编写者并不能预见这一点，它只知道那个抽象的Document类。如何能够构造一个独立于具体应用的框架?

- 两类问题（续）：
  - 第二类问题：
  - 程序中可能存在这样的平行结构：两组相关联的对象，一组中的某一个，与另一组中的一个存在关联关系——一组中一个对象如果存在，另一组中的关联对象也应该存在。如何表示这种关联关系？
  - 第二类问题的例子：一个可交互图形的例子



Factory Method模式的优点

- 多态性：客户代码可以做到与特定应用无关，适用于任何实体类。
- Provides hooks for subclasses。基类为factory method提供缺省实现，子类可以重写新的实现，也可以继承父类的实现。--加一层间接性，增加了灵活性。
- Connects parallel class hierarchies。
- 良好的封装性，代码结构清晰。
- 扩展性好，在增加产品类的情况下，只需要适当修改具体的工厂类或扩展一个工厂类，就可“拥抱变化”。
- 屏蔽产品类。产品类的实现如何变化，调用者都不需要关心，只需关心产品的接口，只要接口保持不变，系统中的上层模块就不会发生变化。
- 典型的解耦框架。高层模块只需要知道产品的抽象类，其他的实现类都不需要关心。

Factory Method模式的缺点

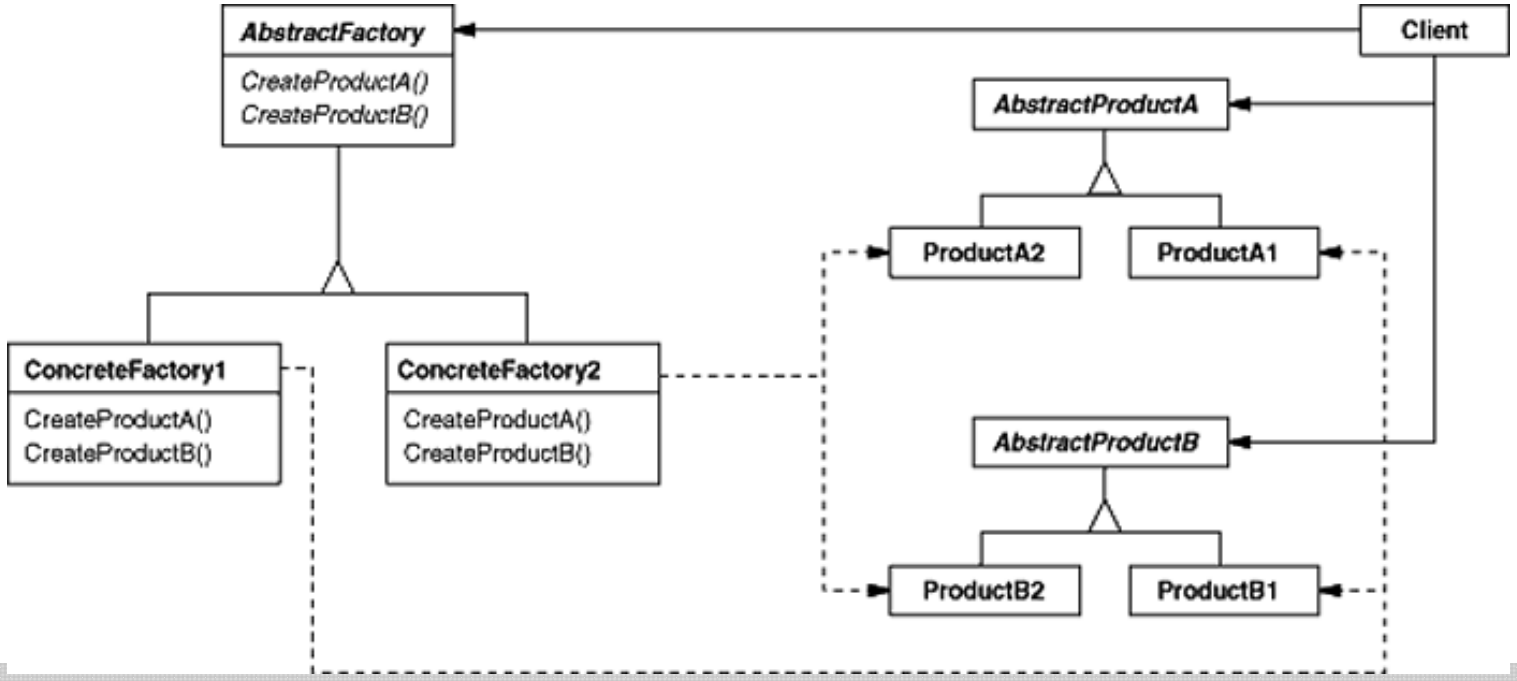
- 需要Creator和相应的子类作为factorymethod的载体，如果应用模型确实需要creator和子类存在，则很好；否则的话，需要增加一个类层次。

适用场景

- 在以下情况下可以使用工厂方法模式：
  - 一个类不知道它所需要的对象的类：在工厂方法模式中,客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可，具体的产品对象由具体工厂类创建；客户端需要知道创建具体产品的工厂类。
  - 一个类通过其子类来指定创建哪个对象：在工厂方法模式中，对于抽象工厂类只需要提供一个创建产品的接口，而由其子类来确定具体要创建的对象，利用面向对象的多态性和里氏代换原则，在程序运行时，子类对象将覆盖父类对象，从而使得系统更容易扩展。
  - 将创建对象的任务委托给多个工厂子类中的某一个，客户端在使用时可以无须关心是哪一个工厂子类创建产品子类，需要时再动态指定，可将具体工厂类的类名存储在配置文件或数据库中。

2）抽象工厂 Abstract Factory

- 举例
  - 麦当劳和肯德基除了汉堡，还卖鸡翅，虽然口味有所不同，但不管你带朋友去麦当劳或肯德基，只管向服务员说“来四个鸡翅”就行了。汉堡和鸡翅是不同产品簇，麦当劳和肯德基就是生产汉堡和鸡翅的Factory。



Abstract Factory模式解决的问题

- 将对象的创建与对象的使用分离
  - 例子：设计一个系统来显示和打印数据库中读出的图形，根据当前所使用硬件的配置来选择驱动：
    - 速度快的机器选择高分辨率的显示、打印驱动
    - 速度慢的选择低分辨率的驱动

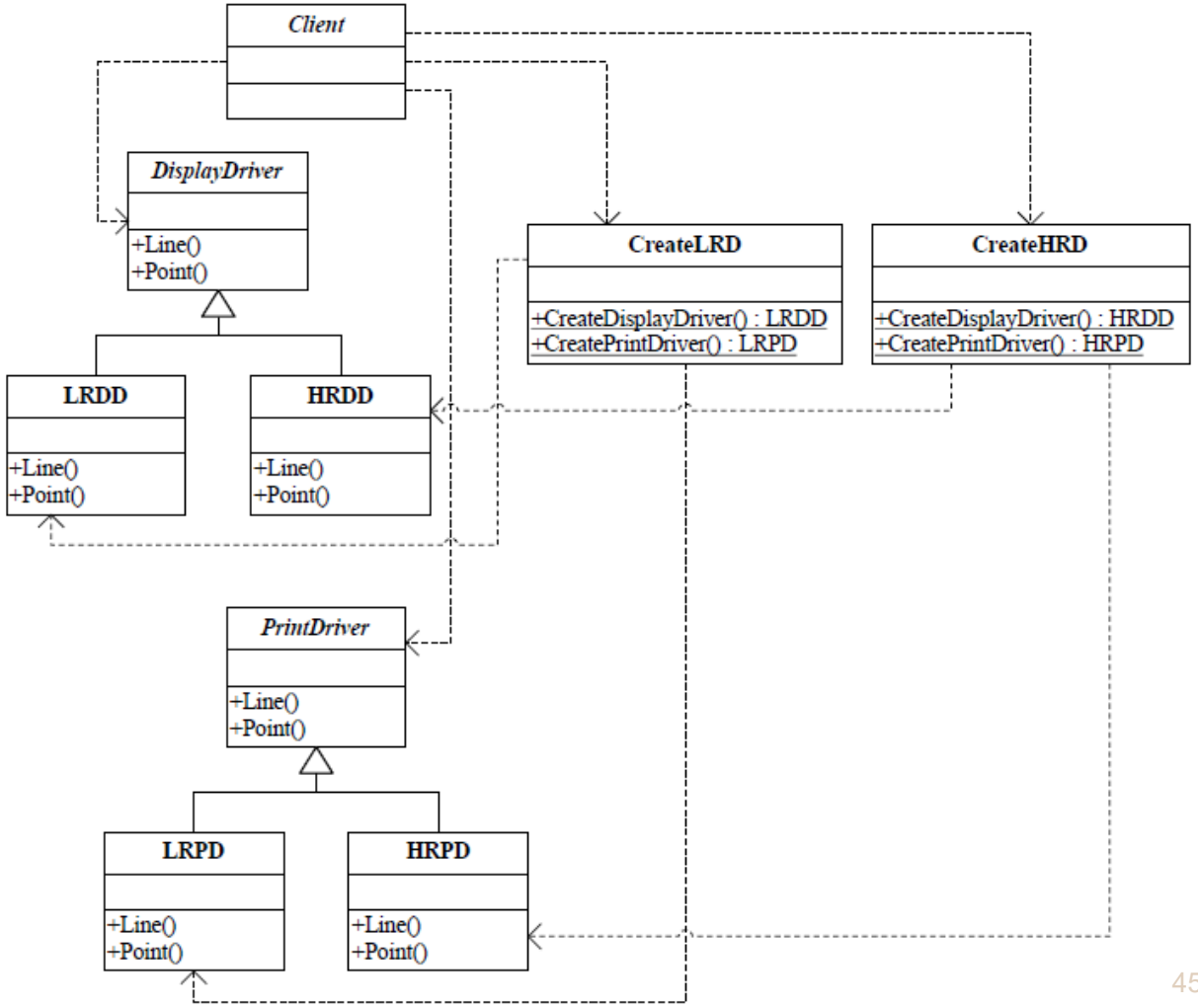
驱动类型	在低配置硬件情况下	在高配置硬件情况下
显示驱动	LRDD (Low-resolution display driver)	HRDD (High-resolution display driver)
打印驱动	LPPD (Low-resolution print driver)	HPPD (High-resolution print driver)

Abstract Factory

更好的解决方案

对象创建与对象使用的关注点已经分离：用另外一个对象来专门负责对象的生成

——产生了工厂类的概念



Abstract Factory模式的优缺点

- 优点：
  - 分离了具体的类，一个工厂封装创建产品对象的责任和过程，它将客户与类的实现分离。
  - 易于变换产品系列，只需改变具体的工厂就可以使用不同的产品配置。
  - 有利于产品的一致性，当一个系列中的产品对象被设计成一起工作时，一个应用一次只能使用同一个系列中的对象。
- 缺点：
  - 难以支持新的产品等级结构(AbstractProductX)，支持新的产品等级结构就要扩展抽象工厂接口。

适用场景

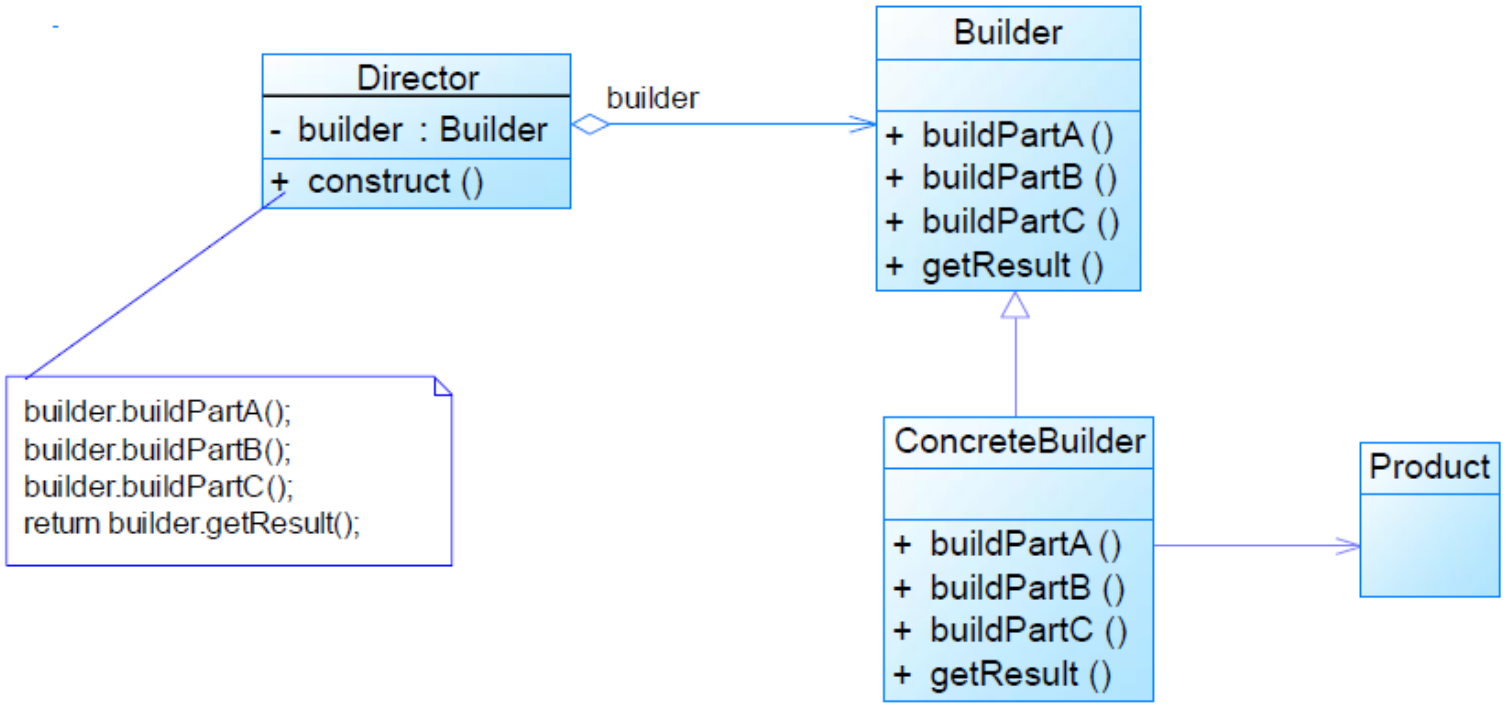
- 在以下情况下应当考虑使用抽象工厂模式：
  - 系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有形态的工厂模式都重要。
  - 系统有多于一个的产品族，而系统只消费其中某一产品族。
  - 同属于同一个产品族的产品是在一起使用的，这一约束必须在系统的设计中体现出来。
  - 系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于实现。
- 应用场景
  - 支持多种观感标准的用户界面工具箱（**Kit**）。
  - 游戏开发中的多风格系列场景，比如道路，房屋，管道等。
  - ……

3） 建造者模式 Builder

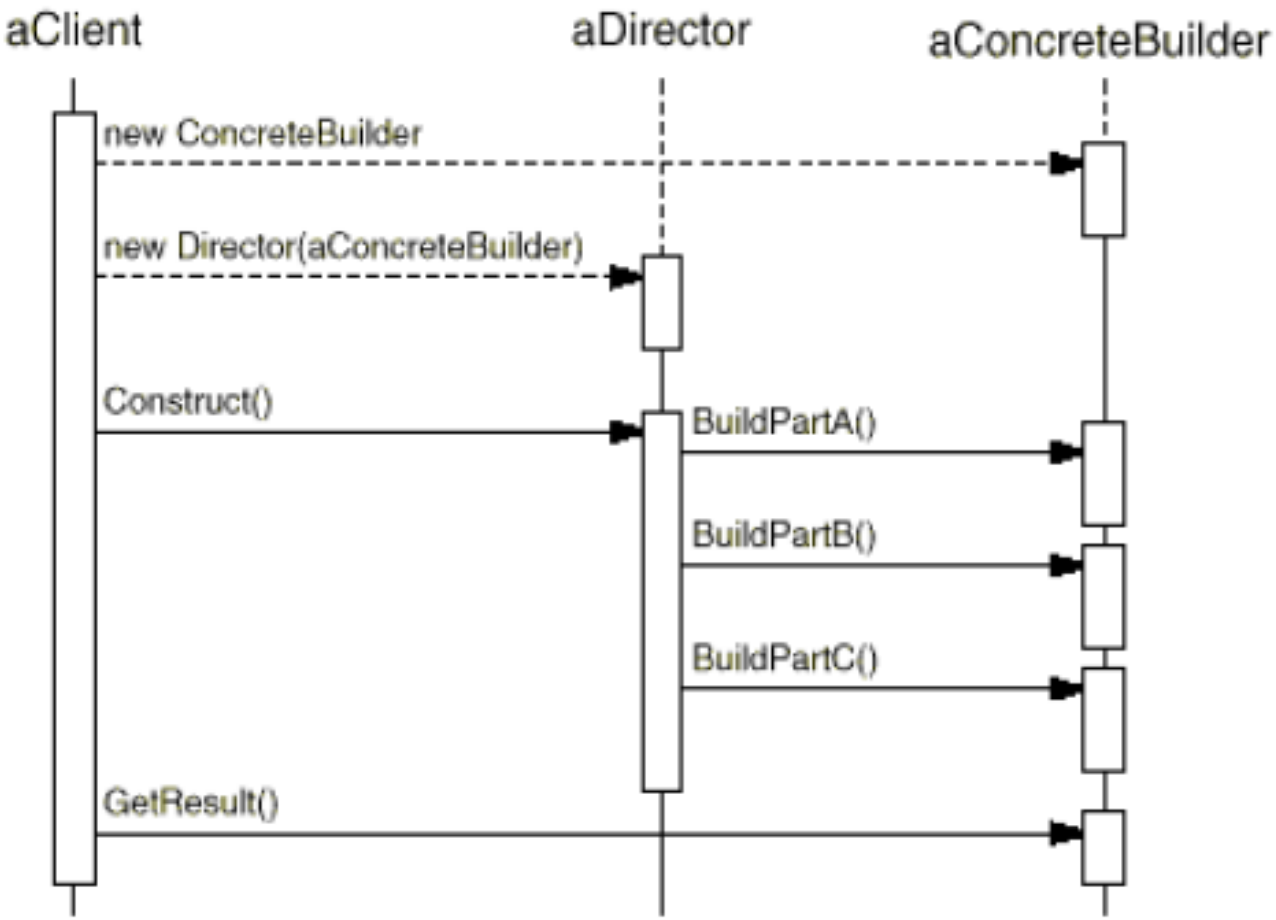
- 举例：假设要组装一辆自行车，并且自行车就是车轮和车架组成，自行车有永久牌和凤凰牌
  - Builder对应于组装自行车所使用的车轮和车架
  - ConcreteBuilder1对应于永久牌车轮和车架，同时可以返回一辆永久牌自行车，ConcreteBuilder2对应于凤凰牌车轮和车架，同时可以返回一辆凤凰牌自行车
  - Product对应于自行车
  - Director表示组装过程
- 我们再来理解这句话：“在软件系统中，有时候面临着“一个复杂对象”的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法确相对稳定。”
- 自行车就是“一个复杂对象”，它有车轮和车架组成，但不管车轮和车架这两个部件怎么变化，生产一辆自行车的过程是不会变的，即组装过程是不会变的。

Builder模式的类图

- **Builder模式**将复杂对象的构建与其表示分离，以使得相同的构建过程可以创建不同的对象表示。

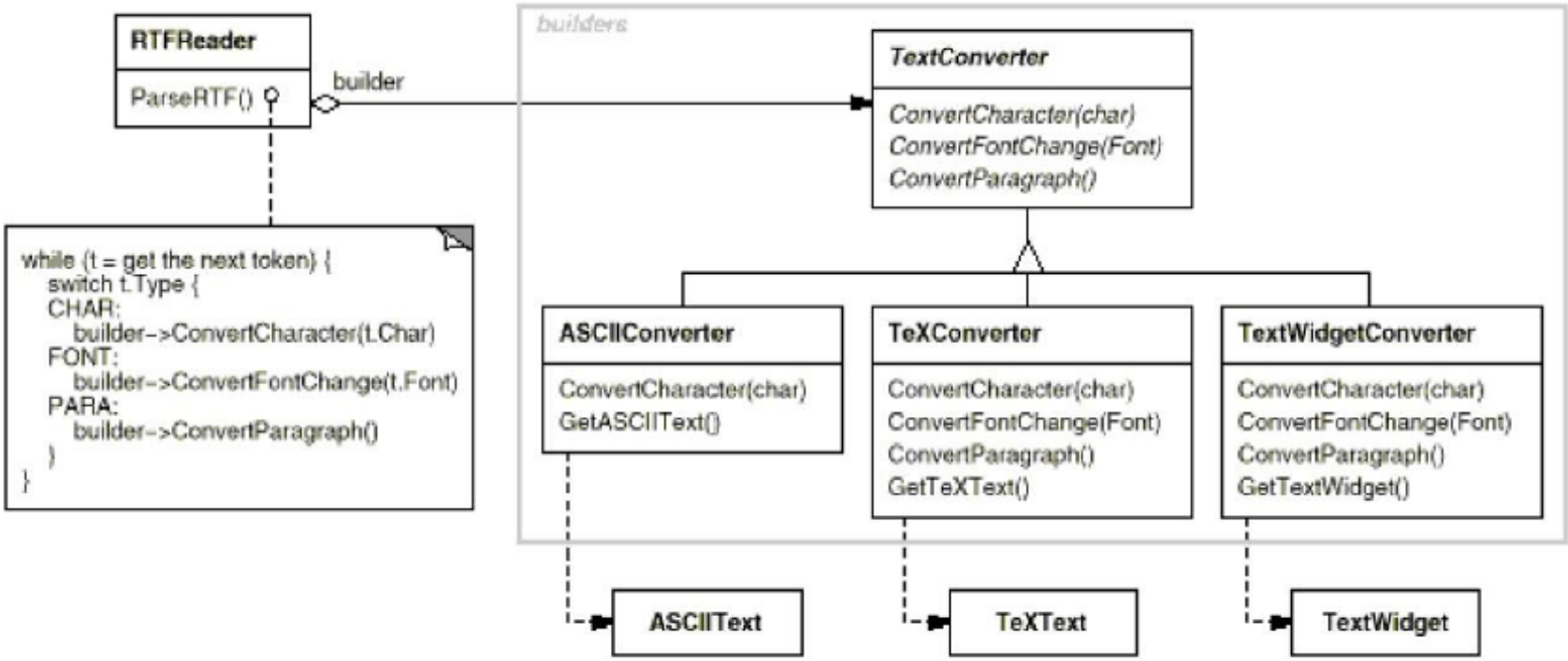


Builder模式的时序图



例子：RTF阅读器

- 一个RTF(RichTextFormat) 格式的阅读器应能将RTF转换为多种正文格式。该阅读器可以将RTF文挡转换成普通ASCII文本或转换成一个能以交互方式编辑的正文窗口组件。但问题在于可能转换的正文格式数目是无限的。因此要能够很容易实现新的转换的增加，同时却不改变RTF阅读器。
- 一个解决办法是用一个可以将RTF转换成另一种正文表示的TextConverter对象配置这个RTFReader类。当RTFReader对RTF文档进行语法分析时，它使用TextConverter去做转换。无论何时RTFReader识别了一个RTF标记(或是普通正文或是一个RTF控制字)，它都发送一个请求给TextConverter去转换这个标记。TextConverter对象负责进行数据转换以及用特定格式表示该标记，如下图所示。



- 每种转换器类将创建和装配一个复杂对象的机制隐含在抽象接口的后面。转换器独立于阅读器，阅读器负责对一个**RTF**文档进行语法分析。
- TextConvert的子类对不同转换和不同格式进行特殊处理。例如，一个ASCIIConverter只负责转换普通文本，而忽略其他转换请求。另一方面，一个TeXConverter将会为实现对所有请求的操作，以便生成一个获取正文中所有风格信息的TEX表示。一个TextWidgetConverter将生成一个复杂的用户界面对象以便用户浏览和编辑正文。

Builder模式的优点

- 在建造者模式中，客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的产品对象。
- 每一个具体建造者都相对独立，而与其他的具体建造者无关，因此可以很方便地替换具体建造者或增加新的具体建造者，用户使用不同的具体建造者即可得到不同的产品对象。
- 可以更加精细地控制产品的创建过程。将复杂产品的创建步骤分解在不同的方法中，使得创建过程更加清晰，也更方便使用程序来控制创建过程。
- 增加新的具体建造者无须修改原有类库的代码，指挥者类针对抽象建造者类编程，系统扩展方便，符合“开闭原则”。

Builder模式的缺点

- 建造者模式所创建的产品一般具有较多的共同点，其组成部分相似，如果产品之间的差异性很大，则不适合使用建造者模式，因此其使用范围受到一定的限制。
- 如果产品的内部变化复杂，可能会导致需要定义很多具体建造者类来实现这种变化，导致系统变得很庞大。

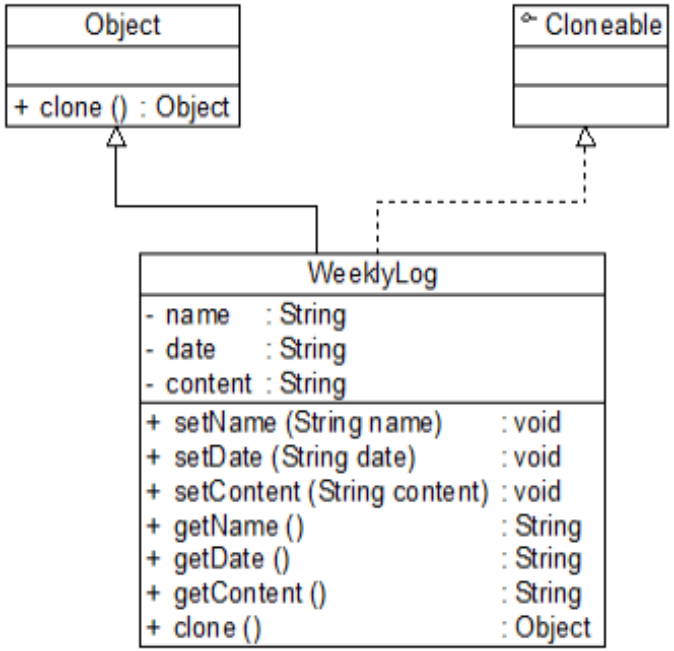
适用场景

- 需要生成的产品对象有复杂的内部结构。每一个内部成分本身可以是对象，也可以是一个对象的一个组成部分。
- 需要生成的产品对象的属性相互依赖。建造者模式可以强制实行一种分步骤进行的建造过程
  - 如果产品对象的一个属性必须在另一个属性被赋值之后才可以被赋值，使用建造模式就是一个很好的设计思想。
  - 有时产品对象的属性并无彼此依赖的关系，但在产品的属性没有被确定之前，产品对象不能使用，这时产品对象的实例化、属性的赋值和使用仍然是分步骤进行的，建造模式有意义。
- 在对象创建过程中会使用到系统中的其他一些对象，这些对象在产品对象的创建过程中不易得到。
- 示例：在很多游戏软件中，地图包括天空、地面、背景等组成部分，人物角色包括人体、服装、装备等组成部分，可以使用建造者模式对其进行设计，通过不同的具体建造者创建不同类型的地图或人物。

4）原型模式 Prototype

- 举例：
  - 某公司使用自行开发的一套OA系统进行日常工作办理，但在使用过程中，员工对工作周报的创建和编写模块产生了抱怨。由于某些岗位每周工作存在重复性，工作周报内容都大同小异，这些周报只有一些小地方存在差异，但是现行系统每周默认创建的周报都是空白报表，用户只能通过重新输入或不断复制粘贴来填写重复的周报内容，工作效率低。
  - 如何快速创建相同或者相似的工作周报，成为公司OA开发人员面临的问题。

- 使用原型模式来实现工作周报的快速创建：



- 图中，WeeklyLog充当具体原型类，Object类充当抽象原型类，clone()方法为原型方法。

浅克隆和深克隆

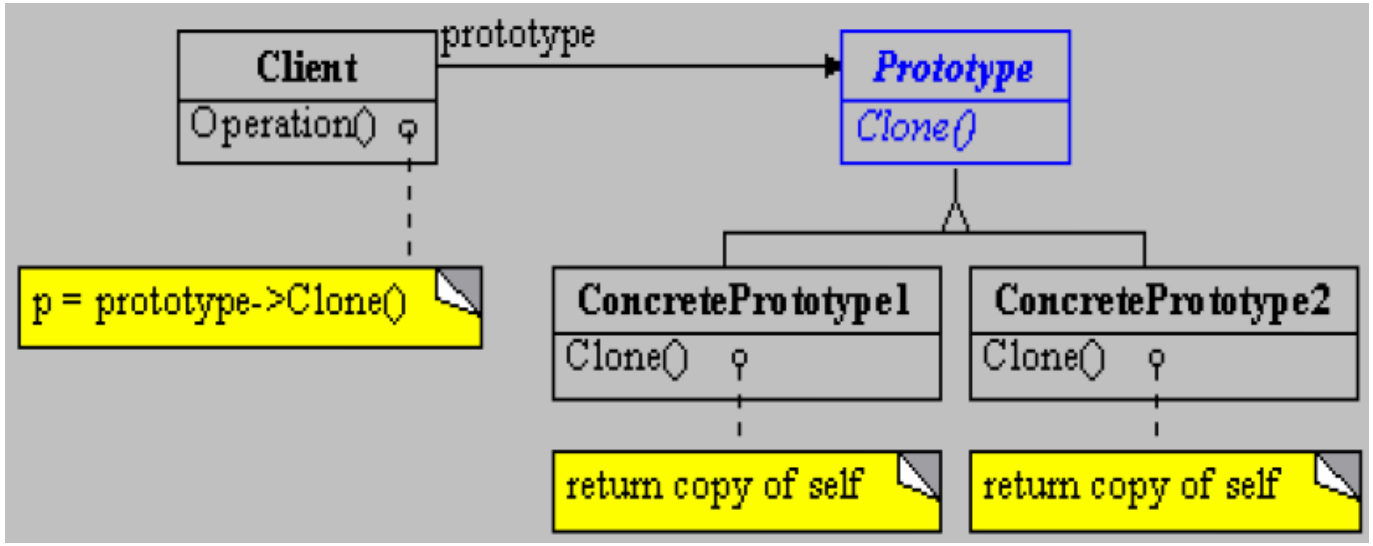
- 但：有些工作周报带有附件
- 　如何才能实现周报和附件的同时复制呢?
- 根据其成员对象是否也克隆，分为两种不同的克隆方法：
  - 在**浅克隆**(Shallow Clone)中，当对象被复制时它所包含的成员对象却没有被复制。
  - 在**深克隆**(Deep Clone)中，除了对象本身被复制外，对象包含的引用也被复制，即其中的成员对象也将复制。
- 在Java语言中，通过覆盖Object类的clone()方法可以实现浅克隆；如果需要实现深克隆，可以通过序列化等方式来实现。

Prototype

- **Intent**
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. (用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象)
- **Motivation**
  - 以一个已有的对象作为原型，通过它来创建新的对象。在增加新的对象的时候，新对象的细节创建工作由自己来负责，从而使新对象的创建过程与框架隔离开来

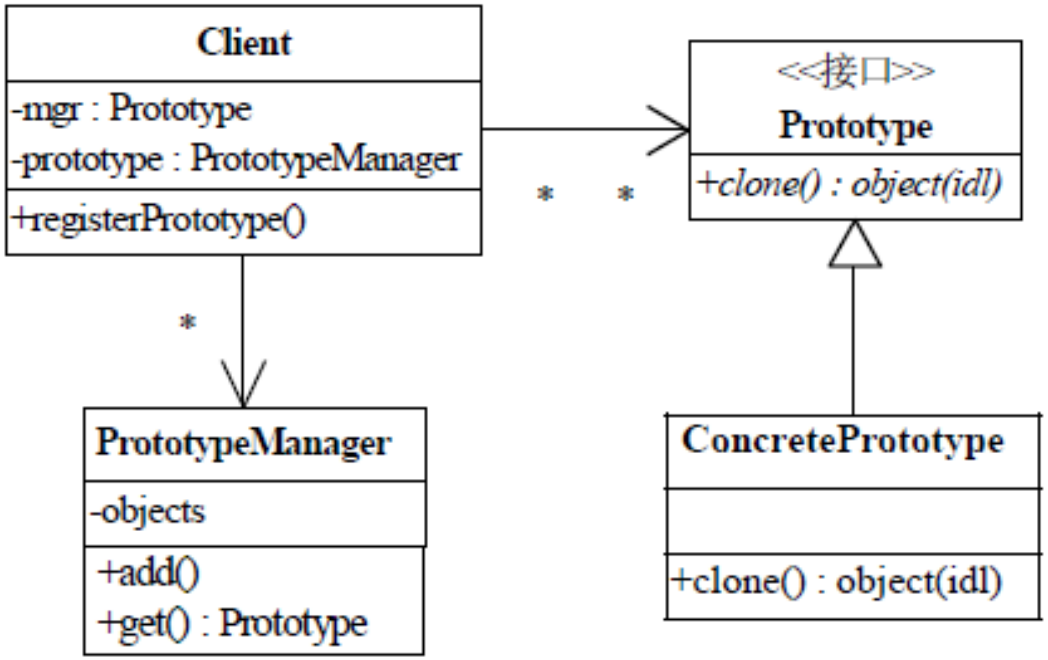
Solution

1、简单形式的原型模式



- Prototype: 声明一个克隆自身的接口
- ConcretePrototype: 实现一个克隆自身的接口
- Client:让一个原型克隆自身从而创建一个新的对象。

2、登记形式的原型模式：使用一个原型管理器



当一个系统中原型数目不固定时，要保持一个可用原型的注册表

- Client :客户端向管理员提出创建对象的请求；
- Prototype Manager: 创建具体原型类的对象，并记录每一个被创建的对象

Prototype模式的优点

- 原型模式允许动态地增加或减少产品类。由于创建产品类实例的方法是产品类内部具有的，因此增加新产品对整个结构没有影响。
- 原型模式提供简化的创建结构。工厂方法常需要有一个与产品类相同的等级结构，而原型模式不需要。对Java设计者，原型模式有其特有方便之处，Java语言已将原型模式设计到语言模型里。如果善于利用原型模式和Java语言特点，可事半功倍。
- 具有给一个应用软件加载新功能的能力。如：一个分析web服务器的记录文件的应用软件，针对每一种记录文件格式，都可以由一个相应的“格式类”负责。如果出现了应用软件所不支持的新的web服务器，只需提供一个格式类的克隆，并在客户端登记即可，不必给每个软件的用户提供一个全新的软件包。
- 产品类不需要非得有任何事先确定的等级结构，因为原型模式适用于任何的等级结构。

Prototype模式的缺点

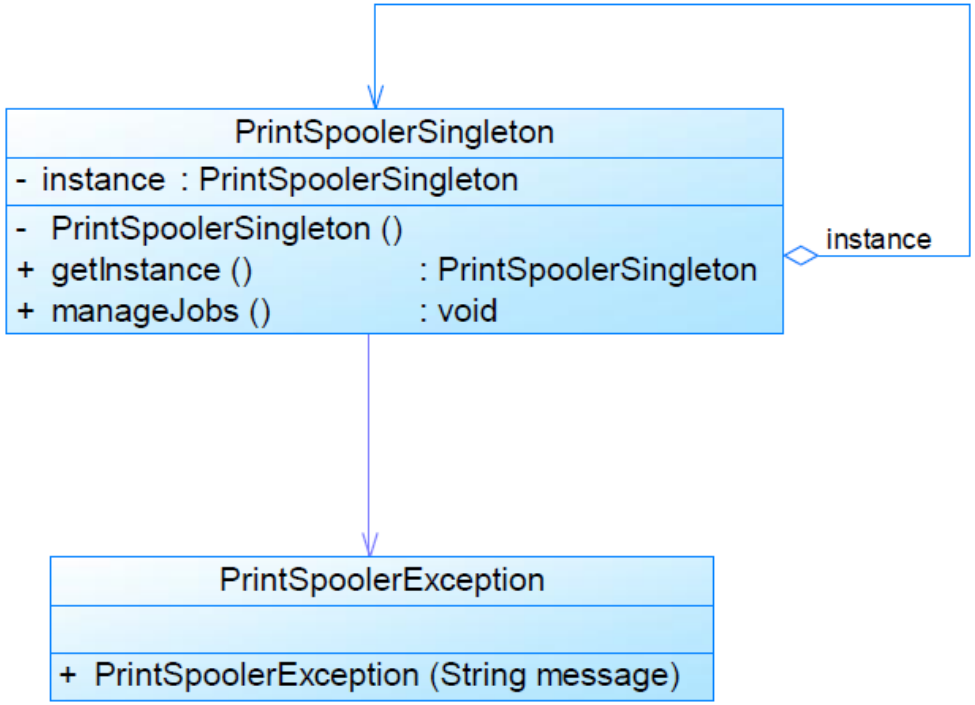
- 每一个类必须配备一个克隆方法。
- 配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类不是很难，但对于已有的类不一定很容易，特别当一个类引用不支持串行化的间接对象，或者引用含有循环结构的时候。

Prototype模式的适用性

- 原型模式应用于很多软件中，如果每次创建一个对象要花大量时间，原型模式是最好的解决方案。
- 很多软件提供的复制(Ctrl+C)和粘贴(Ctrl+V)操作就是原型模式的应用，复制得到的对象与原型对象是两个类型相同但内存地址不同的对象，通过原型模式可以大大提高对象的创建效率。
- 使用场景：
  - 资源优化场景
  - 性能和安全要求的场景
  - 一个对象多个修改者的场景

5）单例模式 Singleton

- 举例：在操作系统中，打印池 (PrintSpooler) 是一个用于管理打印任务的应用程序，通过打印池用户可以删除、中止或者改变打印任务的优先级，在一个系统中只允许运行一个打印池对象，如果重复创建打印池则抛出异常。

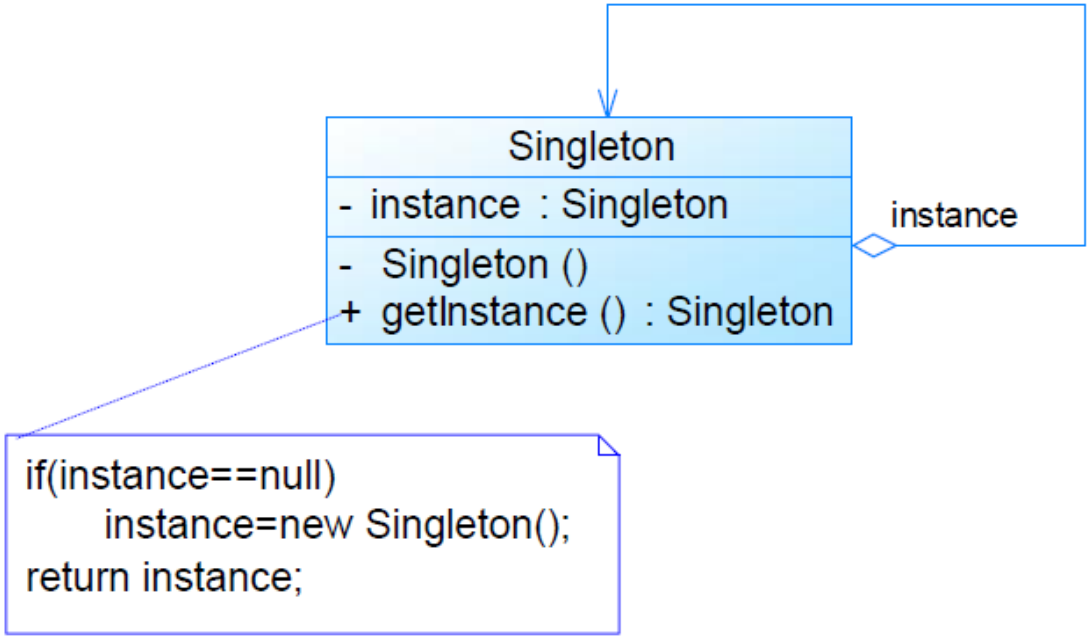


特殊的类

- 类和它的实例间一般是一对多的关系。对大多数的类而言，都可以创建多个实例。在需要这些实例时创建它们，在这些实例不再有用时删除它们。这些实例的来去伴随着内存的分配和归还。
- 但有一些类，应该只有一个实例。这个实例似乎应该在程序启动时被创建出来，且只有在程序结束时才被删除。
  - 如：应用程序的基础对象，通过这些对象可以得到系统中的许多其他对象；
  - 如：工厂对象，用于创建系统中其它对象
  - 如：管理器对象，负责管理某些其他对象并以合适的方式去控制它们。

- 如何保证一个类只有一个实例并且这个实例易于被访问呢?
  - 定义一个全局变量可以确保对象随时都可以被访问，但不能防止我们实例化多个对象。
  - 一个更好的解决办法是让类自身负责保存它的唯一实例。这个类可以保证没有其他实例被创建，并且它可以提供一个访问该实例的方法。这就是单例模式的模式动机。

Singleton的设计方案



单例模式的优点

- 提供了对唯一实例的受控访问。因为单例类封装了它的唯一实例，所以它可以严格控制客户怎样以及何时访问它，并为设计及开发团队提供了共享的概念。
- 由于在系统内存中只存在一个对象，因此可以节约系统资源，对于一些需要频繁创建和销毁的对象，单例模式无疑可以提高系统的性能。
- 允许可变数目的实例。我们可以基于单例模式进行扩展，使用与单例控制相似的方法来获得指定个数的对象实例。

单例模式的缺点

- 由于单例模式中没有抽象层，因此单例类的扩展有很大的困难。
- 单例类的职责过重，在一定程度上违背了“单一职责原则”。因为单例类既充当了工厂角色，提供了工厂方法，同时又充当了产品角色，包含一些业务方法，将产品的创建和产品的本身的功能融合到一起。
- 滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出；现在很多面向对象语言(如Java、C#)的运行环境都提供了自动垃圾回收的技术，因此，如果实例化的对象长时间不被利用，系统会认为它是垃圾，会自动销毁并回收资源，下次利用时又将重新实例化，这将导致对象状态的丢失。

适用场景

- 在以下情况下可以使用单例模式：
  - 系统只需要一个实例对象，如系统要求提供一个唯一的序列号生成器，或者需要考虑资源消耗太大而只允许创建一个对象。
  - 客户调用类的单个实例只允许使用一个公共访问点，除了该公共访问点，不能通过其他途径访问该实例。
  - 在一个系统中要求一个类只有一个实例时才应当使用单例模式。反过来，如果一个类可以有几个实例共存，就需要对单例模式进行改进，使之成为多例模式。
- 实例：
  - 垃圾回收站、数据库连接池、Windows中的文件管理器等。
  - 配置信息类、管理类、控制类、门面类、代理类等。
  - Java的Struts、Spring框架，.Net的Spring.Net框架，以及Php的Zend框架都大量使用了单例模式。

Summary：Creational Patterns

- Factory Method
  - 本质：用一个virtual method完成创建过程
- Abstract Factory
  - 一个product族的factory method构成了一个factory接口
- Builder
  - 通过一个构造算法和builder接口把构造过程与客户隔离开
- Prototype
  - 通过product原型来构造product，Clone + prototype manager
- Singleton
  - 单实例类型。由构造函数，一个静态变量，以及一个静态方法对实例化进行控制和限制

- Creational Patterns的目的就是封装对象创建的变化。
  - 例如FactoryMethod模式和AbstractFactory模式，建立了专门的抽象的工厂类，以此来封装未来对象的创建所引起的可能变化。而Builder模式则是对对象内部的创建进行封装，由于细节对抽象的可替换性，使得将来面对对象内部创建方式的变化，可以灵活地进行扩展或替换。

Structural Patterns

- 在面向对象软件系统中，每个类都承担了一定的职责，它们可以相互协作，实现一些复杂的功能。
  - 结构型模式关注的是如何将现有类或对象组织在一起形成更加强大的结构
  - 不同的结构型模式从不同的角度来组合类和对象

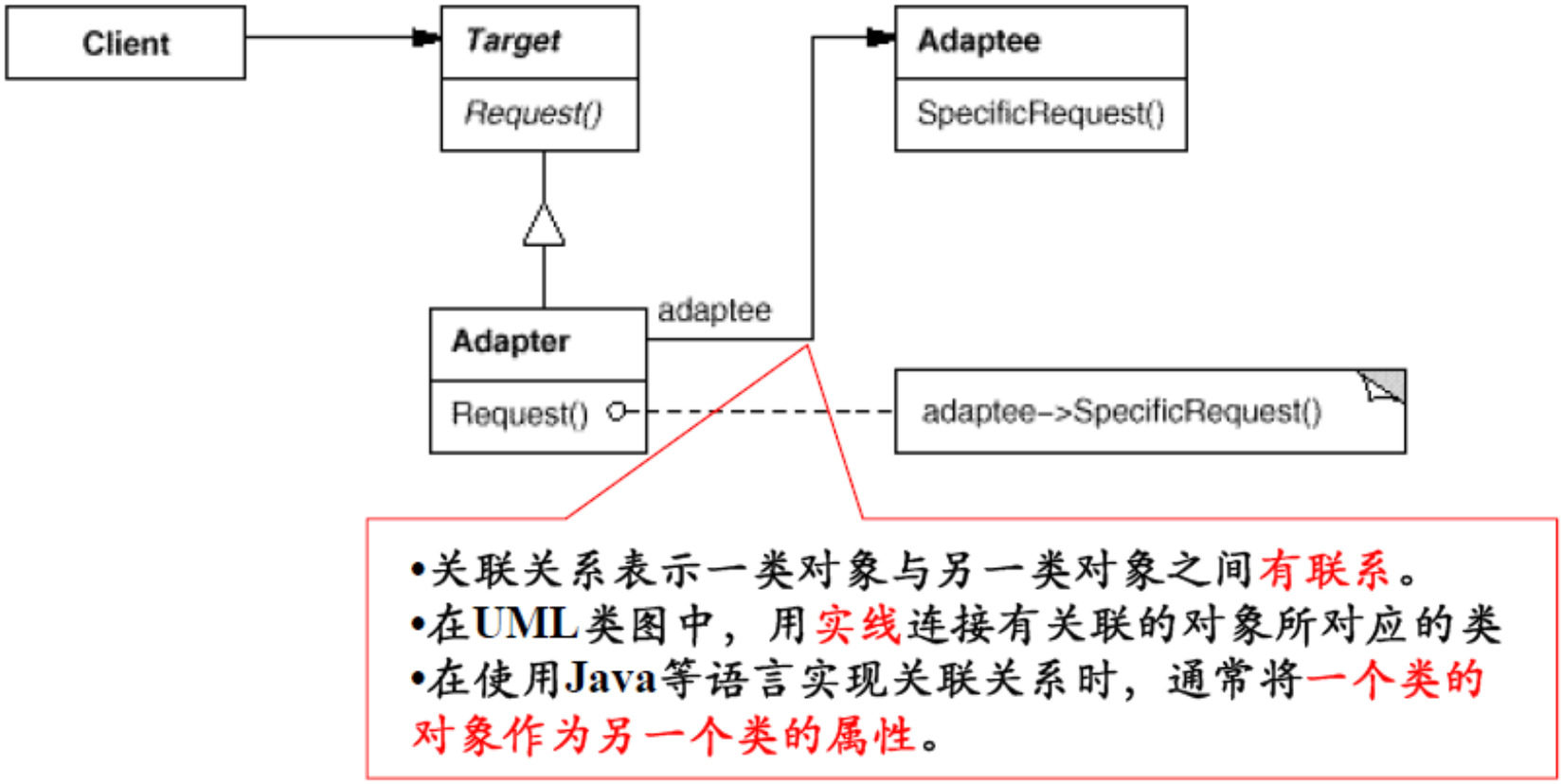
6）适配器模式Adapter

- 适配是在不改变原有实现的基础上，将原先不兼容的接口转换成兼容的接口。




- 在软件系统中，由于应用环境的变化，常常需要将“一些现存的对象”放在新的环境中应用，但是新环境要求的接口不满足这些现存对象的要求。
  - 如何应对这种“迁移的变化”？如何既能利用现有对象的良好实现，又能同时满足新的应用环境所要求的接口?

2.对象Adapter：Adapter与Adaptee是关联关系



类的适配器模式 *VS* 对象的适配器模式

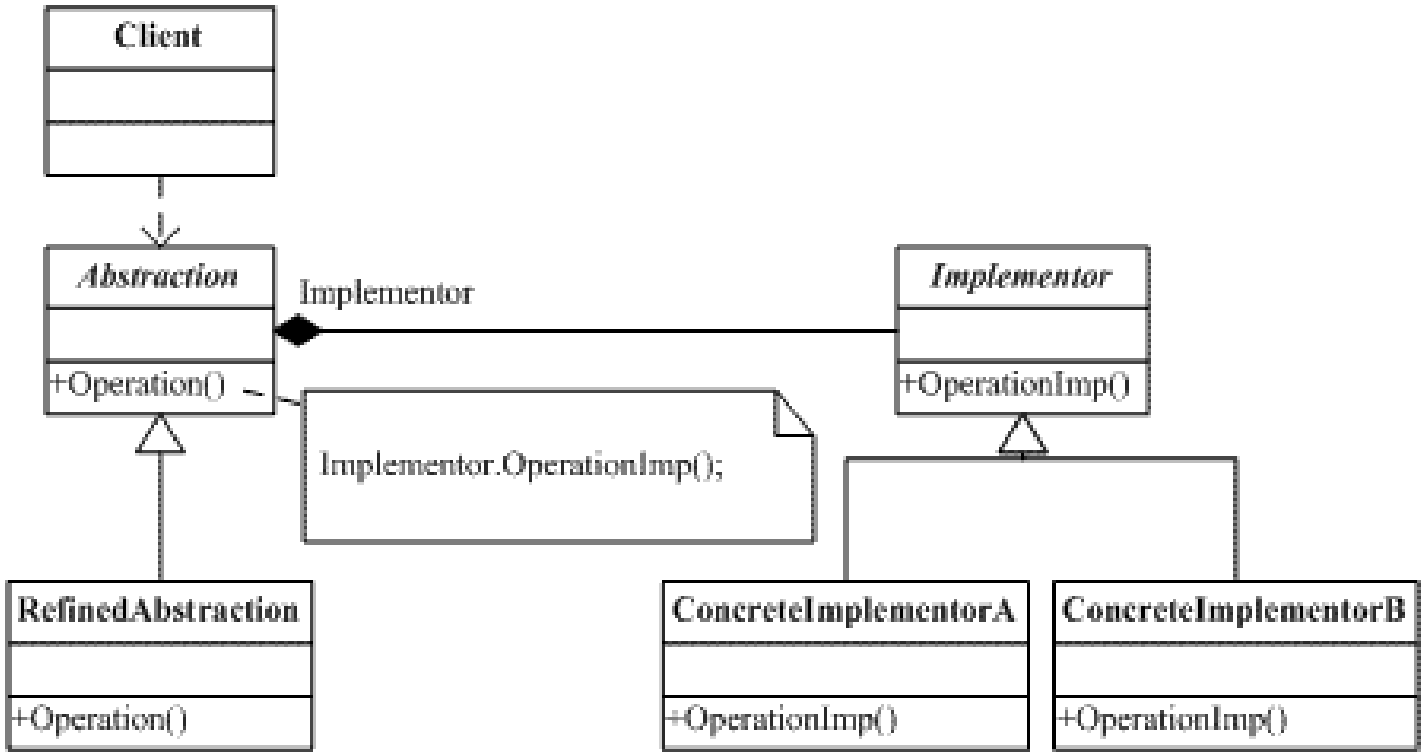
- 类的适配器模式
  - 引入一个适配器类，由于适配器类是源类的子类，因此可以在适配器类中置换掉源的一些方法。
- 对象的适配器模式
  - 一个适配器可以把多个不同的源适配到同一个目标。即：同一个适配器可以把源类和它的子类都适配到目标接口。
  - 与类的适配器模式相比，想要置换源类的方法不容易。如果一定要置换掉源类的一个或多个方法，就需要先做一个源类的子类，将源类的方法置换掉，然后再把源类的子类当作真正的源进行适配。
  - 虽然要想置换源类的方法不容易，但是要想增加一些新的方法很方便，而且新增的方法可同时适用于所有的源。

适用场景

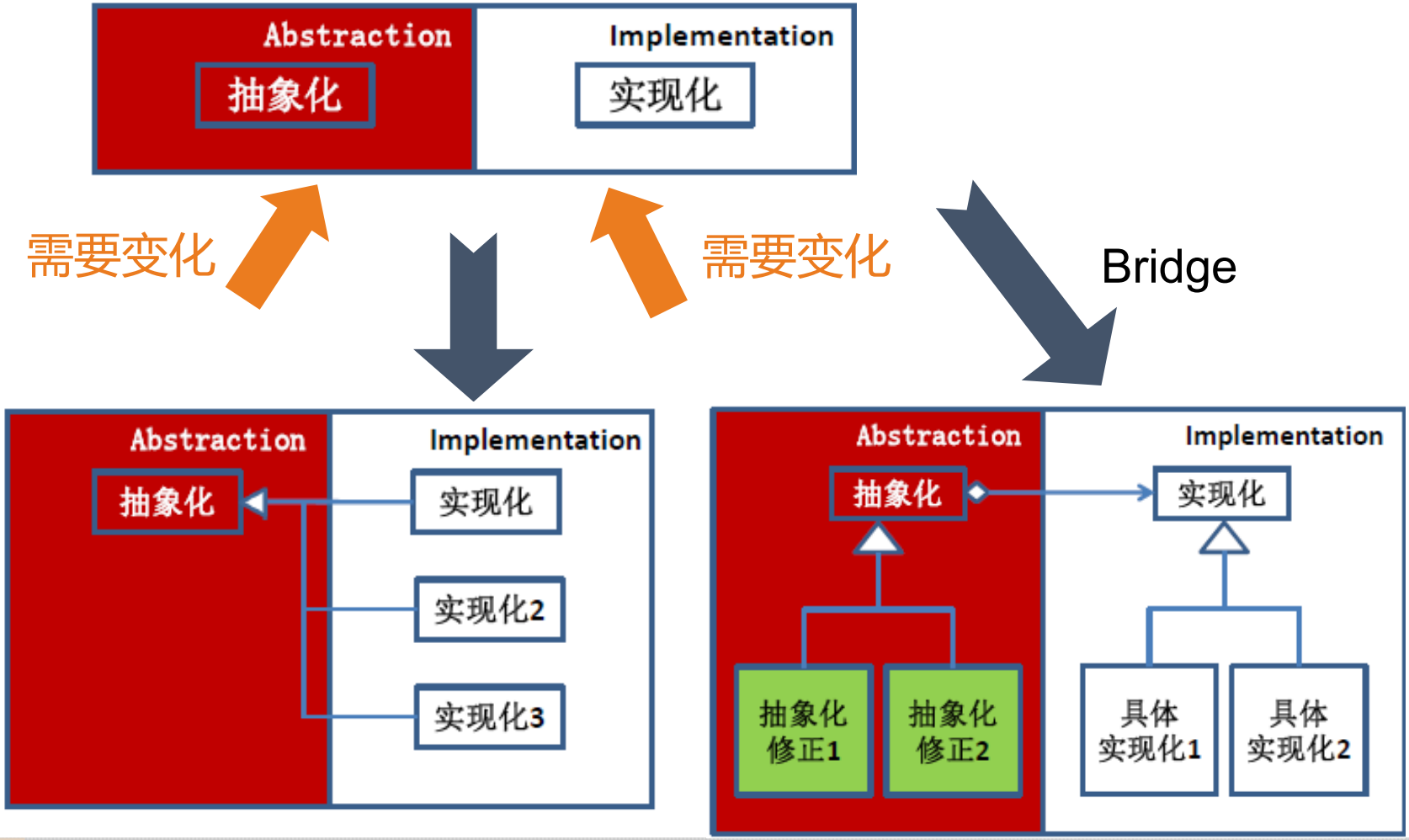
- 应用场景:
  - 扩展应用时------想要修改一个投产中的接口时
- Tips:
  1. 用于解决正在服役的项目问题，在详细设计阶段一般不予考虑。
  2. 补偿模式，用于解决接口不相容问题，通过把非本系统接口的对象包装成本系统可接受的对象，简化了系统大规模变更的风险。

Bridge模式

- Intent
  - 将抽象与其实现解耦，使它们可以独立变化
- Solution

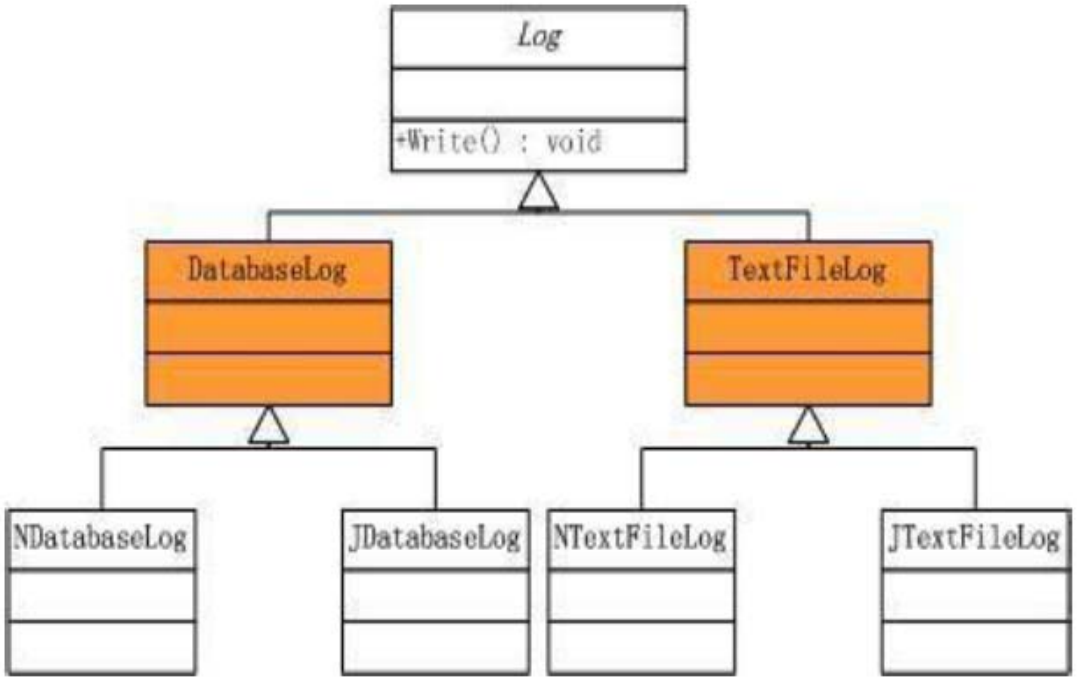


对“变化”的封装

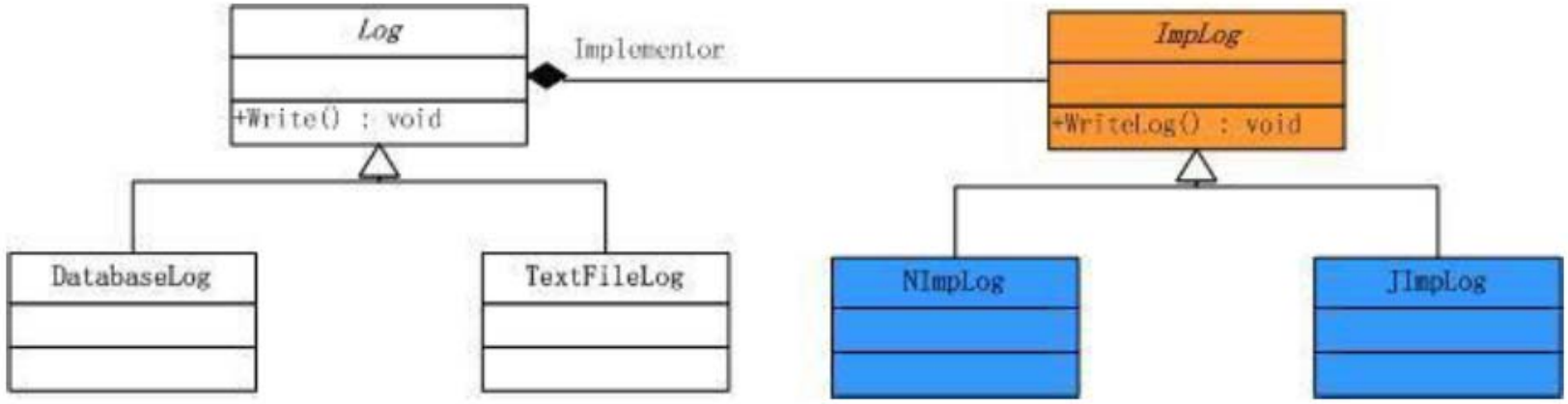

Abstraction
抽象化
Implementation
实现化
需要变化
需要变化
Bridge
Abstraction
抽象化
Implementation
实现化
实现化2
实现化3
Abstraction
抽象化
Implementation
实现化
抽象化
修正1
抽象化
修正2
具体
实现化1
具体
实现化2

举例：Log工具

- 现在我们要开发一个通用的日志记录工具
  - 支持数据库记录DatabaseLog和文本文件记录FileLog两种方式
  - 同时它既可以运行在.NET平台，也可以运行在Java平台上
- 如果不考虑Bridge模式，传统的OO设计将得到下面的类结构



应用Bridge模式的设计



适用场景

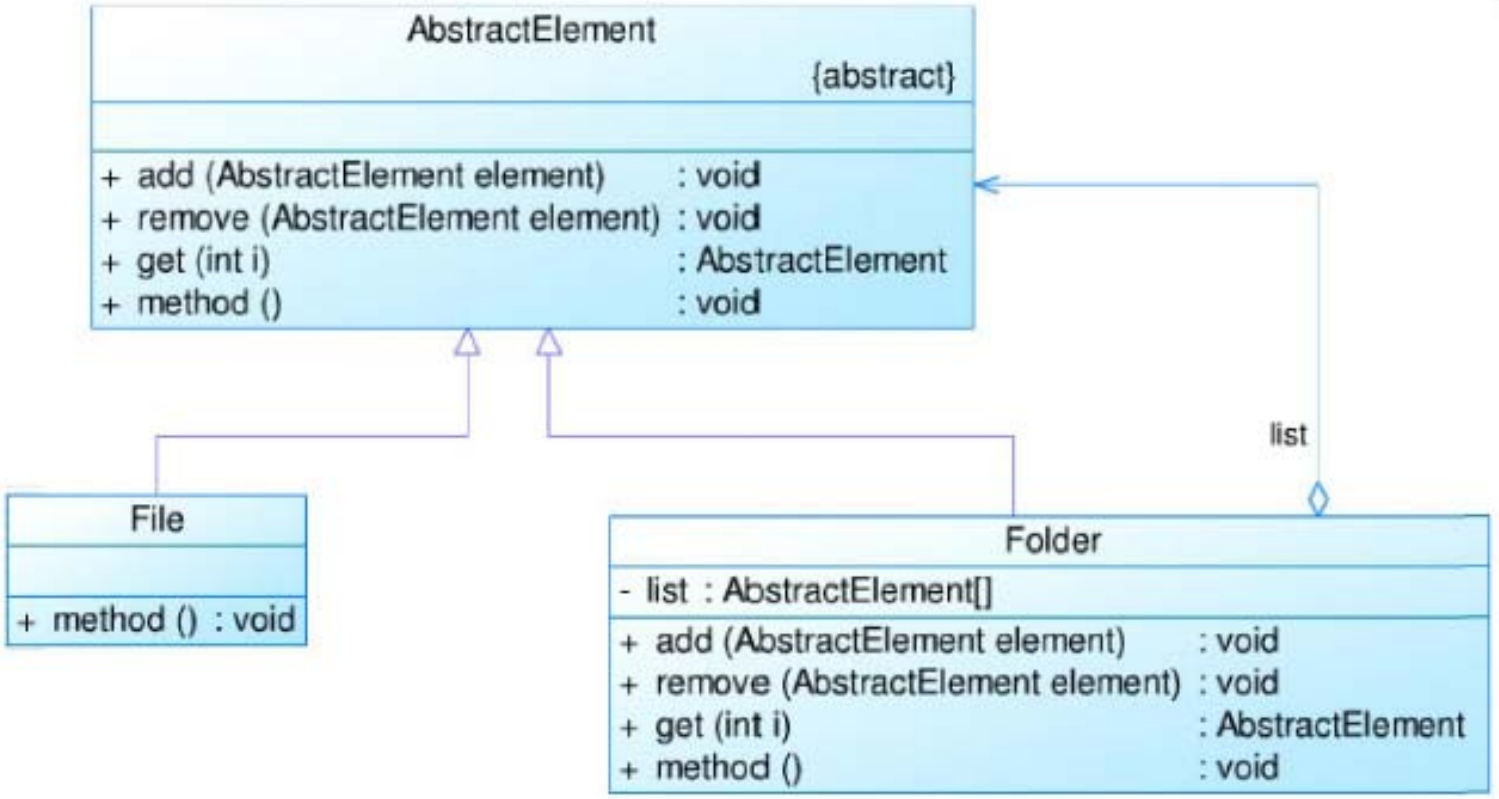
- 如果系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的联系；
- 设计要求实现化角色的任何改变不应当影响客户端，或者说实现化角色的改变对客户端是完全透明的；
- 一个构件有多于一个的抽象化角色和实现化角色，系统需要它们之间进行动态耦合；
- 虽然在系统中使用继承是没有问题的，但是由于抽象化角色和具体化角色需要独立变化，设计要求需要独立管理这两者。

8）组合模式Composite

- 在面向对象的系统中，我们常会遇到一类容器对象----它们在充当对象的同时，又是其他对象的容器。
- 客户代码过多地依赖于容器对象复杂的内部实现结构,容器对象内部实现结构(而非抽象接口)的变化将引起客户代码的频繁变化，带来了代码的维护性、扩展性差等问题。
- 如何将“客户代码与复杂的容器对象结构”解耦？让容器对象自己来实现自身的复杂结构，从而使得客户代码就像处理简单对象一样来处理复杂的容器对象？

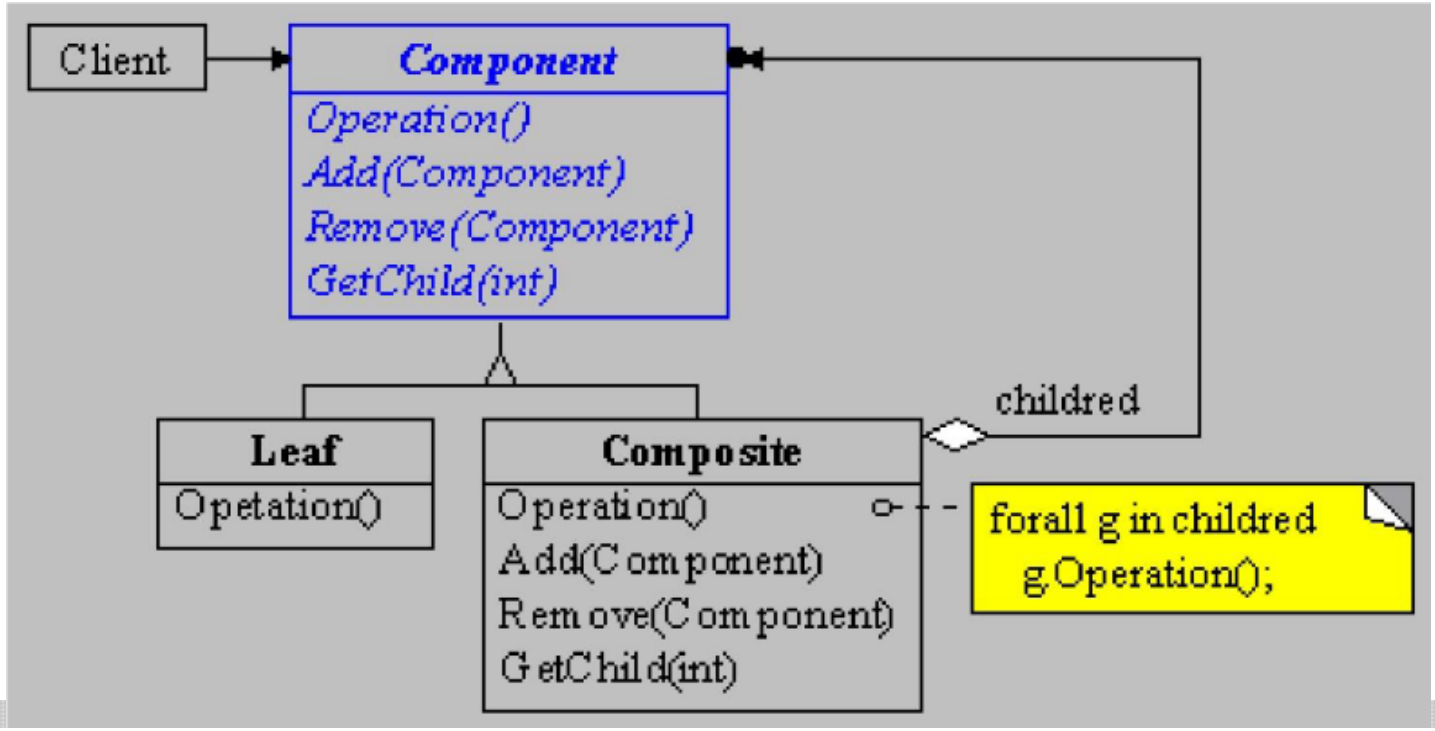
举例: 文件系统

- 文件系统组合模式结构图:

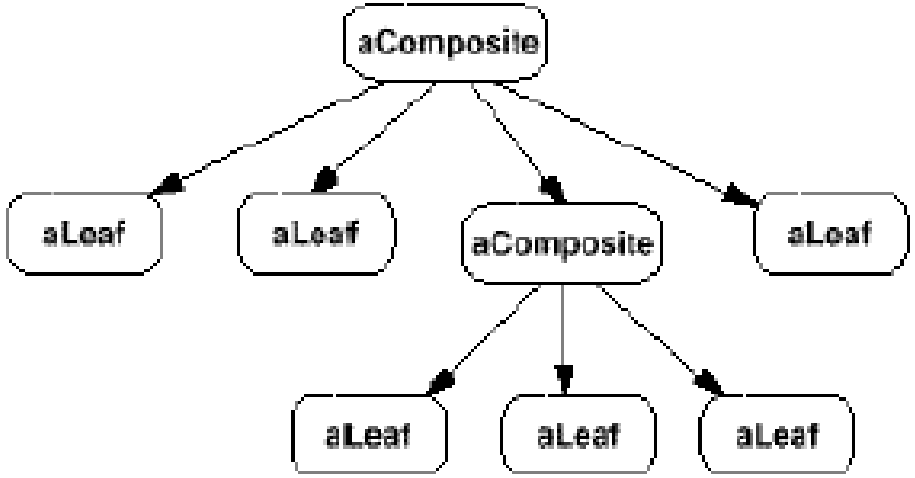


Composite模式的结构

- 组合模式定义了一个抽象构件类，它既可以代表叶子，又可以代表容器，而客户端针对该抽象构件类进行编程，无须知道它到底表示的是叶子还是容器，可以对其进行统一处理。



- 容器对象与抽象构件类之间还建立一个聚合关联关系，在容器对象中既可以包含叶子，也可以包含容器，以此实现递归组合，形成一个树形结构。



组合模式的实现方法

- 安全式组合模式实现方法
  - 只有容器构件角色才配备管理聚集的方法，而叶子构件没有这些方法。
- 透明式组合模式实现方法
  - 容器构件和叶子构件角色都配备管理聚集的方法

举例：一个绘图软件

- 一个绘图软件给出各种工具用来描绘由线、长方形和圆形等基本图形组成的图形。
  - 设计应当包括Line、Rectangle和Circle等对象，而且每一个对象都应当配备有一个draw()方法，在调用时会画出对象所代表的图形。
  - 由于一个复杂的图形是由基本图形组合而成的，因此，一个组合的图形应当有一个列表，存储对所有的基本图形对象的引用。复合图形的draw()方法在调用时，应当逐一调用所有列表上的基本图形对象的draw()方法。

Composite模式的优缺点

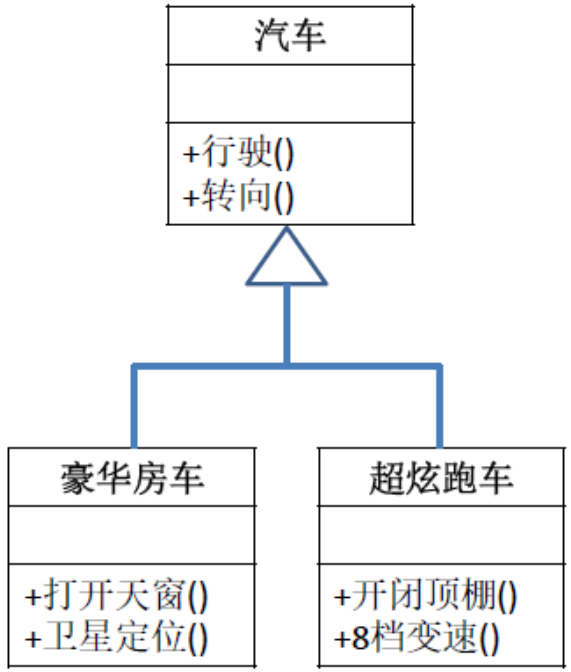
- 优点
  - 组合模式可以很容易地增加新种类的构件；
  - 使用组合模式可以使客户端变得很容易设计，因为客户端不需要知道构件是叶子还是容器对象。
- 缺点
  - 使用组合模式后，控制容器对象的类型就不太容易；
  - 用继承的方法增加新的行为很困难。

适用场景

- 想表示对象的“部分-整体”层次结构
- 希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象

9）装饰器模式Decorator

- 我们已经有了一些基本功能，可以通过继承在此基础上对已有功能进行扩展
  - 例如：
    - 已经有了汽车——具有：行驶、转向、停止等基本功能
    - 扩展豪华房车——除了基本功能外，增加：开闭天窗、卫星定位、车窗防弹、车载冰箱等功能
    - 扩展超炫跑车——除了基本功能外，增加：开闭顶棚、8档变速、涡轮加速、自动驾驶等功能
  - 继承可以方便地实现上述扩展



- 但是，对于超豪华、定制的名车怎么办？
  - 有钱的到一定程度的人连汽车也要量身定做，上面的功能他们希望能够自由选择：
    - 开闭天窗、卫星定位、车窗防弹、车载冰箱
    - 开闭顶棚、8档变速、涡轮加速、自动驾驶
  - 比如一辆既有豪华房车般奢华，同时又能如超炫跑车般享受驾驶乐趣的车：
    - 卫星定位、车载冰箱、8档变速、自动驾驶
  - 或者：
    - 开闭天窗、卫星定位、车窗防弹、8档变速、涡轮加速、自动驾驶
  - 或者:
    - 开闭天窗、车载冰箱、涡轮加速、自动驾驶
- 用继承的方式无法应对上面的变化性，因为订做的情况下，可能的组合有太多种了

- 对上述问题的总结：

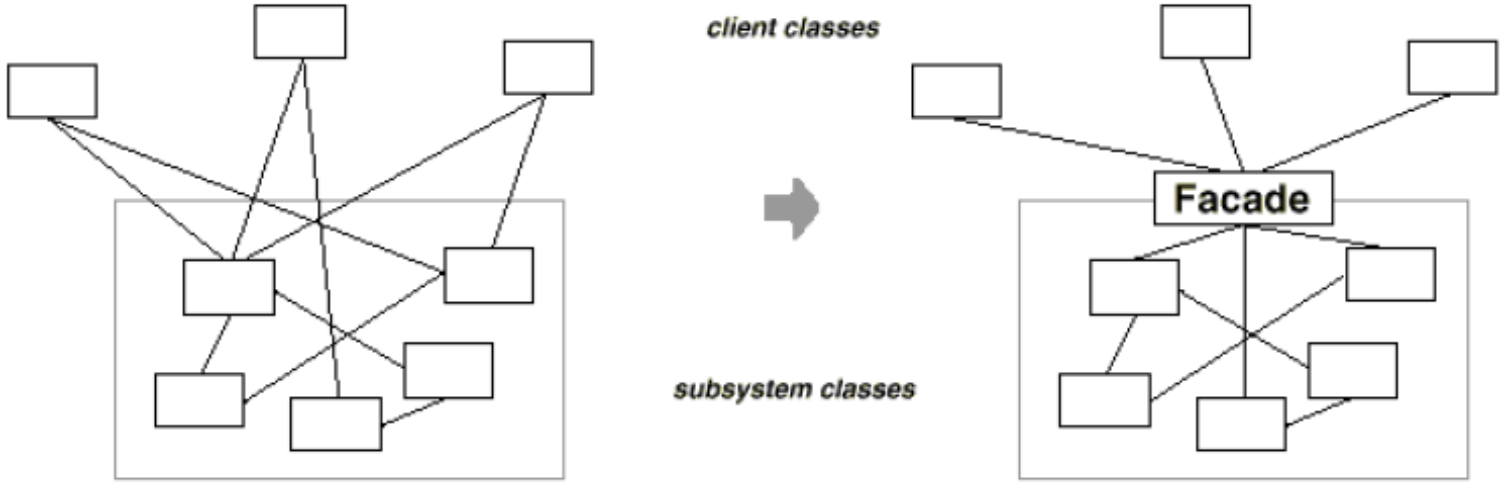
  我们如何应对给一个对象，而不是整个类添加功能

——毕竟，定制的轿车不需要量产，也就是说，我们不需要为每一款定制的靓车编写一个类，并期望用这个类派生多个实例。

如何使“对象功能的扩展”能够根据需要来动态地实现？同时避免“扩展功能的增多”带来的子类膨胀问题？从而使得任何“功能扩展变化”所导致的影响降为最低？

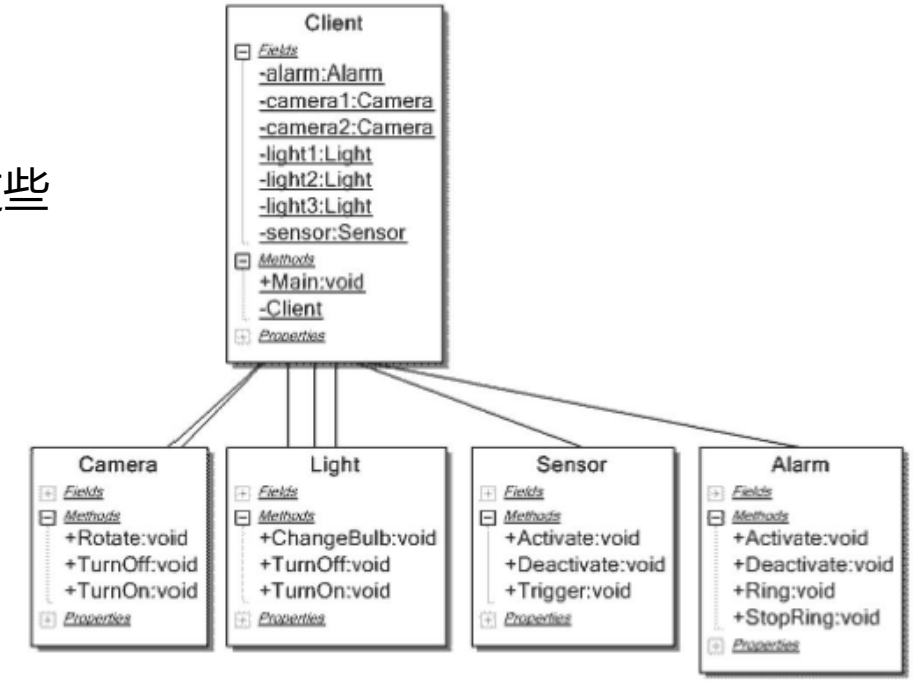
Façade模式

- Intent
  - 为子系统中的一组接口提供一个一致的界面，Façade模式定义了一个高层接口，这个接口使这一子系统更加容易使用
- Structure

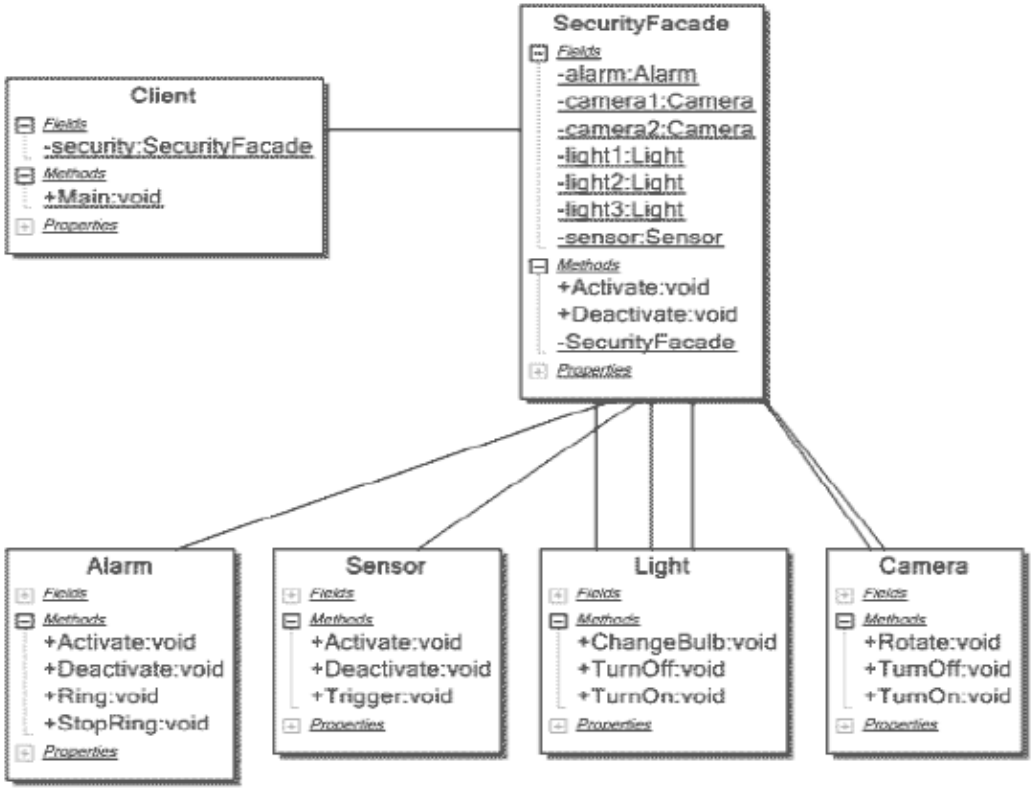


举例：保安系统

- 问题：一个保安系统由两个录像机、三个电灯、一个遥感器和一个警报器组成。保安系统的操作人员需要经常将这些仪器启动和关闭。
- 设计方案1：不使用门面模式的设计
  - 操作这个保安系统的操作员必须直接操作所有的这些部件



- 设计方案2：使用门面模式的设计
  - 一个合情合理的改进方法就是准备一个系统的控制台，作为保安系统的用户界面。



适用场景

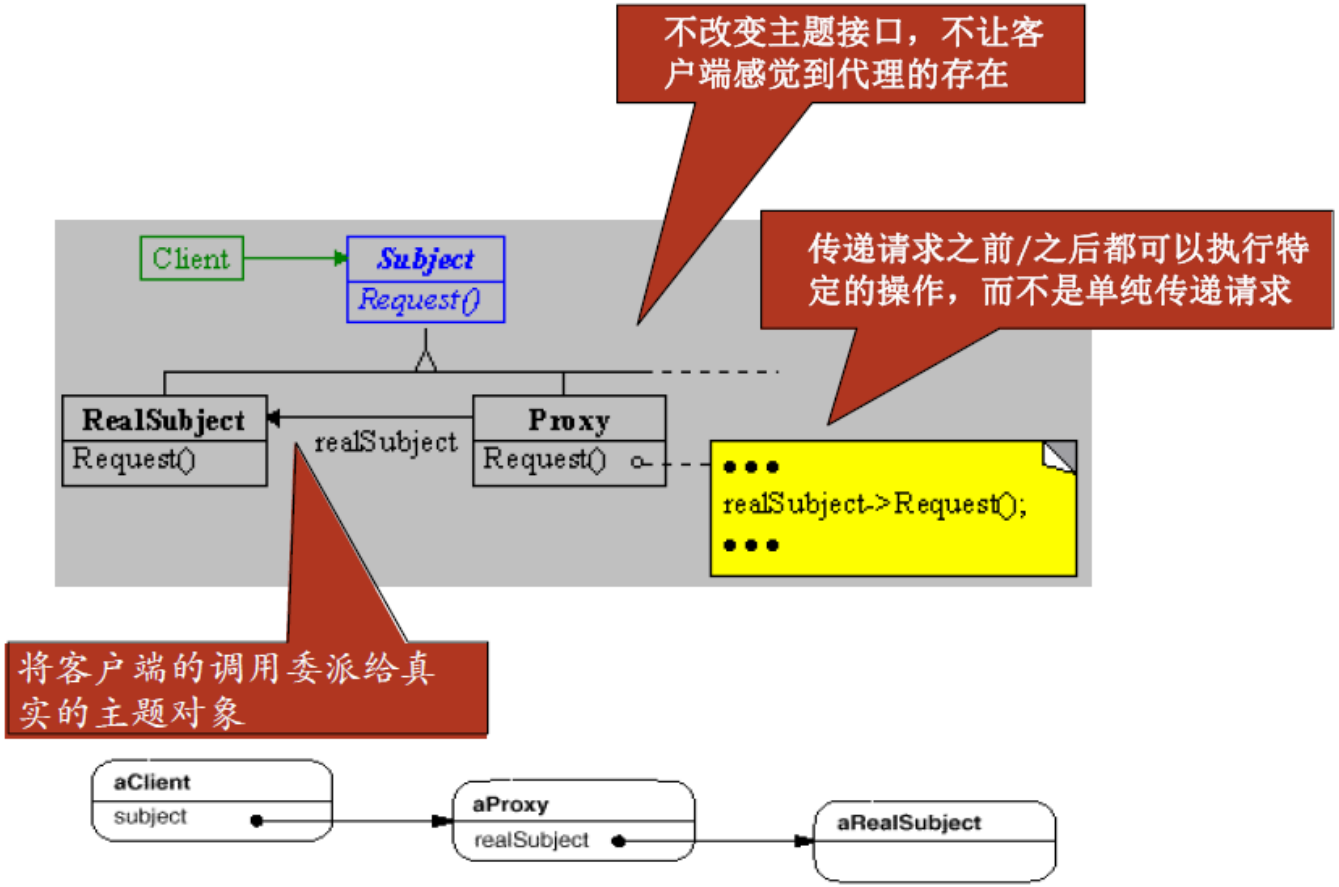
- 为一个复杂子系统提供一个简单接口
- 提高子系统的独立性
- 在层次化结构中，可以使用Façade模式定义系统中每一层的入口

11）代理模式Proxy

- 在软件系统中，我们无时不在跨越障碍，当我们访问网络上一台计算机的资源时，我们正在跨越网络障碍，当我们去访问服务器上数据库时，我们又在跨越数据库访问障碍，同时还有网络障碍。跨越这些障碍有时非常复杂，如果我们更多的去关注处理这些障碍问题，可能就会忽视了本来应该关注的业务逻辑问题，Proxy模式有助于我们去解决这些问题。
- 举例：我们以一个简单的数学计算程序为例，这个程序(Math)只负责进行简单的加减乘除运算。

Proxy模式

- 为其他对象提供一种代理以控制对这个对象的访问。



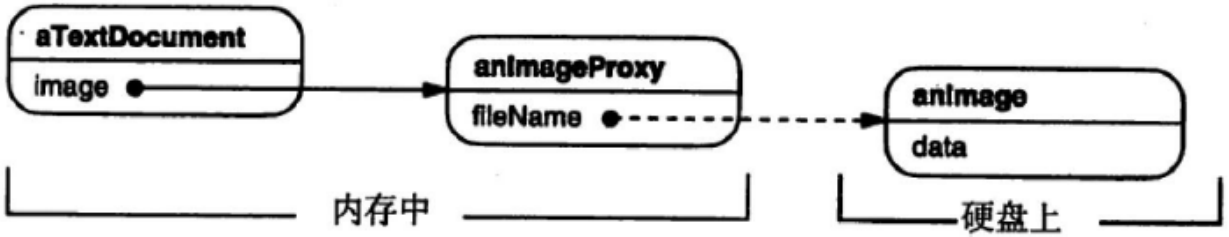
常见的代理类型

- **远程(Remote)代理**：为一个位于不同的地址空间的对象提供一个本地的代理对象，这个不同的地址空间可以是在同一台主机中，也可是在另一台主机中，远程代理又叫做大使(Ambassador)。
- **虚拟(Virtual)代理**：如果需要创建一个资源消耗较大的对象，先创建一个消耗相对较小的对象来表示，真实对象只在需要时才会被真正创建。
- **Copy-on-Write代理**：它是虚拟代理的一种，把复制（克隆）操作延迟到只有在客户端真正需要时才执行。一般来说，对象的深克隆是一个开销较大的操作，Copy-on-Write代理可以让这个操作延迟，只有对象被用到的时候才被克隆。

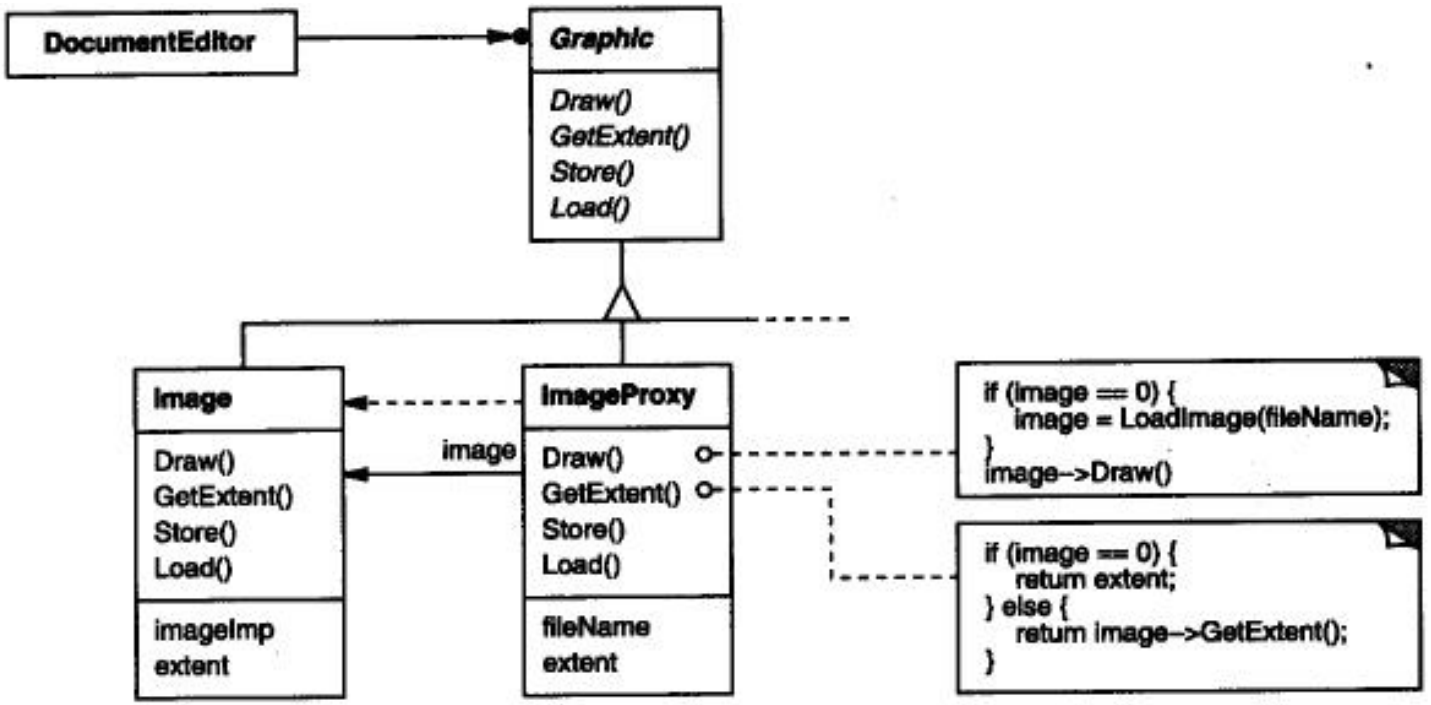
- **保护(Protector Access)代理**：控制对一个对象的访问，可以给不同的用户提供不同级别的使用权限。
- **缓冲(Cache)代理**：为某一个目标操作的结果提供临时的存储空间，以便多个客户端可以共享这些结果。
- **防火墙(Firewall)代理**：保护目标不让恶意用户接近。
- **同步化(Synchronization)代理**：使几个用户能够同时使用一个对象而没有冲突。
- **智能引用(SmartReference)代理**：当一个对象被引用时，提供一些额外的操作，如将此对象被调用的次数记录下来等。

举例：文档编辑器

- 我们开发一个可以在文档中嵌入图形对象的文档编辑器。有些图形对象(如大型光栅图像)的创建开销很大。但是打开文档必须很迅速，因此我们在打开文档时应避免一次性创建所有开销很大的对象。因为并非所有这些对象在文档中都同时可见，所以也没有必要同时创建这些对象。
- 这一限制条件意味着，对于每一个开销很大的对象，应该根据需要进行创建，当一个图像变为可见时会产生这样的需要。但是在文档中我们用什么来代替这个图像呢?我们又如何才能隐藏根据需要创建图像这一事实，从而不会使得编辑器的实现复杂化呢?例如，这种优化不应影响绘制和格式化的代码。
- 问题的解决方案是使用另一个对象，即图像Proxy，替代那个真正的图像。Proxy可以代替一个图像对象并且在需要时负责实例化这个图像对象。



- 只有当文档编辑器激活图像代理的Draw操作以显示这个图像的时候图像Proxy才创建真正的图像。Proxy直接将随后的请求转发给这个图像对象。因此在创建这个图像以后，它必须有一个指向这个图像的引用。
- 我们假设图像存储在一个独立的文件中。这样我们可以把文件名作为对实际对象的引用。Proxy还存储了图像的尺寸(extent)，即它的长和宽。有了图像尺寸，Proxy元须真正实例化这个图像就可以响应格式化程序对图像尺寸的请求。



文档编辑器通过抽象的Graphic类定义的接口访问嵌入的图像。ImageProxy是那些根据需要创建的图像的类，ImageProxy保存了文件名作为指向磁盘上的图像文件的指针。该文件名被作为一个参数传递给ImageProxy的构造器。

ImageProxy还存储了这个图像的边框以及对真正的Image实例的指引，直到代理实例化真正的图像时，这个指引才有效。Draw操作必须保证在向这个图像转发请求之前，它已经被实例化了。GetExtent操作只有在图像被实例化后才向它传递请求，否则，ImageProxy返回它存储的图像尺寸。

Proxy模式的优点

- 代理模式能够协调调用者和被调用者，在一定程度上降低了系统的耦合度。
- 远程代理使得客户端可以访问在远程机器上的对象，远程机器可能具有更好的计算性能与处理速度，可以快速响应并处理客户端请求。
- 虚拟代理通过使用一个小对象来代表一个大对象，可以减少系统资源的消耗，对系统进行优化并提高运行速度。
- 保护代理可以控制对真实对象的使用权限。

Proxy模式的缺点

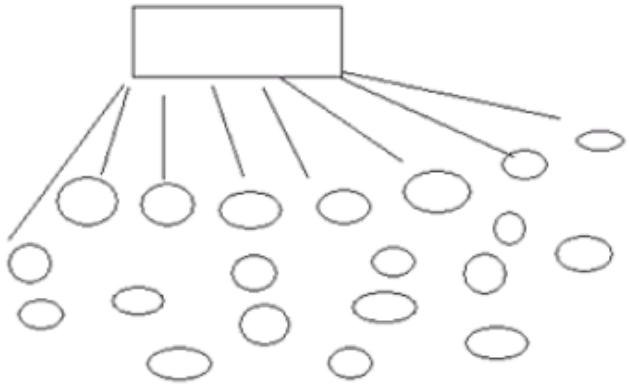
- 由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢。
- 实现代理模式需要额外的工作，有些代理模式的实现非常复杂。

适用场景

- **Remote Proxy**：为一个对象在不同的地址空间提供局部代表
- **Virtual Proxy**：根据需要创建开销很大的对象
- **Protection Proxy**：控制对原始对象的访问，用于对象应该有不同访问权限的时候
- **Smart Reference**：取代了简单的指针，在访问对象时执行一些附加操作

12）享元模式Flyweight

- 面向对象很好地解决了系统抽象性的问题，同时在大多数情况下，也不会损及系统的性能。但在某些特殊的应用中，由于对象的数量太大，采用面向对象会给系统带来难以承受的内存开销。比如图形应用中的图元等对象、字处理应用中的字符对象等。

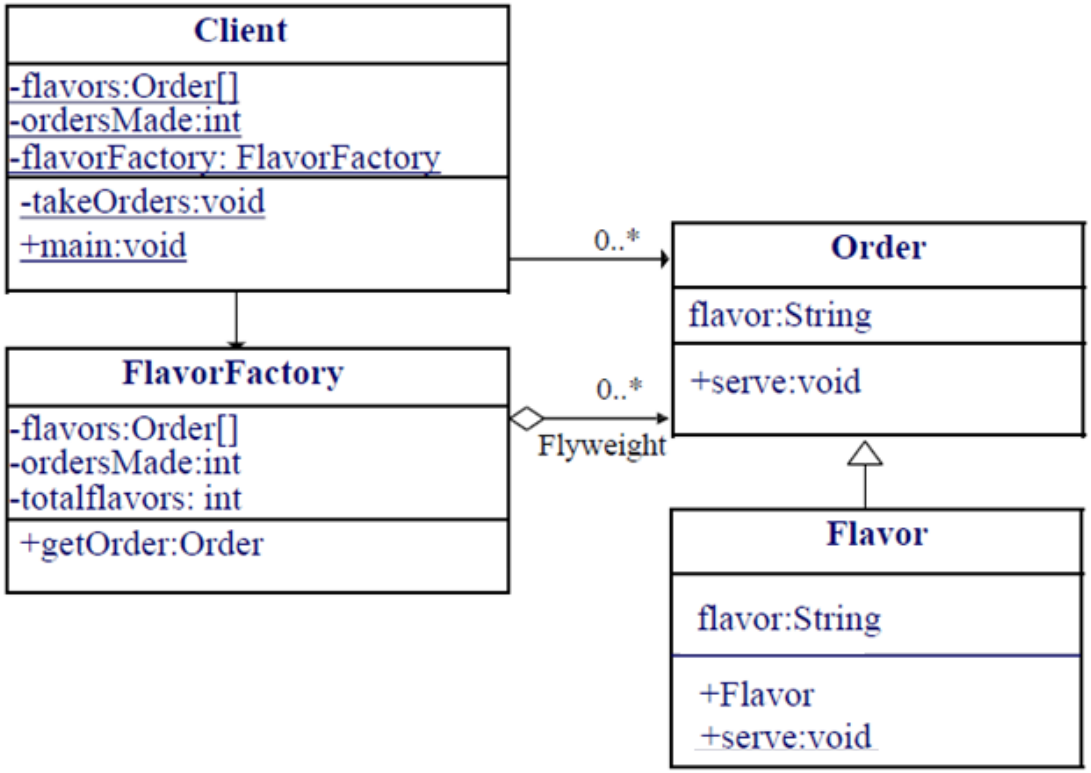


- 采用纯粹对象方案的问题在于大量细粒度的对象会很快充斥在系统中，从而带来很高的运行时代价------主要指内存需求方面的代价。
- 如何在避免大量细粒度对象问题的同时，让外部客户程序仍然能够透明地使用面向对象的方式来进行操作?

举例1：咖啡摊(Coffee Stall)项目

- 在这个咖啡摊系统里，有一系列的咖啡“风味(Flavor)”。 客人到摊位上购买咖啡，所有的咖啡均放在台子上，客人自己拿到咖啡后就离开摊位。咖啡有内蕴状态，也就是咖啡的风味；咖啡没有环境因素，也就是说没有外蕴状态。如果系统为每一杯咖啡都创建一个独立的对象的话，那么就需要创建出很多的细小对象来。这样就不如把咖啡按照种类(即“风味”)划分，每一种风味的咖啡只创建一个对象，并实行共享。
- 所有的咖啡都可按"风味"划分成如Capucino、Espresso等， 每一种风味的咖啡不论卖出多少杯，都是全同、不可分辨的。所谓共享，就是咖啡风味的共享，制造方法的共享等。 因此，享元模式对咖啡摊来说，就意味着不需要为每一份单独调制。摊主可以在需要时，一次性地调制出足够一天出售的某一种风味的咖啡。

采用单纯享元模式



- 抽象享元角色Order类，是所有具体享元类的超类，为这些类规定出需要实现的公共接口。“风味”角色Flavor类实现了Order角色所声明的接口，就是享元对象，flavor 属性是Flavor对象的内蕴状态。一个享元对象的内蕴状态在对象被创建出来以后就不再改变。

- 所有“风味”对象都应由“风味”工厂提供，而不应由客户直接创建。

```
public class FlavorFactory
{
  private Order[] flavors = new Flavor[10];
  private int ordersMade = 0;
  private int totalFlavors = 0 ;
  public Order getOrder(String flavorToGet)
  {
     if (ordersMade > 0) {
       for (int i =0; i < ordersMade;i++) {
         if (flavorToGet.equals((flavors[i]).getFlavor()))
         {
           return flavors[i];
         }
       }
     }
     flavors[ordersMade]= new Flavor(flavorToGet);
     totalFlavors++;
     return flavors[ordersMade++];
     }
   public  int getTotalFlavorsMade() {
        return totalFlavors ;
     }
}
```

- 系统的客户端角色，该角色实际上代表咖啡摊的工作人员。

```
public class Client
{
  private static Order[] flavors= new Flavor[100 ];
  private static int ordersMade =0;
  private static FlavorFactory flavorFactory;
  private static void takeOrders(String aFlavor)
  {
    flavors[ordersMade++]=
          flavorFactory.getOrder(aFlavor);
  }
```

```
public static void main(String[] args)
{
  flavorFactory =new FlavorFactory( );
  takeOrders("Black Coffee");
  takeOrders("Capucino" );
  takeOrders("Espresso" );
  takeOrders("Espresso" );
  takeOrders("Capucino" );
  takeOrders("Capucino" );
  takeOrders("Black Coffee");
  takeOrders("Espresso");
  takeOrders("Capucino" );
  takeOrders("Black Coffee");
  takeOrders("Espresso");
  for(int i = 0; i < ordersMade; i++)
  {
     flavors[i].serve ();
  }
  System.out.println( "\n Total teaFlavor objects
made:" +  flavorFactory.getTotalFlavorsMade()) ;
}
```

Tips:

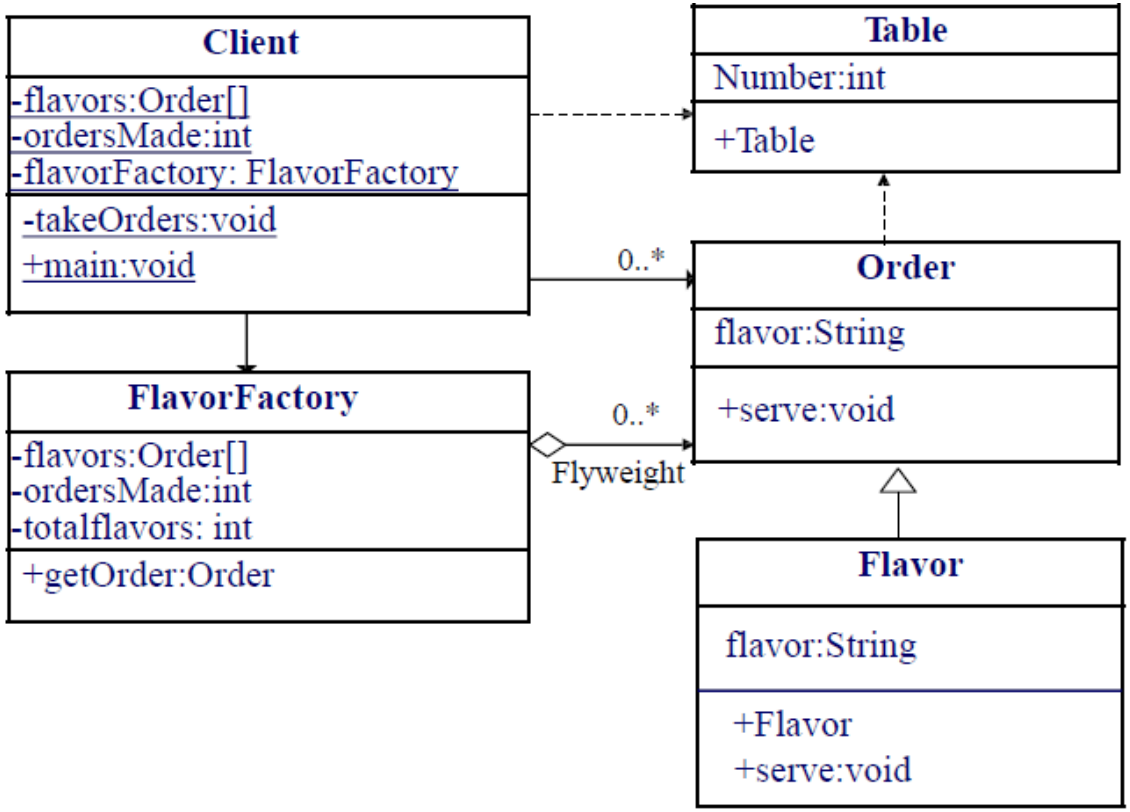
- 该咖啡摊为客人准备最多10种不同风味的咖啡。
- 虽然上述Client对象叫了11杯咖啡，但所有咖啡的风味只有三种，即BlackCoffee、Capucino和Espresso.

举例2：咖啡屋(Coffee Shop)项目

- 在前面的咖啡摊项目里，由于没有供客人坐的桌子，所有的咖啡均没有环境的影响。换言之，咖啡仅有内蕴状态，也就是咖啡的种类，而没有外蕴状态。
- 下面考虑一个规模稍稍大一点的咖啡屋(Coffee Shop) 项目。屋子里有很多的桌子供客人坐，系统除了需要提供咖啡的"风味"之外，还需要跟踪咖啡被送到哪一个桌位上，因此，咖啡就有了桌子作为外蕴状态。

采用复合享元模式

- 由于外蕴状态的存在，没有外蕴状态的单纯享元模式不再符合要求。系统的设计可以利用有外蕴状态的复合享元模式。

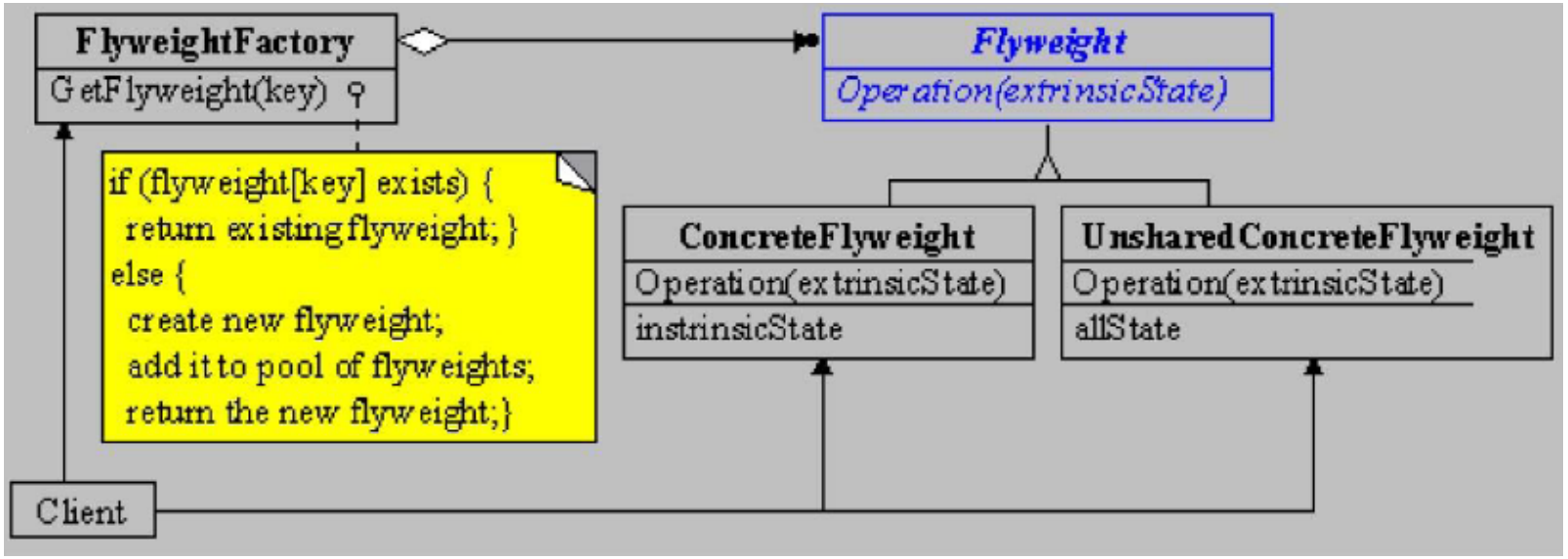


- Table类扮演环境角色，是所有享元角色所涉及的外蕴状态，在本项目中，代表咖啡屋里给客人座的桌子。
  - 每一杯咖啡都以“风味”作为内蕴状态，这个内蕴状态在享元对象被创建之后就不会改变；
  - 桌子作为享元对象的外蕴状态，由客户端保存，并在享元对象被创建出来后赋值给它。

Flyweight模式

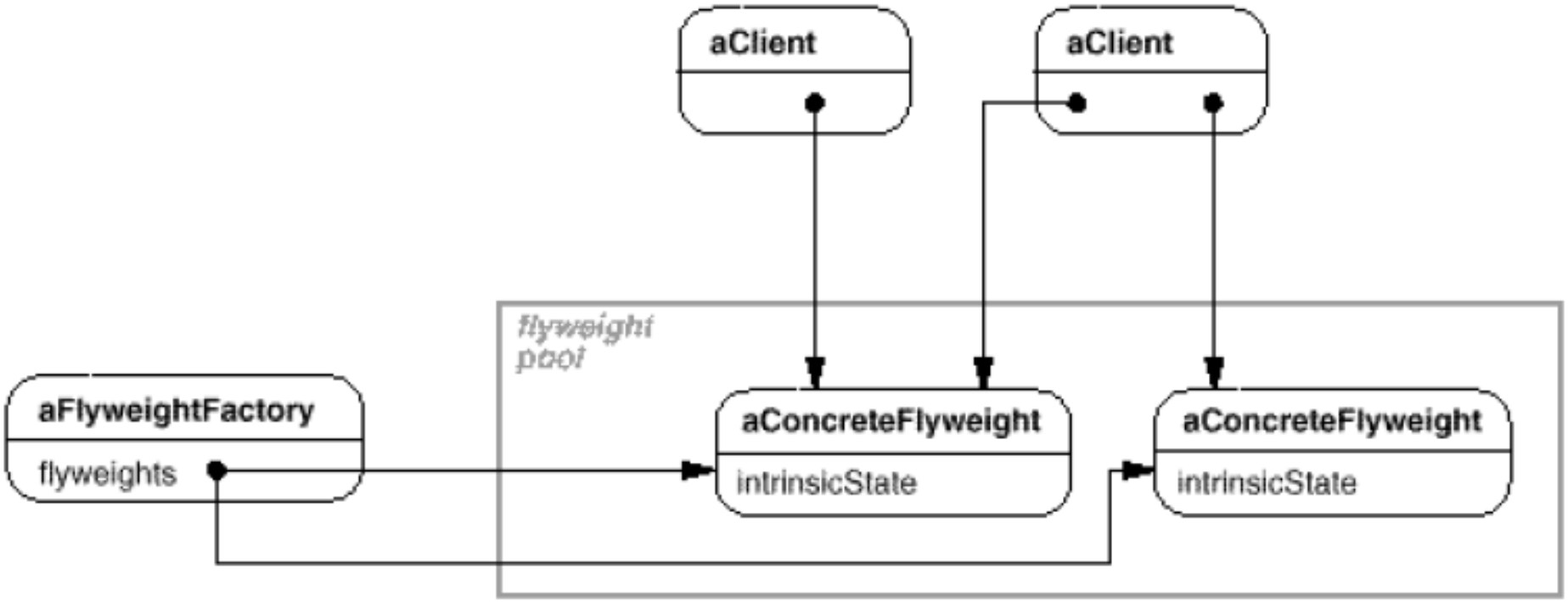
- **Intent**
  - 运用共享技术有效支持大量细粒度的对象。
- **Motivation**
  - 当对象的粒度太小的时候，大量对象将会产生巨大的资源消耗，因此考虑用共享对象(flyweight)来实现逻辑上的大量对象。Flyweight对象可用于不同的context中，本身固有的状态(内蕴状态)不随context发生变化，而其他的状态(外蕴状态)随context而变。
  - Flyweight在拳击比赛中指最轻量级，即"蝇量级"，有些翻译为"羽量级"。这里使用"享元模式"更能反映模式的用意。

Flyweight模式的structure



- Flyweight 描述一个接口，通过这个接口flyweight可以接受并作用于外部状态
- ConcreteFlyweight实现Flyweight接口，并为内部状态增加存储空间。必须是可以共享的，存储的状态必须是内部的
- UnsharedConcreteFlyweight不强制共享
- FlyweightFactory创建并管理flyweight对象, 确保合理地共享flyweight，提供已创建的flyweight实例或者创建一个 (如果不存在的话)
- Client维持一个对flyweight的引用, 计算或存储一个(多个)flyweight的外部状态

Object graph

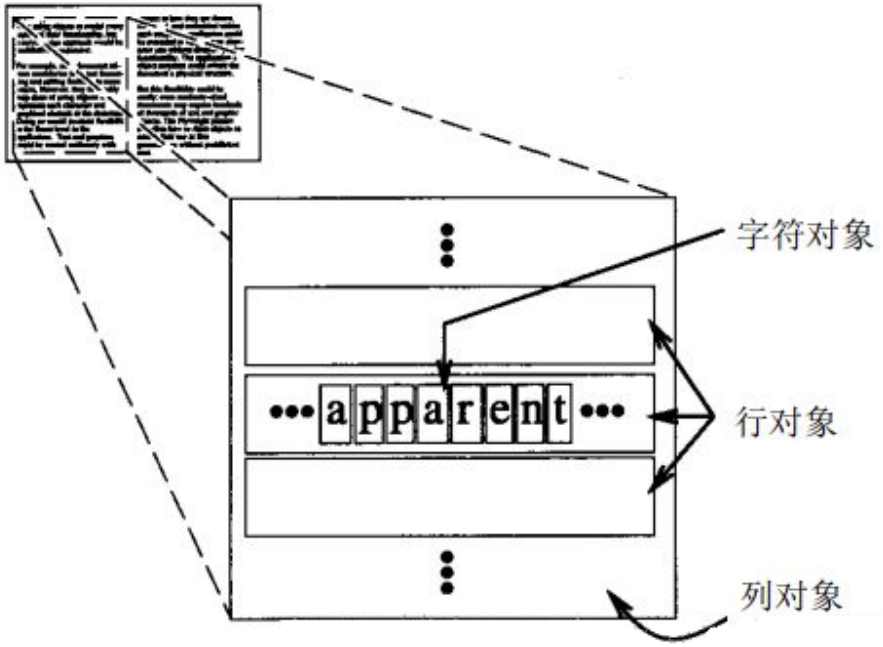


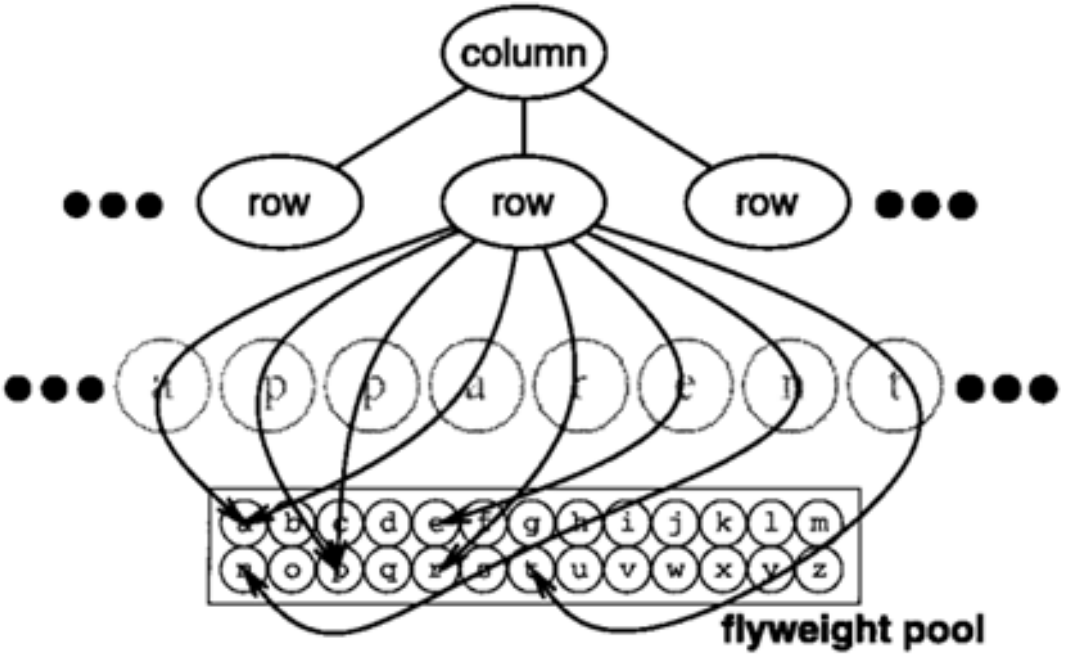
适用场景

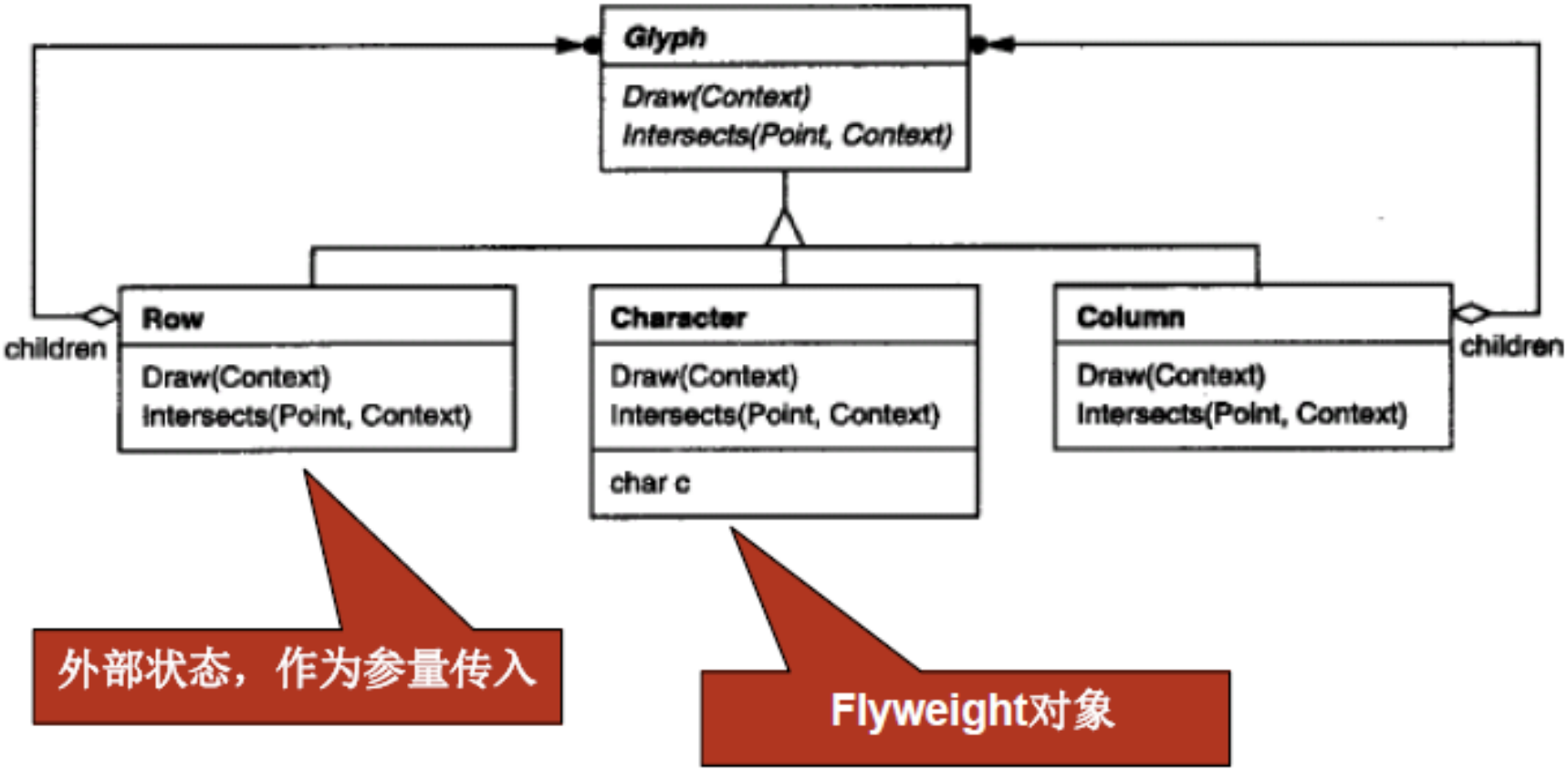
- 当以下所有的条件都满足时，可以考虑使用享元模式：
  - 一个系统有大量的对象。
  - 这些对象耗费大量的内存。
  - 这些对象的状态中的大部分都可以外部化。
  - 这些对象可以按照内蕴状态分成很多的组，当把外蕴对象从对象中剔除时，每一个组都可以仅用一个对象代替。
  - 软件系统不依赖于这些对象的身份，换言之，这些对象可以是不可分辨的。

举例：编辑器系统

- 享元模式在编辑器系统中大量使用
  - 一个文本编辑器往往会提供很多种字体，而通常的做法就是将每一个字母做成一个享元对象。享元对象的内蕴状态就是这个字母，而字母在文本中的位置和字模风格等其他信息则是外蕴状态。
  - 比如，字母a可能出现在文本的很多地方，虽然这些字母a的位置和字模风格不同，但是所有这些地方使用的都是同一个字母对象。这样一来，字母对象就可以在整个系统中共享。


字符对象
行对象
列对象
appparent


column
row
row
row
flyweight pool


Glyph
Draw(Context)
Intersects(Point, Context)
children
Row
Draw(Context)
Intersects(Point, Context)
Character
Draw(Context)
Intersects(Point, Context)
char c
Column
Draw(Context)
Intersects(Point, Context)
children
外部状态，作为参量传入
Flyweight对象

Summary：Structural Patterns

- 结构型模式，关注的是对象之间组合的方式。
  - 本质上说，如果对象结构可能存在变化，主要在于其依赖关系的改变。对于结构型模式来说，处理变化的方式不仅仅是封装与抽象那么简单，还要合理地利用继承与聚合的方法，灵活地表达对象之间的依赖关系。
  - 例如Decorator模式，描述的就是对象间可能存在的多种组合方式，这种组合方式是一种装饰者与被装饰者之间的关系，因此封装这种组合方式，抽象出专门的装饰对象显然正是“封装变化”的体现。同样地，Bridge模式封装的则是对象实现的依赖关系，而Composite模式所要解决的则是对象间存在的递归关系。

Behavioral Patterns

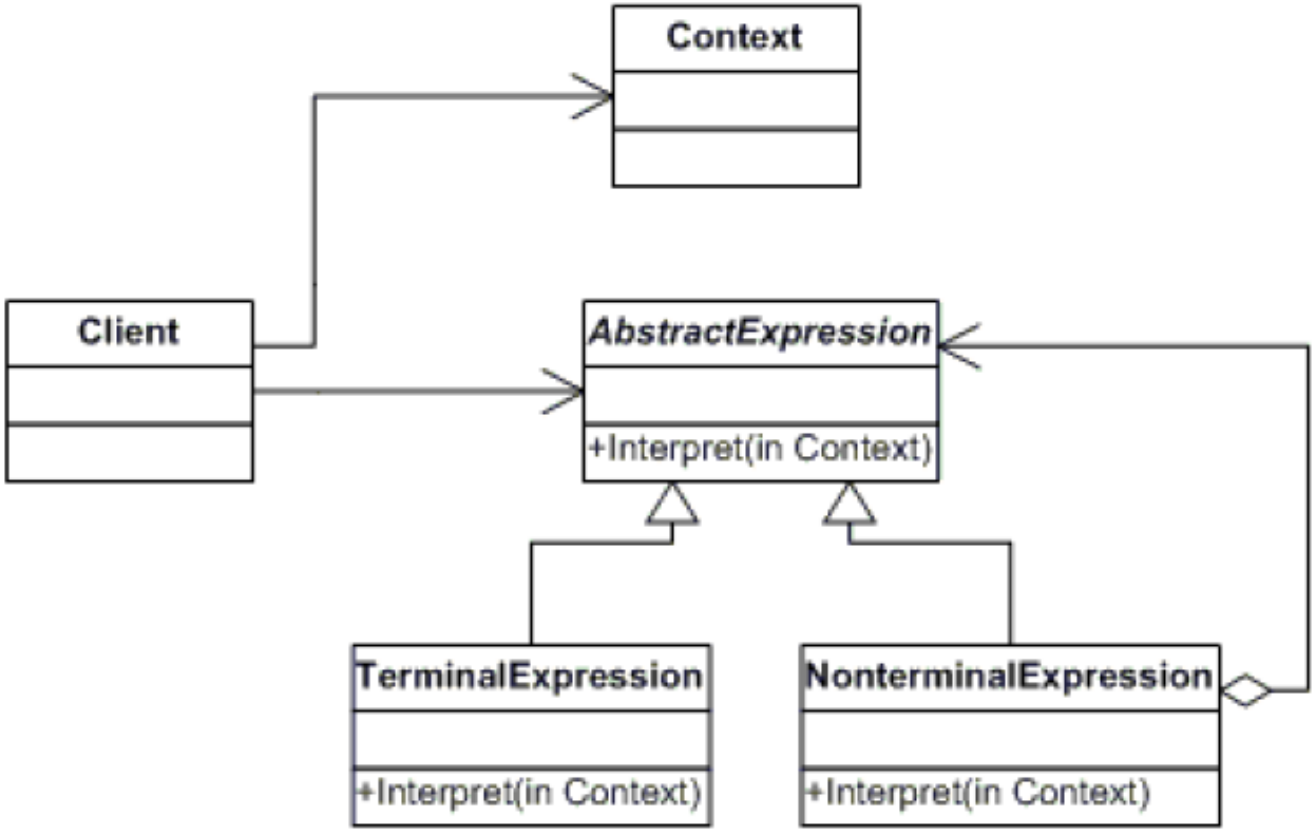
行为型模式关注的是对象的行为。

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy Flyweight	Chain of Responsibility Command Iterator Strategy Mediator Memento Observer State Visitor

13）解释器模式Interpreter

- 在软件构建过程中，如果某一特定领域的问题比较复杂，类似的模式不断重复出现，如果使用普遍的编程方式来实现将面临非常频繁的变化。
- 在这种情况下，将特定领域的问题表达为某种语法规则下的句子，然后构建一个解释器来解释这样的句子，从而达到解决问题的目的。即设计一种领域特定语言（Domain Specific Language, DSL），并为此开发一个解释器。
- 解释器模式描述了如何构成一个简单的语言解释器。

Interpreter模式的结构



- AbstractExpression(抽象表达式)
  - 声明一个抽象的解释操作，这个接口为抽象语法树中所有的节点所共享
- TerminalExpression(终结符表达式)
  - 实现与文法中的终结符相关联的解释操作
  - 一个句子中的每一个终结符需要该类的一个实例
- NonterminalExpression(非终结符表达式)
  - 对文法中的每一条规则R ::= R1R2. . . Rn都需要一个NonterminalExpression类
  - 为从R1到Rn的每个符号都维护一个AbstractExpression 类型的实例变量
  - 为文法中的非终结符实现Interpret操作，一般要递归地调用表示R1到Rn的那些对象的解释操作
- Context(环境类)
  - 包含解释器之外的一些全局信息
- Client(客户类)
  - 建造一个语法抽象树；调用interpret()

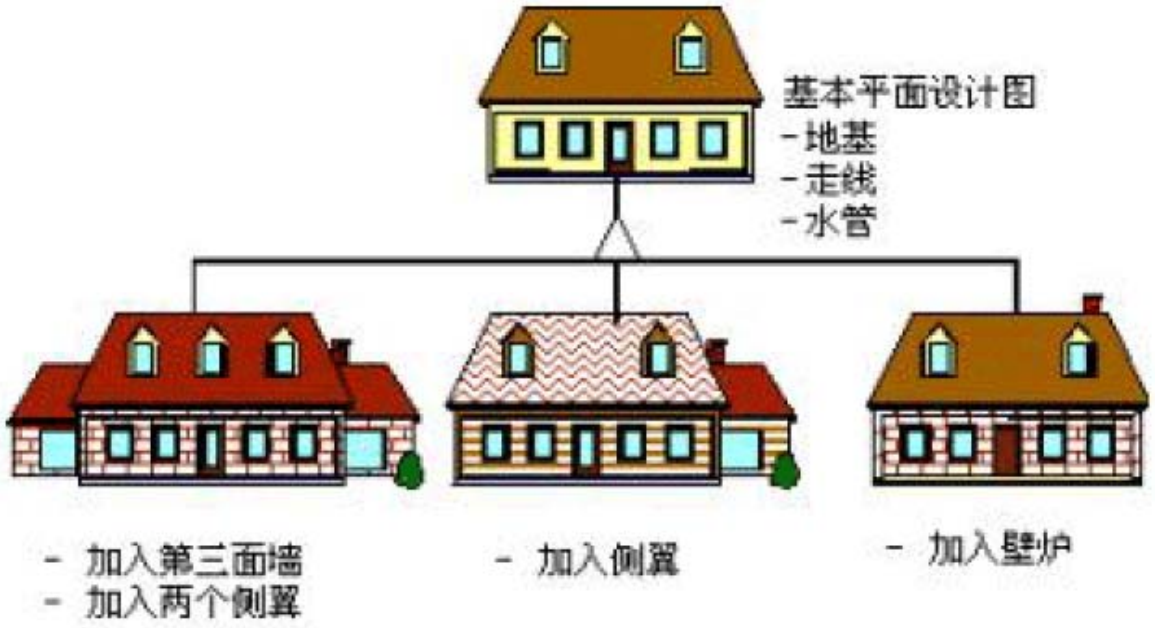
适用场景

- 适用性：
  - 系统有一个简单的语言可供解释；
  - 一些重复发生的问题可以用这种简单的语言表达；
  - 效率不是主要的考虑。

- 举例：
  - 规则语言的解释器
  - Smalltalk语言的编译器

14）模板方法模式Template Method

- 举例：



  - 房屋建筑师在开发新项目时，一个典型的规划包括一些建筑平面图，每个平面图体现了不同部分。在一个平面图中，地基、结构、上下水和走线对于每个房间都是一样的。只有在建筑的后期才开始有差别而产生了不同的房屋样式。

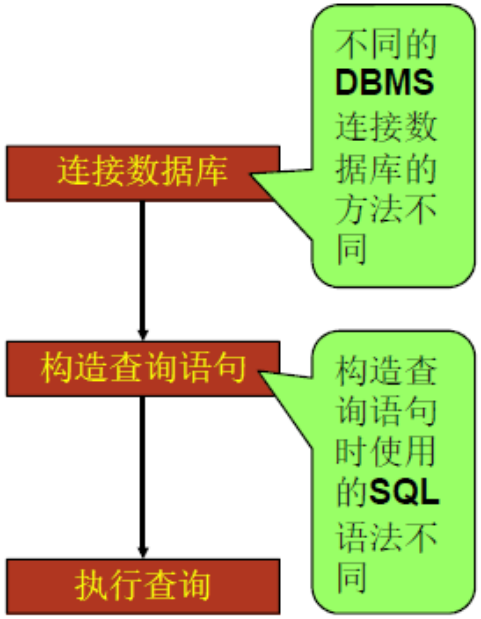
Template Method模式解决的问题

两类问题

- 问题1：复用问题

  我们经常会面临许多框架结构上相似，但是却在细节上稍有不同的代码

  - 例如：一个电子商务应用软件中，经常需要查询数据库，处理数据库中的数据。这个软件需要支持两种不同类型的DBMS——Oracle 和MS SQL Server。两个不同的DBMS虽然都支持SQL语句，但是其语句在语法上却略有不同。



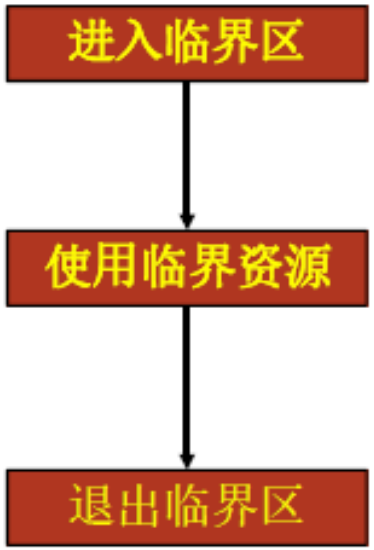
如何复用这样的代码？

两类问题

- 问题2：约束和规则

  程序中有些规则、约束，我们希望不会被违反

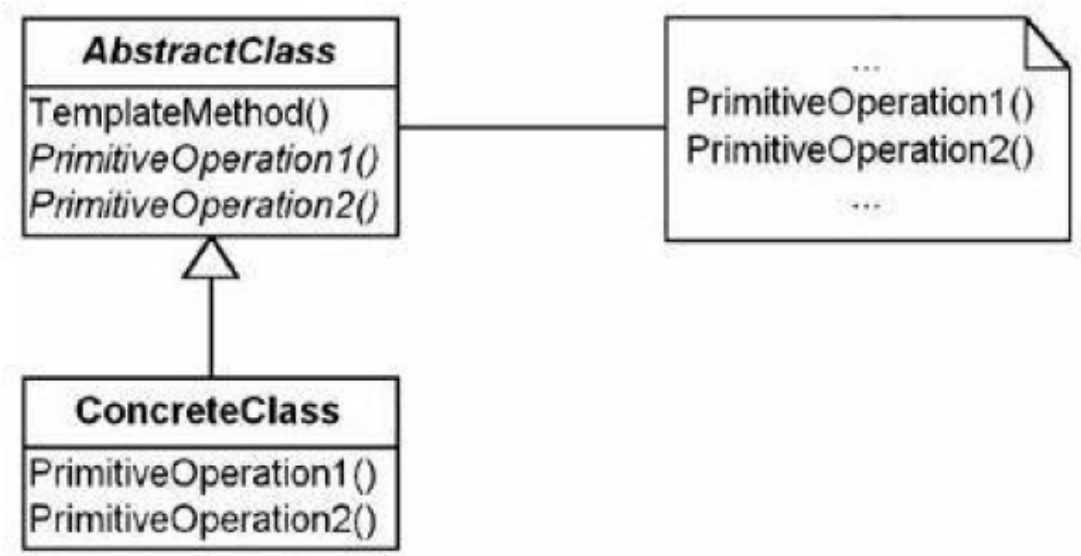
  - 例如：我们希望在使用一些临界资源之前一定要首先进入临界区，然后使用，使用完后一定要退出临界区。这是一个约束，我们希望所有的程序都要满足这一约束一种策略是：让所有使用临界资源的程序自己对约束负责。但是这样既不符合一个规则只实现一次的原则，也不能很好的保证约束不被违背



如何一次性编码这种约束，并使其对所有程序有效?

Template Method模式

- 思路：
  - 我们能够写一个Template，在其中编码了那些高层的策略、规则、流程。
  - 需要针对具体问题进行精化时，我们通过继承来扩展这个Template中的一些细节。



- AbstractClass(抽象类)
  - 定义抽象的原语操作(primitive operation)，具体的子类将重定义它们以实现一个算法的各步骤
  - 实现一个模板方法,定义一个算法的骨架。该模板方法不仅调用原语操作，也调用定义在AbstractClass或其他对象中的操作。
- ConcreteClass(具体子类)
  - 实现原语操作以完成算法中与特定子类相关的步骤

子类可以置换掉父类的可变部分，

但是子类却不可以改变模板方法所代表的顶级逻辑！

典型的抽象类代码

```
public abstract class AbstractClass
{
    public void templateMethod() //模板方法
    {
        primitiveOperation1();
        primitiveOperation2();
        primitiveOperation3();
    }
    public void primitiveOperation1() //基本方法—具体方法
    {
        //实现代码
    }
    public abstract void primitiveOperation2(); //基本方法—抽象方法
    public void primitiveOperation3() //基本方法—钩子方法
    {
    }
}
```

典型的具体子类代码

```
public class ConcreteClass extendsAbstractClass
{
    public void primitiveOperation2()
    {
        //实现代码
    }
    public void primitiveOperation3()
    {
        //实现代码
    }
}
```

模式中的方法

- 模板方法：一个模板方法是定义在抽象类中的、把基本操作方法组合在一起形成一个总算法或一个总行为的方法。
- 基本方法：基本方法是实现算法各个步骤的方法，是模板方法的组成部分。
  - 抽象方法(Abstract Method)
  - 具体方法(Concrete Method)
  - 钩子方法(Hook Method)：“挂钩”方法和空方法

钩子方法

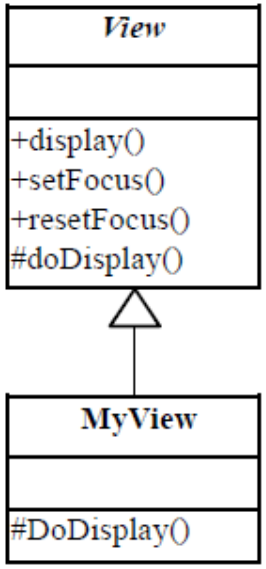
- 在模板方法模式中，钩子方法有两类：
  - 1. 钩子方法是实现为空的具体方法，子类可以根据需要覆盖或者继承这些钩子方法。
    - 如果没有覆盖父类中定义的钩子方法，编译可以通过；但如果没有覆盖父类中定义的抽象方法，编译将报错。
  - 2. 钩子方法与一些具体步骤“挂钩”，以确定在不同条件下执行模板方法的不同步骤，该类钩子方法的返回类型通常是boolean类型，用于对某条件进行判断，如果条件满足则执行某一步骤，否则不执行。

```
……
public void template()
{
   open();
   display();
   if(isPrint())
   {
     print();
   }
}
public boolean isPrint()
{
   return true;
}
……
```

该类钩子方法可以控制方法的执行，对算法进行约束

举例：绘图

- 一个支持在屏幕上绘图的类View。一个View获得焦点之后才能设置特定的图形设备环境（如颜色、字体等），因而只有获得焦点后才能绘图。这是一个约束（或者规则），如何编码这一约束，使得这一约束能够很容易的被遵守。



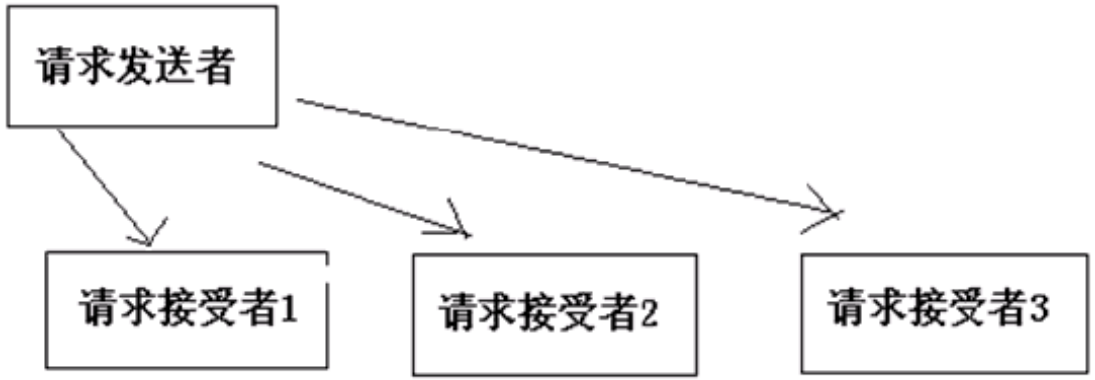

```
void View::Display(){
    setFocus();
    doDisplay();
    resetFocus();
}

view的doDisplay函数定义为纯
    虚函数

void MyView::doDisplay(){
    //render the view's contents
}
```

实际应用

- 在很多企业中都有，叫法不一：
  - 平台
  - 领域中间件
  - 框架
  - 行为架构
  - ……
- 一般是长期积累的，定义之后，用时只需要写不同的地方
- 典型应用：
  - 针对某个行业的，如用友、金蝶、神州数码等，不同项目间有很多相似性，将共性的东西定义出来，将扩展点组装在一起。
- 如何扩展？
  - 用继承

15) 责任链模式Chain of Responsibility

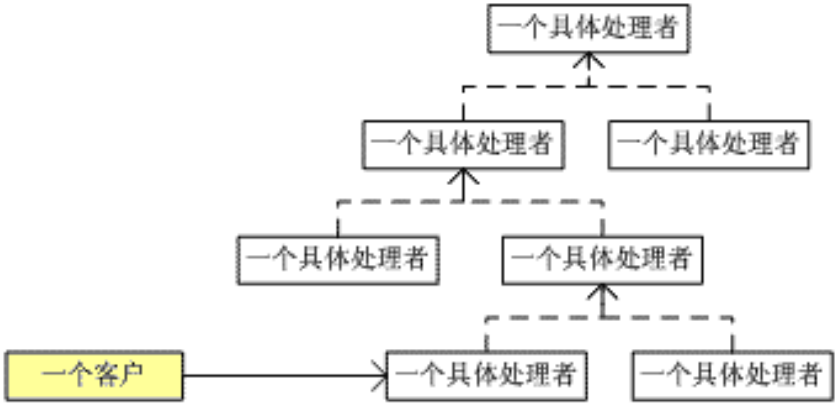
- 某些对象请求的接受者可能有多种，但在运行时只能有一个接受者，如果显式指定，将必不可少地带来请求发送者与接受者的紧耦合。



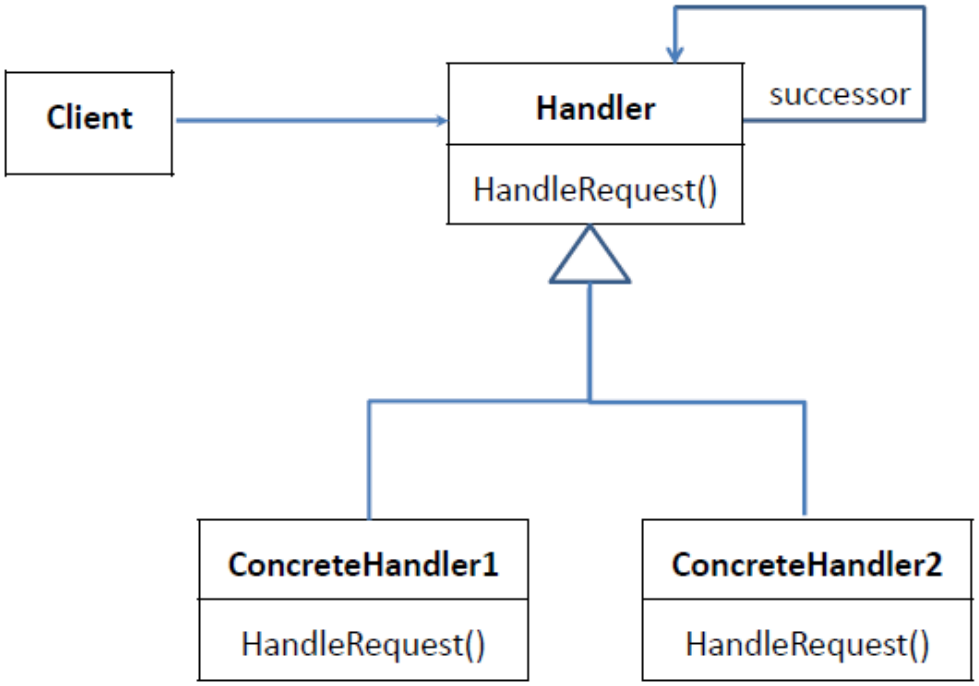
- 如何使请求的发送者不需要指定具体的接受者？让请求的接受者自己在运行时决定来处理请求，从而使二者解藕。

责任链模式

- 在责任链模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。
- 发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织和分配责任。
- 职责链可以是一条直线、一个环或者一个树形结构，最常见的职责链是直线型，即沿着一条单向的链来传递请求。



责任链模式的结构



- Handler
  - 定义一个处理请求的接口
  - (可选)实现后继链
- ConcreteHandler
  - 处理它所负责的请求
  - 可访问它的后继者
  - 如果可处理该请求，就处理之；否则将该请求转发给它的后继者
- Client
  - 提交请求

纯的与不纯的责任链模式

- 一个纯的责任链模式要求一个具体的处理者对象只能在两个行为中选择一个：一个是承担责任，二是把责任推给下家。不允许出现某一个具体处理者对象在承担了一部分责任后又把责任向下传的情况。
- 在一个纯的责任链模式里面，一个请求必须被某一个处理者对象所接收；在一个不纯的责任链模式里面，一个请求可以最终不被任何接收端对象所接收。纯的责任链模式的例子不多见，一般的例子均是不纯的责任链模式的实现。

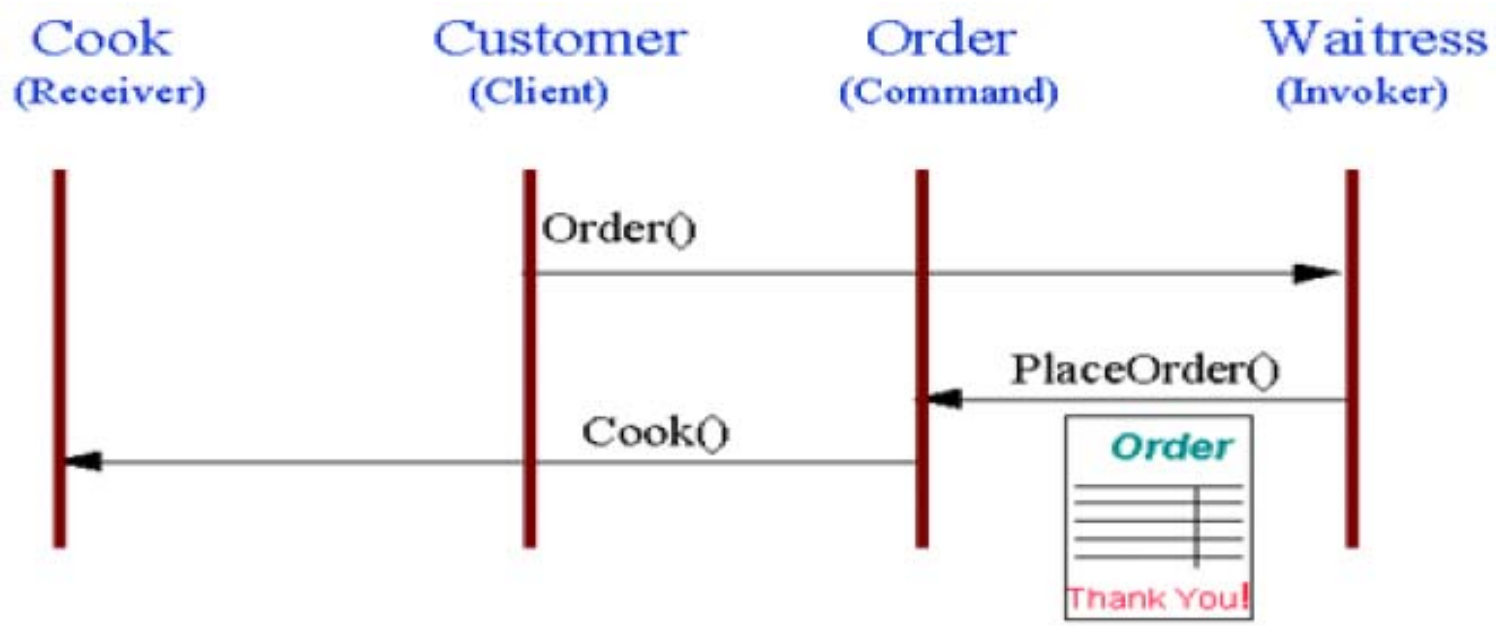
适用场景

- 适用性
  - 有多个对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定。
  - 你想在不明确接收者的情况下，向多个对象中的一个提交一个请求。
  - 可处理一个请求的对象集合应被动态指定。
- 应用实例
  - JS 中的事件冒泡
  - JAVA WEB 中 Apache Tomcat 对 Encoding 的处理，Struts2 的拦截器，jsp servlet 的 Filter。

16）命令模式Command

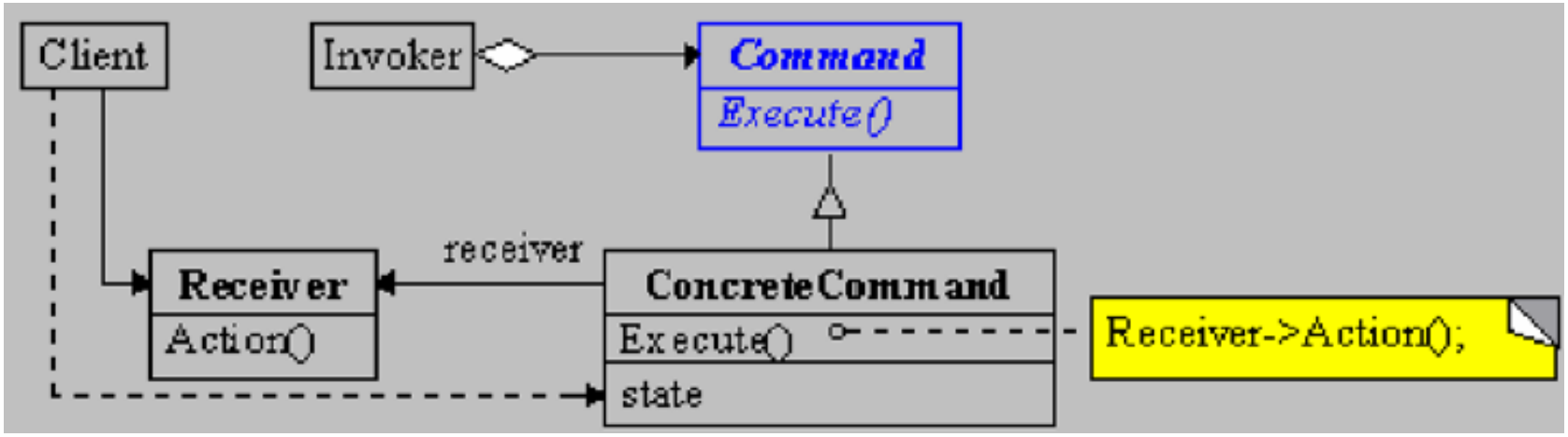
- 耦合是软件不能抵御变化灾难的根本性原因。不仅实体对象与实体对象之间存在耦合关系，实体对象与行为操作之间也存在耦合关系。
  - 例如：要对行为操作进行“记录、撤销/重做、事务” 等处理，这种无法抵御变化的紧耦合是不合适的。
- Command模式将一组行为操作抽象为命令对象，可以实现实体对象与行为操作之间的松耦合。
  - 命令对象可以进行参数化、排队处理、撤销/重做、日志记录等。
  - Command模式封装的是命令，把命令发出者的责任和命令执行者的责任分开。

生活中的例子：



- Command模式将一个请求封装为一个对象。用餐时的Order是Command模式的一个例子。服务员接受顾客的点单，把它记在Order上封装。这个Order被排队等待烹饪。注意这里的Order不是菜单类的方法，而是独立的类，它可以被不同的顾客使用，可以添入不同的点单项目。

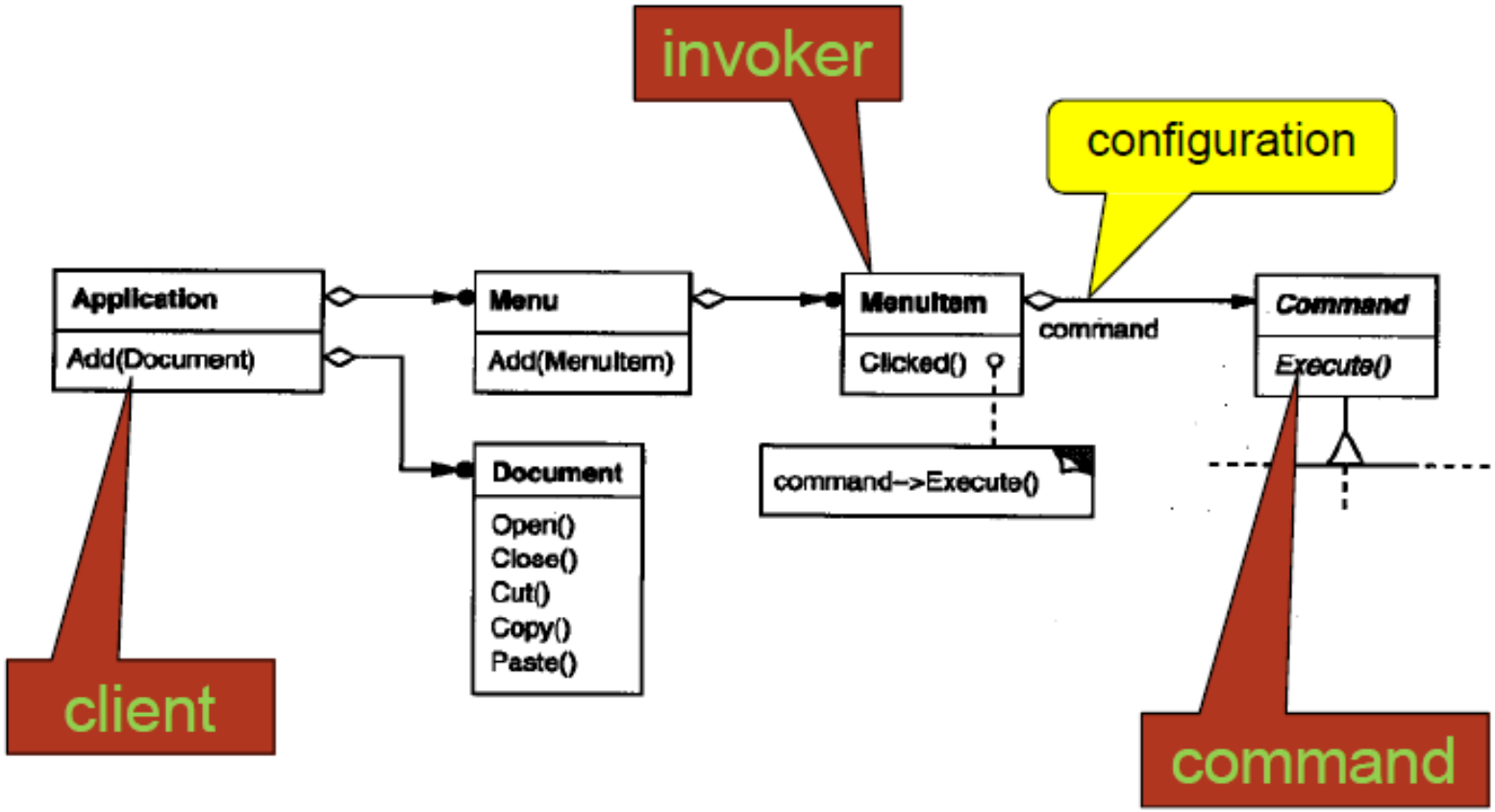
命令模式的结构



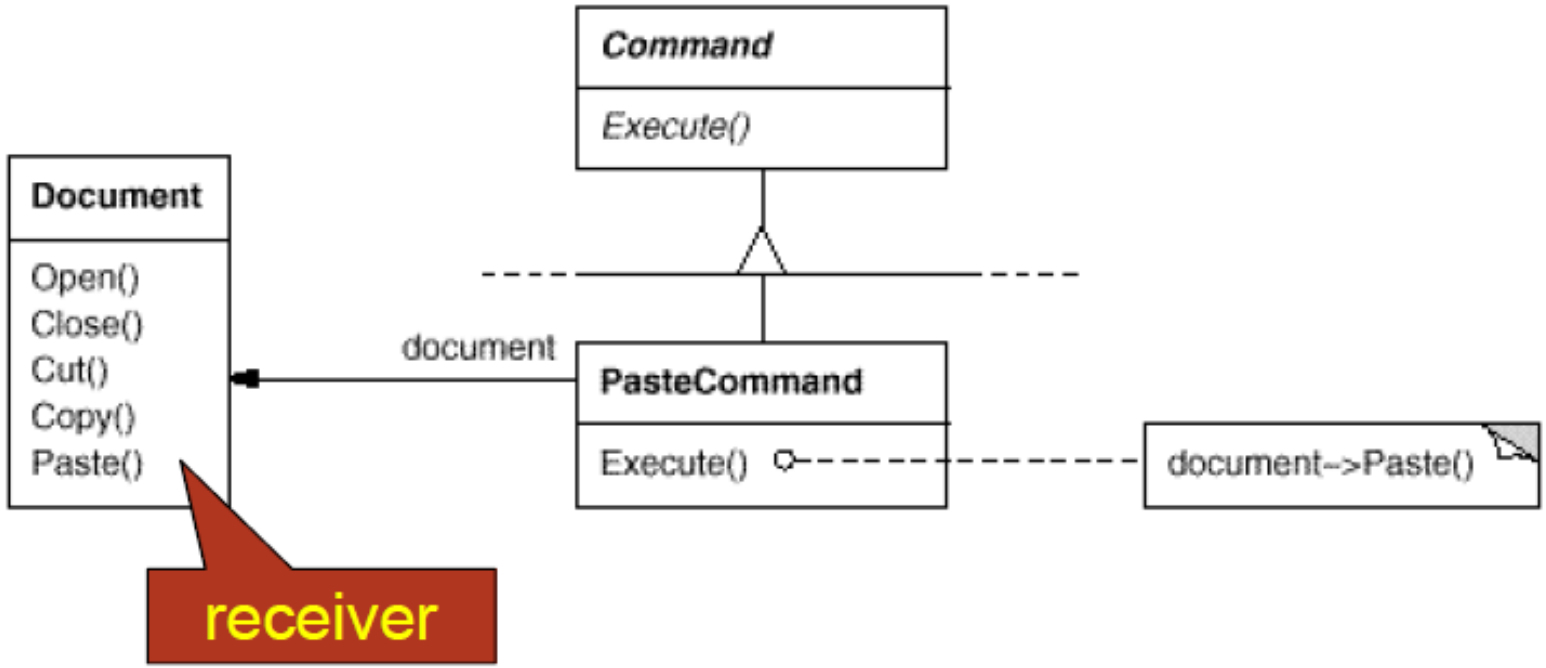
- Command (抽象命令类)：声明执行操作的接口
- ConcreteCommand (具体命令类)：将一个接收者对象绑定于一个动作；调用接收者相应的操作以实现Execute
- Client (客户类)：创建一个具体命令对象并设定它的接收者
- Invoker (调用者)：要求该命令执行这个请求
- Receiver (接收者)：知道如何实施与执行一个请求相关的操作。任何类都可能作为一个接收者

举例：

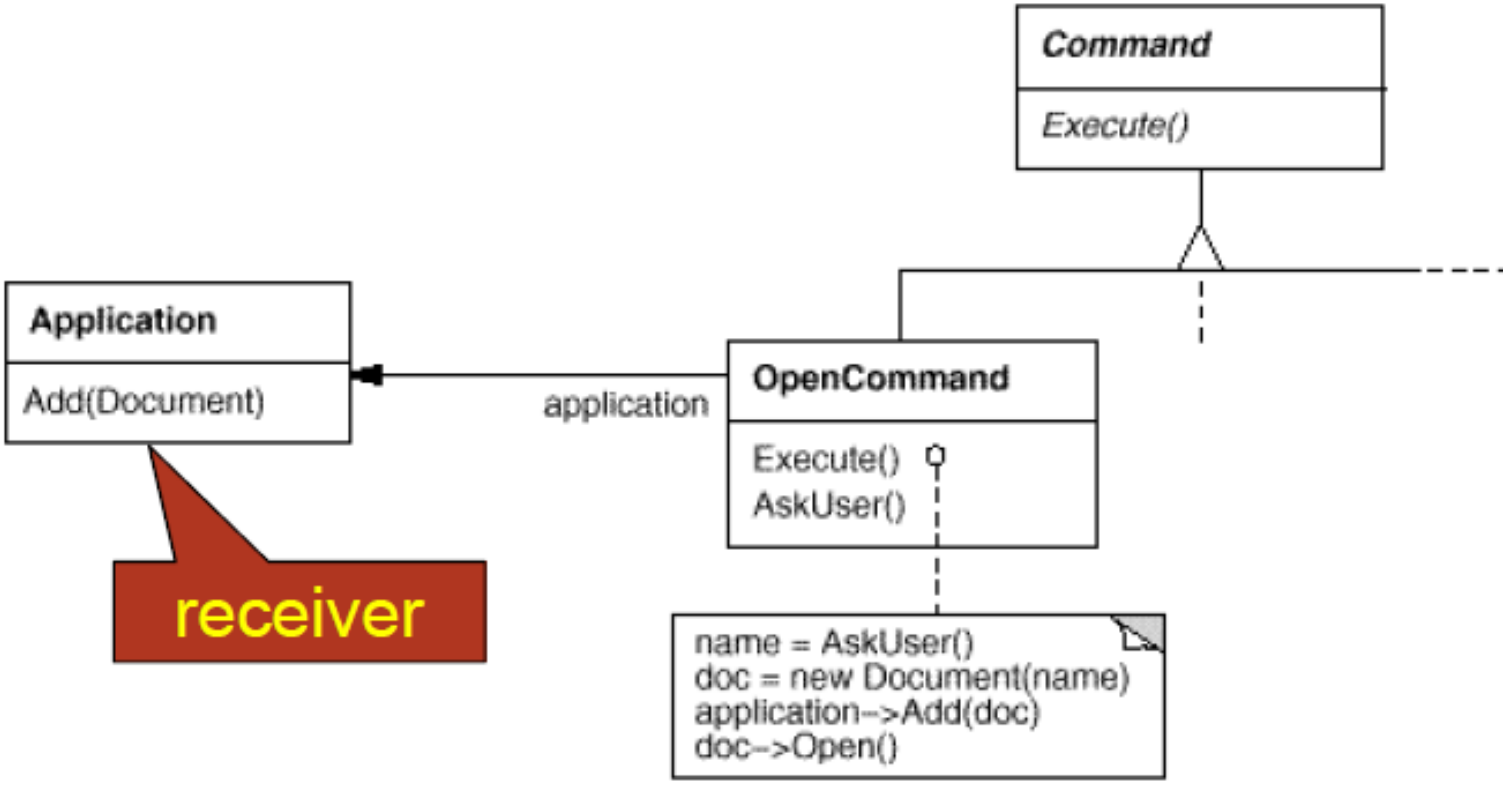
- 菜单命令是Command模式的经典应用。如何设计才能在用户单击菜单的时候调用对应的方法？如：
  - 从剪贴板向一个文档(Document)粘贴正文。
  - 打开一个文档。
  - 使一个页面按正常大小居中。
  - ……



- 每一菜单中的选项都是一个菜单项单项时，该menuItem对象调用它的command对象的Execute方法，而Execute执行相应操作。menuItem对象并不知道它们使用的是command的哪一个子类。command子类里存放着请求的接收者，而Execute操作将调用该接收者的一个或多个操作。

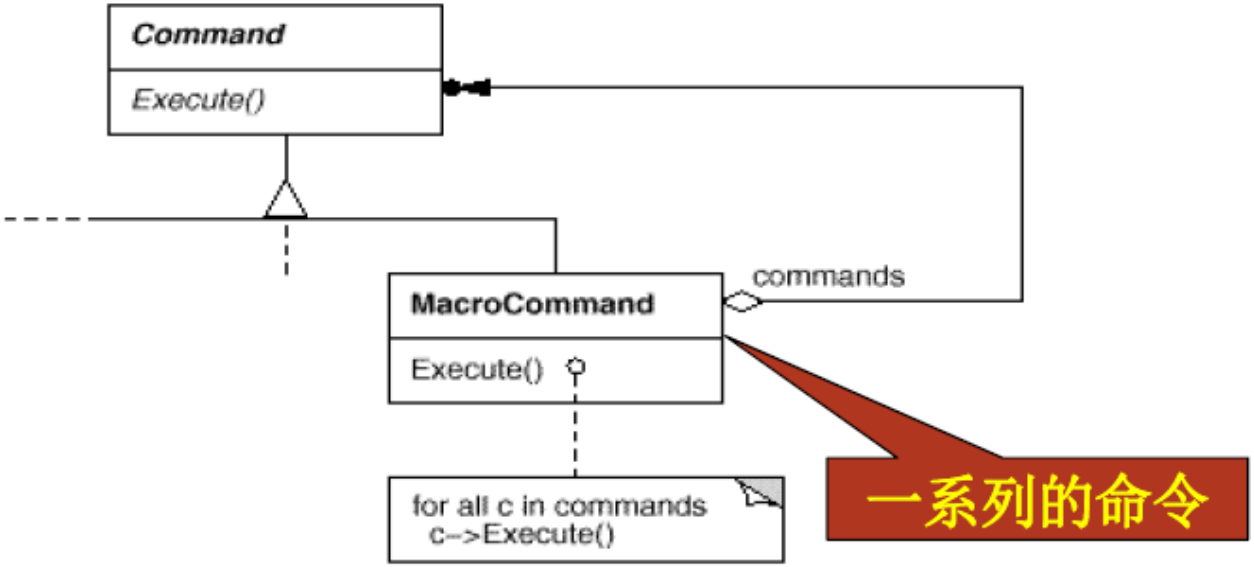


- PasteCommand支持从剪贴板向一个文档(Document)粘贴正文。PasteCommand的接收者是一个文档对象，该对象是实例化时提供的。Execute操作将调用该Document的Paste操作。



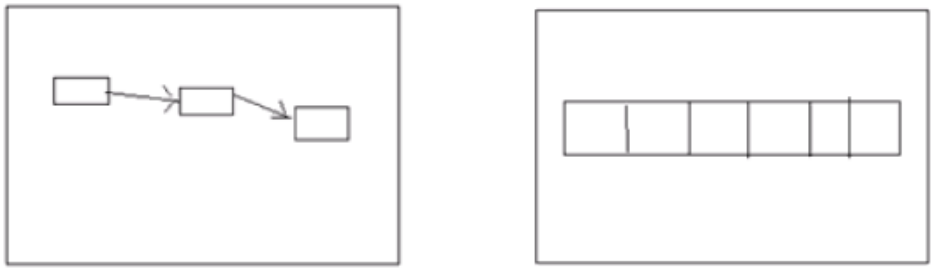
- OpenCommand的Execute操作有所不同：它提示用户输入一个文档名，创建一个相应的文档对象，将其作为接收者的应用对象中，并打该文档。

- 有时一个MenuItem需要执行一系列命令。如，使一个页面按正常大小居中的MenuItem可由一个CenterDocumentCommand对象和一个NormalSizeCommand对象构建。
- 由于这种需要将多条命令串接起来的情况很常见，我们定义一个MacroCommand类来让一个MenuItem执行任意数目的命令。
- MacroCommand是一个具体的Command子类，它执行一个命令序列。MacroCommand没有明确的接收者，而序列中的命令各自定义其接收者。



17）迭代器模式Iterator

- 聚集对象



  - 在软件构建过程中，聚集对象内部结构常常变化各异。但对于这些聚集对象，我们希望在不暴露其内部结构的同时，可以让外部客户代码透明地访问其中包含的元素。
  - 聚集对象拥有两个职责：
    - 存储内部数据
    - 遍历内部数据

  前者为根本属性；后者既是可变化的、可分离的。

- 因此，可以将遍历行为分离出来，抽象为一个迭代器，专门提供遍历聚集内部数据对象的行为，这就是迭代器模式的本质。

举例

我们编写代码经常都会碰到遍历一个数组，使用

For 循环得到数组下标之后去做进一步操作, 如：

```
int[] array=new int[5];
for(int i=0; i<array.length; i++)
{
    System.out.println(""+i);
}
```

如果将i 的行为，抽象化为迭代器，这种模式称之为迭代器模式。

迭代器模式可以用来作为遍历一个聚集体。

迭代器模式解决的问题

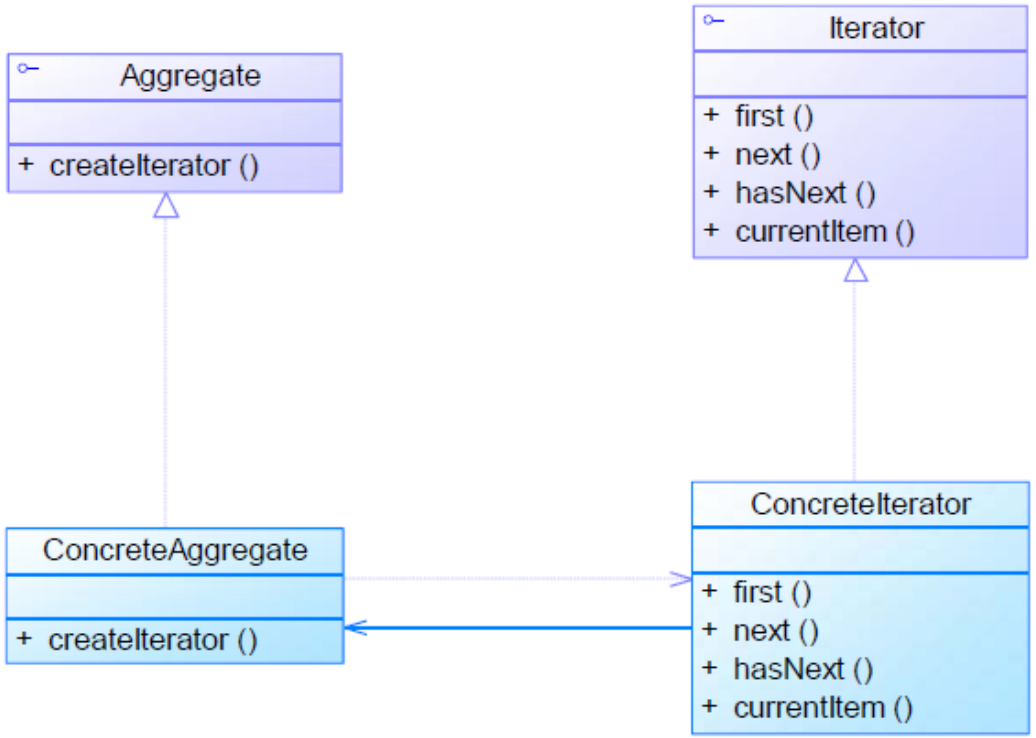
- 一个聚集对象，如列表，应该提供一种方法来让别人可以访问它的元素，而又不需暴露它的内部结构；
- 针对不同的需要，可能要以不同的方式遍历这个列表；
- 即使可以预见所需的遍历操作，可能也不希望列表的接口中充斥着各种不同遍历的操作；
- 有时还需要在同一个表列上同时进行多个遍历。

迭代器模式通过将列表的访问和遍历从列表对象中分离出来并放入一个迭代器对象中。迭代器类定义了一个访问该列表元素的接口。迭代器对象负责跟踪当前的元素，即知道那些元素已经遍历过了。

- 当我们说聚集（aggregate）的时候，我们指的是一群对象。其存储方式可以是各式各样的数据结构，例如：列表、数组、散列表，无论用什么方式存储，一律可以视为聚集。有时候也被称为“聚集（Collection）” 。

- 迭代器（Iterator）模式把元素之间游走的任务交给了迭代器，而不是聚集对象。这不仅让聚集的接口和实现变得更简洁，也让它专注于管理对象，而不必理会遍历的事情。

迭代器模式的结构



- Iterator(抽象迭代器)：迭代器定义访问和遍历元素的接口
- ConcreteIterator(具体迭代器)：具体迭代器实现迭代器接口，对该聚集遍历时跟踪当前位置
- Aggregate(抽象聚集类)：聚集定义创建相应迭代器对象的接口
- ConcreteAggregate(具体聚集类)：具体聚集实现创建相应迭代器的接口，该操作返回ConcreteIterator的一个适当的实例。

模式分析

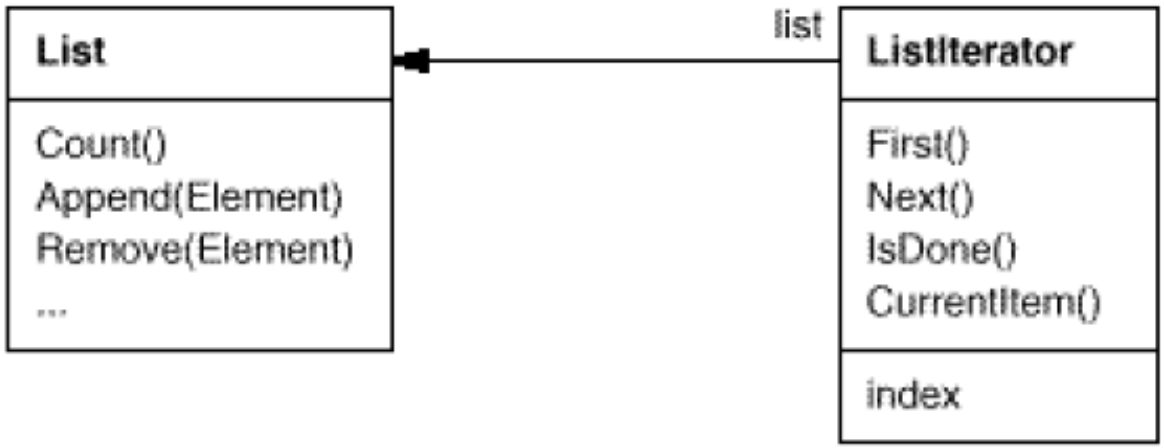
- 在迭代器模式中应用了工厂方法模式，聚集类充当工厂类，而迭代器充当产品类，由于定义了抽象层，系统的扩展性很好，在客户端可以针对抽象聚集类和抽象迭代器进行编程。
- 由于很多编程语言的类库都已经实现了迭代器模式，因此在实际使用中我们很少自定义迭代器，只需要直接使用Java、C#等语言中已定义好的迭代器即可，迭代器已经成为我们操作聚集对象的基本工具之一。

迭代器的实现

- 实现很灵活，需要根据语言的控制结构进行权衡
  - 谁控制迭代？谁定义遍历算法？
  - 迭代器健壮程度如何？
  - 附加的迭代器操作
  - 多态的迭代器？
  - 迭代器可有特权访问
  - 用于复合对象的迭代器，空迭代器

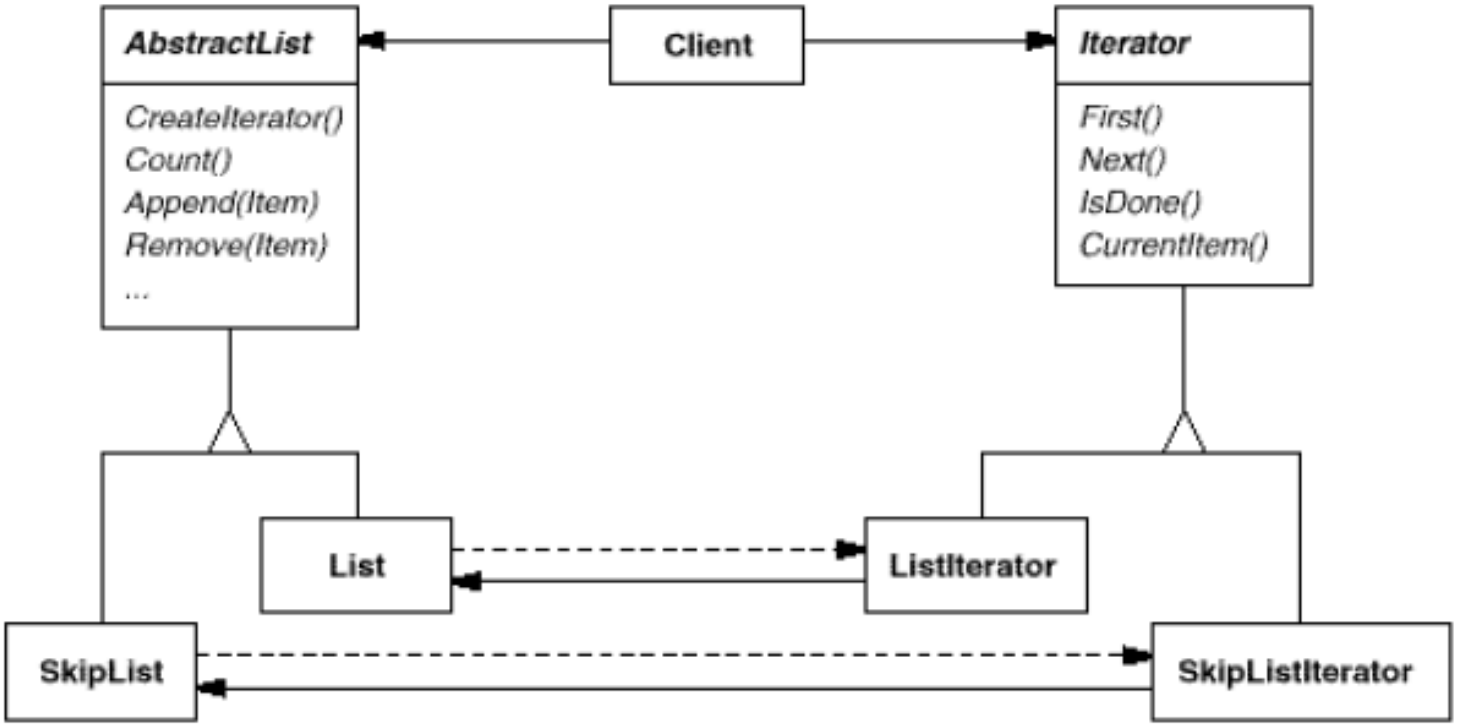
列表迭代器

```
public interface Iterator<E>{
    boolean hasNext();
    <E> next();
    void remove();
}
```



在实例化列表迭代器之前，必须提供待遍历的列表。一旦有了该列表迭代器的实例，就可以顺序地访问该列表的各个元素。CurrentItem操作返回表列中的当前元素，First操作初始化迭代器，使当前元素指向列表的第一个元素，Next操作将当前元素指针向前推进一步，指向下一个元素，而IsDone检查是否已越过最后一个元素，也就是完成了这次遍历。

多态迭代器Polymorphic Iterator



定义一个抽象列表类AbstarctList，它提供操作列表的公共接口；定义一个抽象的迭代器类Iterator，提供公共的迭代接口，然后为每个不同的列表实现定义具体的Iterator子类，使迭代机制与具体的聚集类无关。列表对象中提供CreatorIterator操作，客户请求调用该操作获得一个迭代器对象。

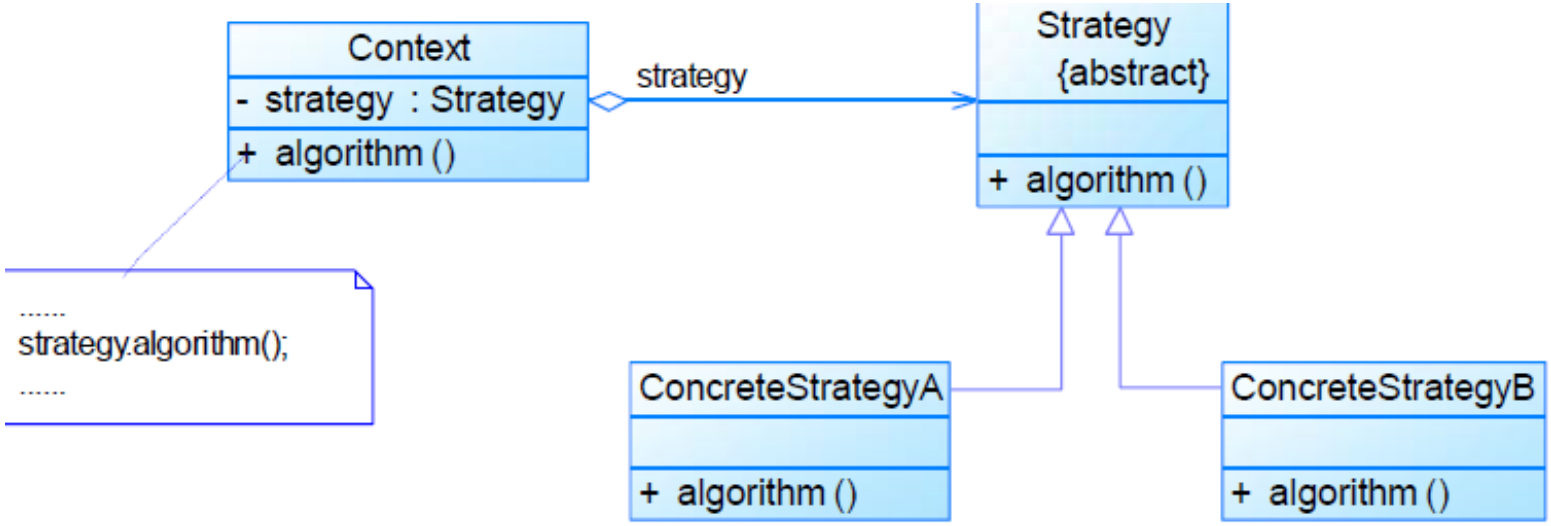
其它考虑

- 主动(Active)迭代器 vs. 被动(Passive)迭代器
  - 主动：由客户调用next()等迭代方法
  - 被动：迭代器自行推进遍历过程
- 静态迭代器vs. 动态迭代器
  - 静态：由聚集对象创建并持有聚集对象的快照，在产生后这个快照的内容不再变化
  - 动态：迭代器保持对聚集元素的引用，任何对聚集内容的修改都会反映到迭代器对象上
- 过滤迭代器：扫过聚集元素的同时进行计算

18）策略模式Strategy

- 在软件构建过程中，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，将会使对象变得异常复杂；而且有时侯支持不使用的算法也是一个性能负担。
- 如何在运行时根据需要透明地更改对象的算法？将算法与对象本身解耦，从而避免上述问题？
- Strategy模式对算法进行封装
  - 它把算法的责任和算法本身分割开，委派给不同的对象管理。策略模式通常把一个系列的算法封装到一系列的策略类里面，作为一个抽象策略类的子类。
  - 简言之，就是“准备一组算法，并将每一个算法封装起来，使得它们可以互换”。

策略模式的结构



- Strategy(抽象策略类)：定义所有支持的算法的公共接口。Context使用这个接口来调用某ConcreteStrategy定义的算法。
- ConcreteStrategy(具体策略类)：Strategy接口实现某具体算法。
- Context(环境类)：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。

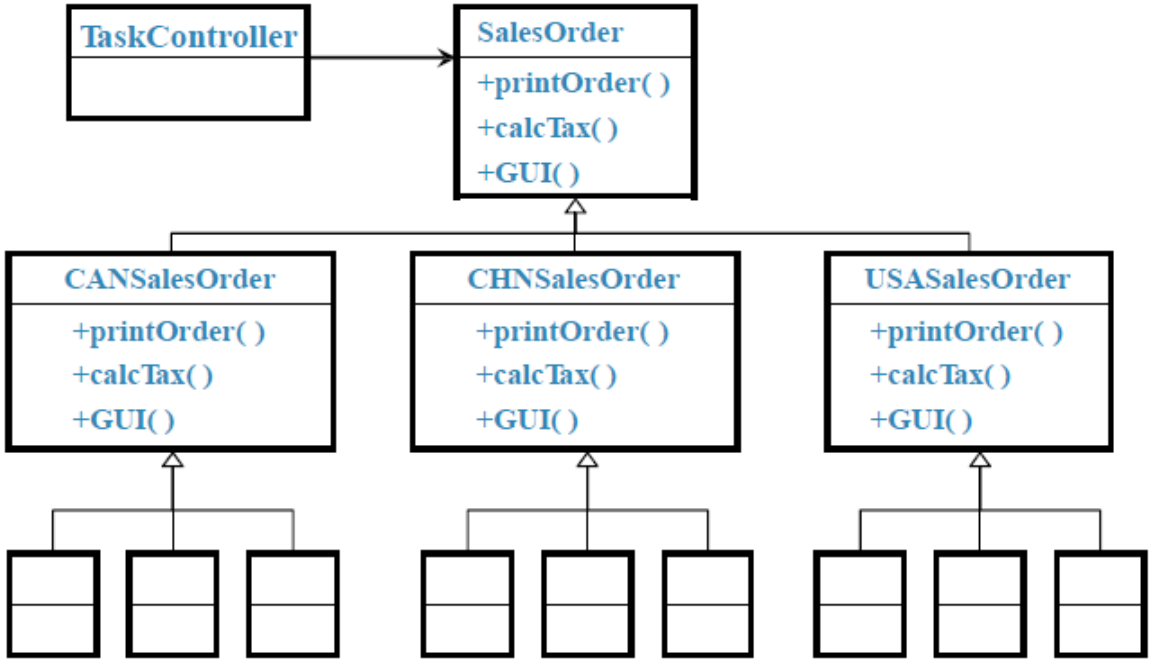
模式分析

- 在策略模式中，应当由客户端自己决定在什么情况下使用什么具体策略角色。
- 策略模式仅仅封装算法，提供新算法插入到已有系统中，以及老算法从系统中“退休”的方便，策略模式并不决定在何时使用何种算法，算法的选择由客户端来决定。这在一定程度上提高了系统的灵活性，但是客户端需要理解所有具体策略类之间的区别，以便选择合适的算法，这也是策略模式的缺点之一，在一定程度上增加了客户端的使用难度。

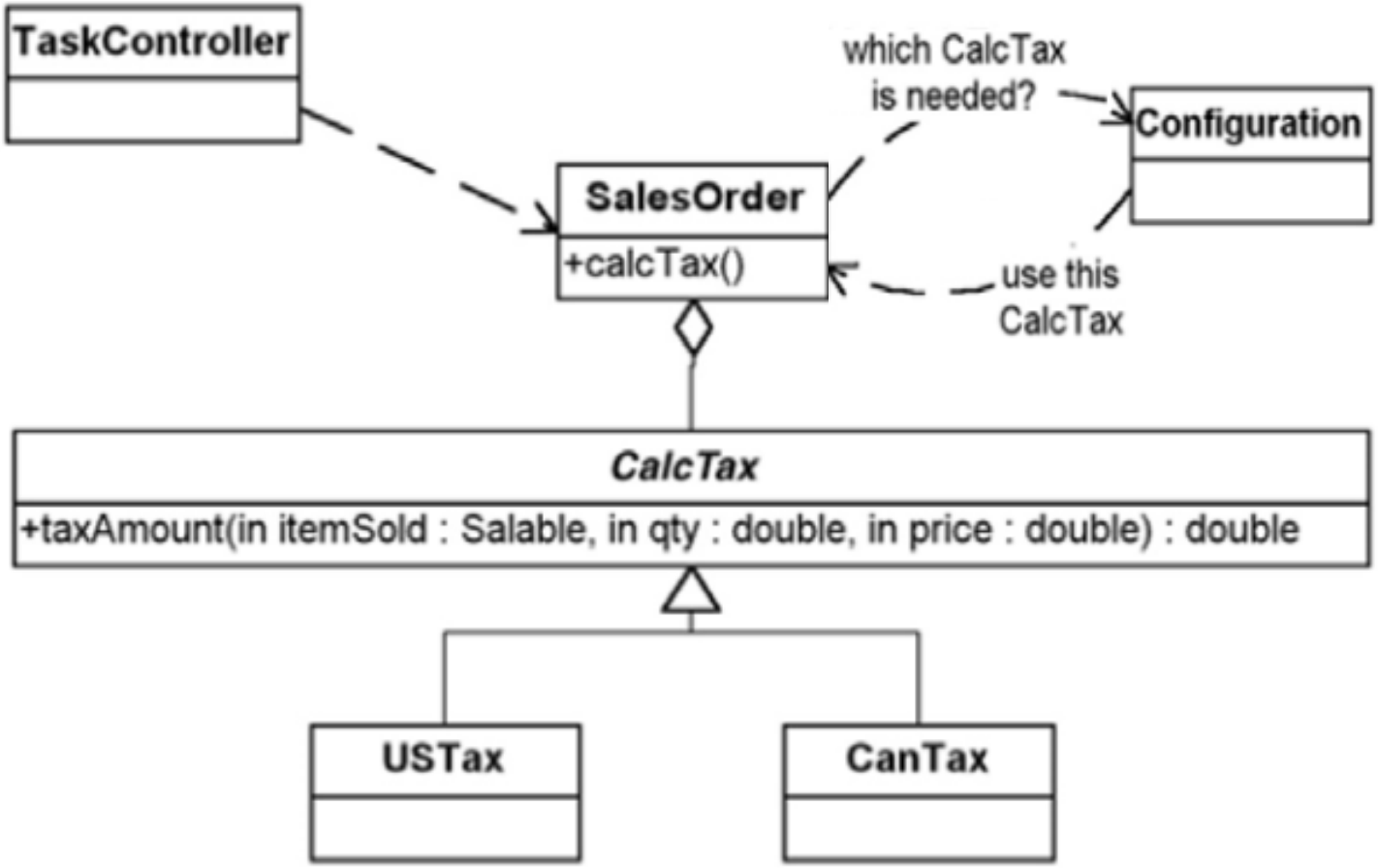
- 新的需求
  - 要处理多种税额计算的方法。
  - 例如要处理美国、加拿大、中国三个国家的税收方法
  - （税法不同计税方法就不同）
- 应对策略：
  - 复制 + 粘贴 + 修改
  - 使用Switch语句
  - 使用继承

前两种方法的问题不言而喻，我们肯定不会采用
看起来，继承是一种不错的方案，我们看看……

- 现在又出现新的变化，希望在打印收据时有大单（A4纸）、小单（B5纸）和固定票据（在一个固定格式的空票据上打印）三种格式。
- 为了应对上述需求，我们修改上述设计。

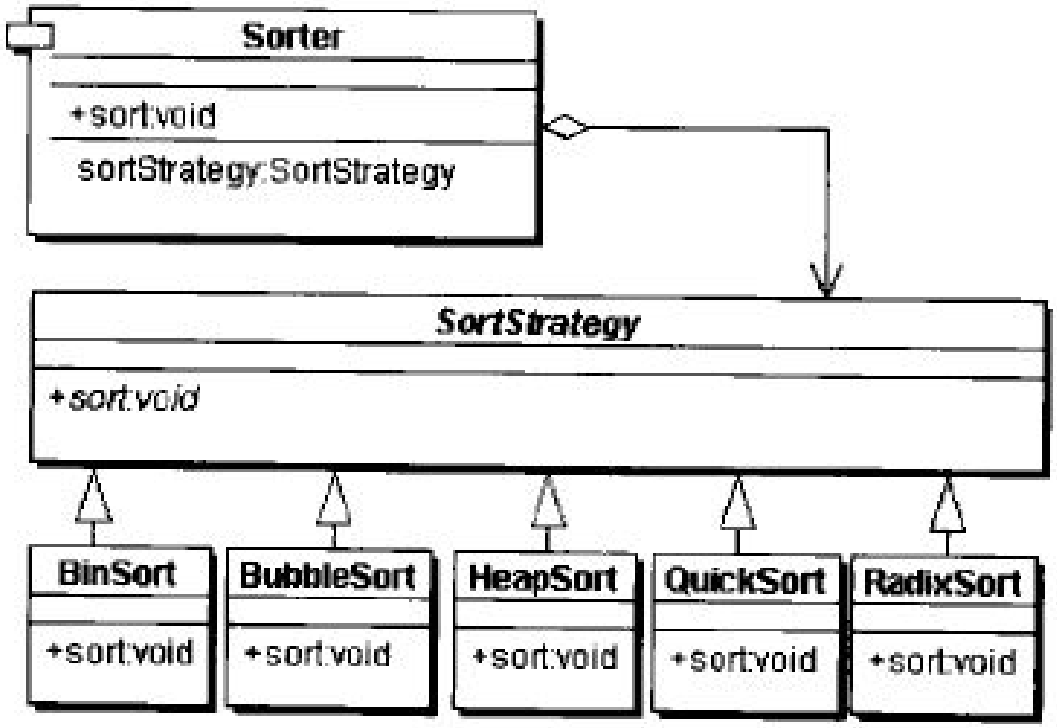


采用Strategy模式!



举例：排序系统

- 现设计一个排序系统，由客户代码动态地决定采用二元排序、冒泡排序、选择排序、插入排序、快速排序。

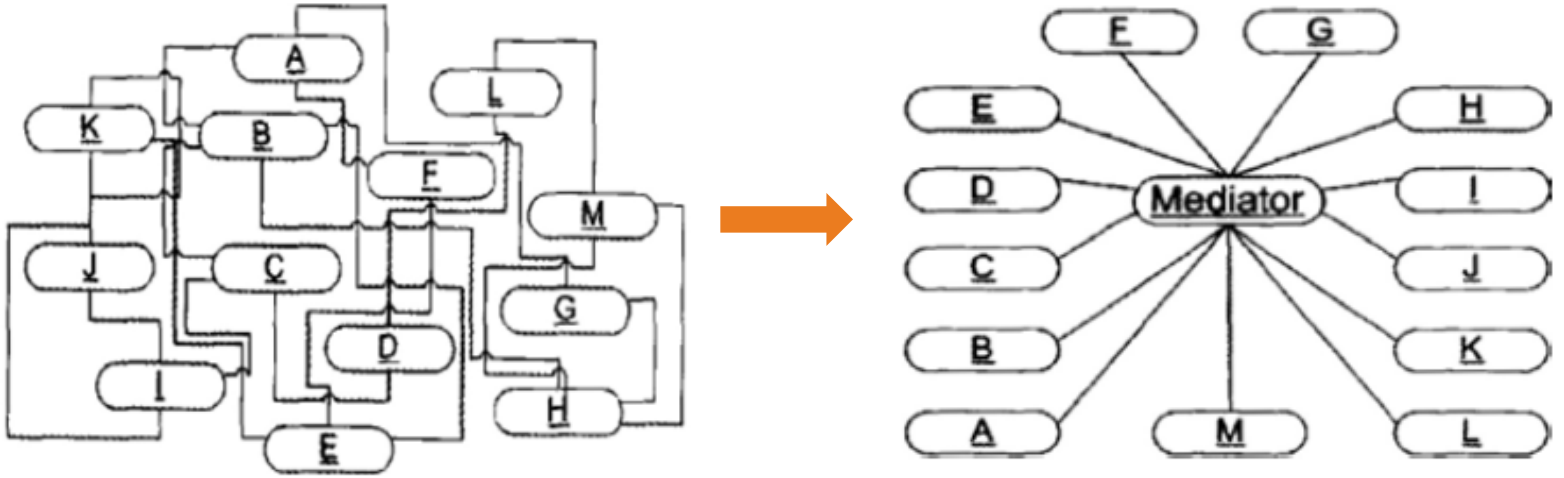


适用场景

- 在下面的情况下应当考虑使用策略模式:
  1. 如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为。
  2. 一个系统需要功态地在几种算法中选择一种。那么这些算法可以包装到一个个的具体算法类里面，而这些具体算法类都是一个抽象算法类的子类。换言之，这些具体算法类均有统一的接口，由于多态性原则，客户端可以选择使用任何一个具体算法类，并只持有一个数据类型是抽象算法类的对象。
  3. 一个系统的算法使用的数据不可以让客户端知道。策略模式可以避免让客户端涉及到不必要接触到的复杂的和只与算法有关的数据。
  4. 如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。此时，使用策略模式，把这些行为转移到相应的具体策略类里面，就可以避免使用难以维护的多重条件选择语句，并体现面向对象设计的概念。

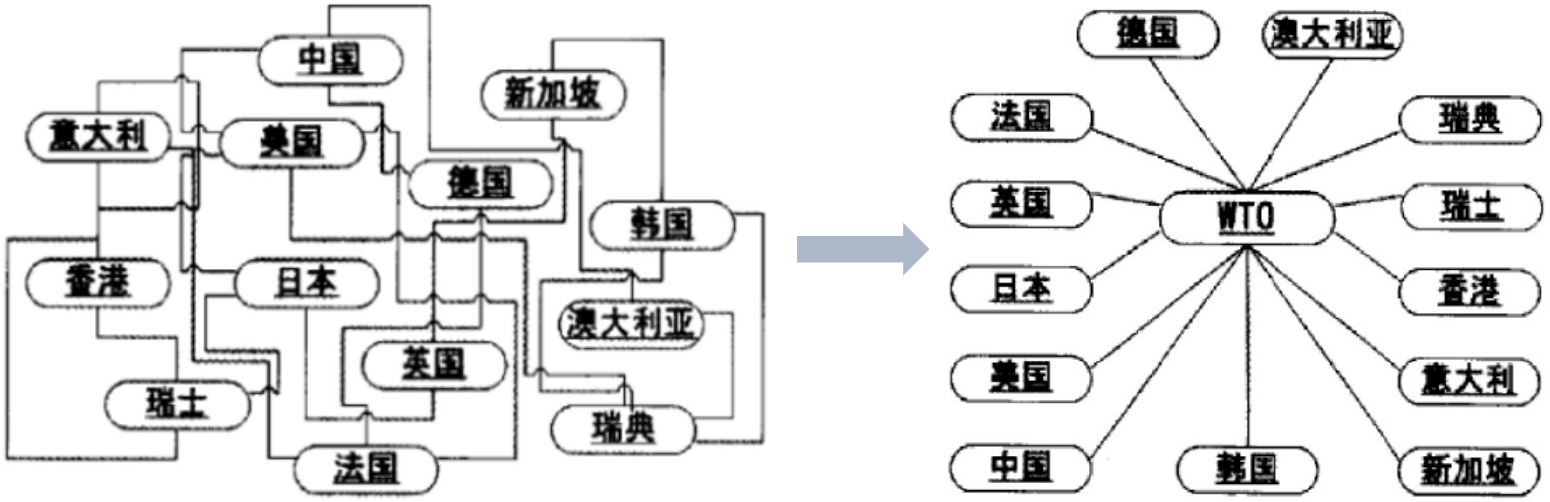
19）中介者模式Mediator

- 在软件构建过程中，经常会出现多个对象互相关联交互的情况，对象之间常常会维持一种复杂的引用关系，如果遇到一些需求的更改，这种直接的引用关系将面临不断的变化。
- 在这种情况下，我们可使用一个“中介对象”来管理对象间的关联关系，避免相互交互的对象之间的紧耦合引用关系，从而更好地抵御变化。

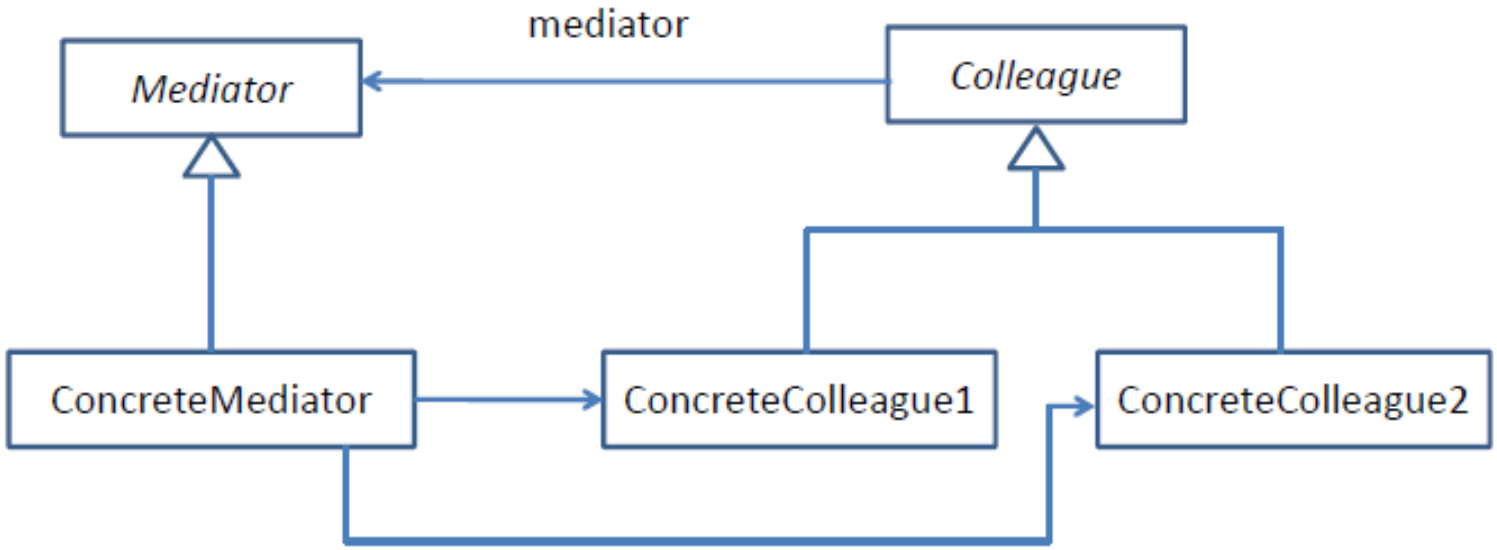


生活中的例子：WTO

- WTO是个协调组织，各个贸易地区可以经由WTO 组织进行协调。WTO 扮演的正是中介者模式中的中介者角色，它取代了原本有各个贸易地区自行进行的相互协调和谈判的强耦合关系。



中介者模式的结构

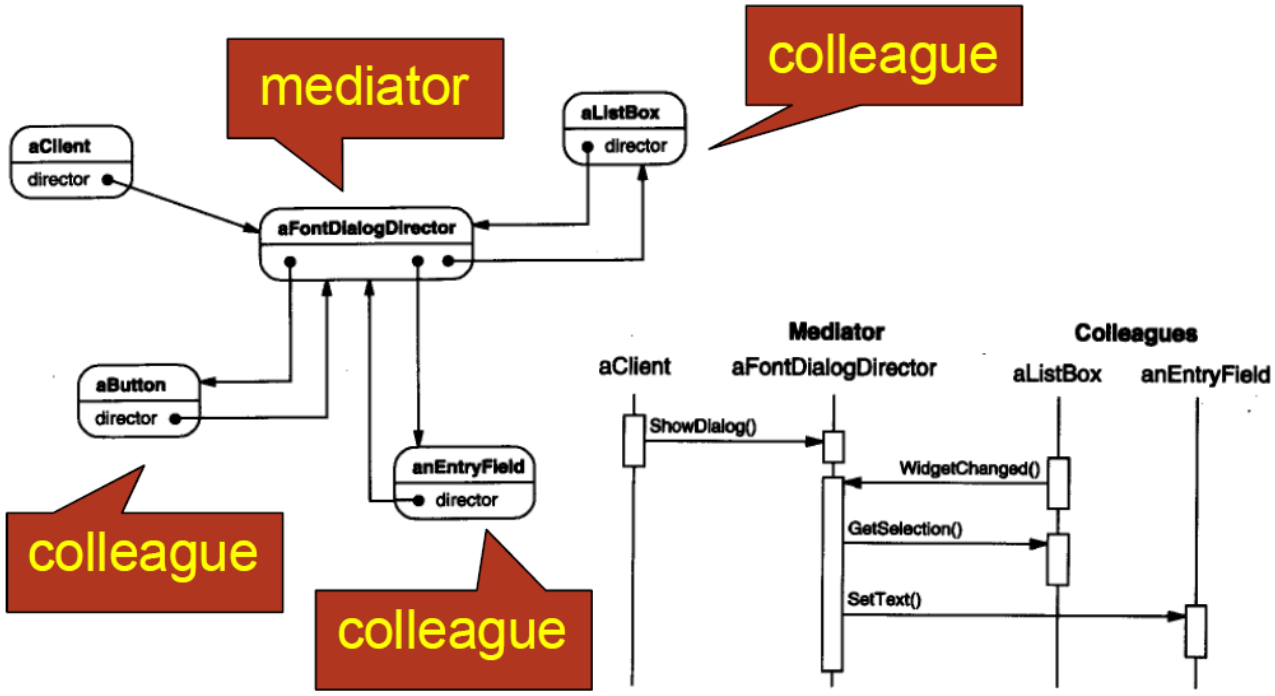


- Mediator（抽象中介者）：中介者定义一个接口用于与各同事(Colleague)对象通信。
- ConcreteMediator（具体中介者）：具体中介者通过协调各同事对象实现协作行为。了解并维护它的各个同事。
- Colleague（同事类）：每一个同事类都知道它的中介者对象，每一个同事对象在需与其他的同事通信的时候，与它的中介者通信。

举例：对话框

- 考虑一个图形用户界面中对话框的实现。对话框使用一个窗口来展现一系列的窗口组件，如按钮、菜单和输入域等。
- 通常对话框中的窗口组件间存在依赖关系，例如：
  - 当一个特定的输入域为空时，某个按钮不能使用；
  - 在列表框的一列选项中选择一个表目可能会改变一个输入域的内容；
  - 在输入域中输入中文可能会自动地选择一个或多个列表框中相应的表目；
  - 一旦正文出现在输入域中，其他一些按钮可能就变得能够使用了，这些按钮允许用户做一些操作，比如改变或删除这些正文所指的东西。
- 不同的对话框会有不同的窗口组件间的依赖关系。

- Solution: 通过将集体行为封装在一个单独的mediator对象
  - 中介者负责控制和协调一组对象间的交互。
  - 中介者充当一个中介以使组中的对象不再相互显式引用。
  - 这些对象仅知道中介者，从而减少了相互连接的数目。



中介者模式的实现要点

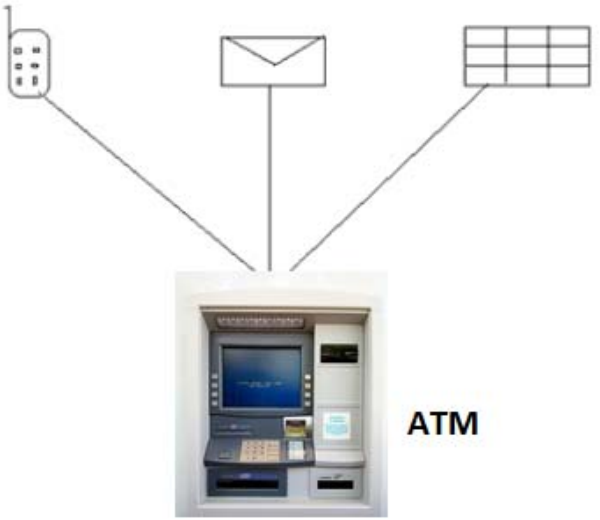
- 当各Colleague仅与一个Mediator一起工作时，没有必要定义一个抽象的Mediator类。
- Colleague—Mediator通信，可以采用Observer模式，或者在Mediator中定义一个特殊的通知接口，各Colleague在通信时直接调用该接口。
- 随着控制逻辑的复杂化，Mediator模式具体对象的实现可能相当复杂。这时候可以对Mediator对象进行分解处理。

适用场景

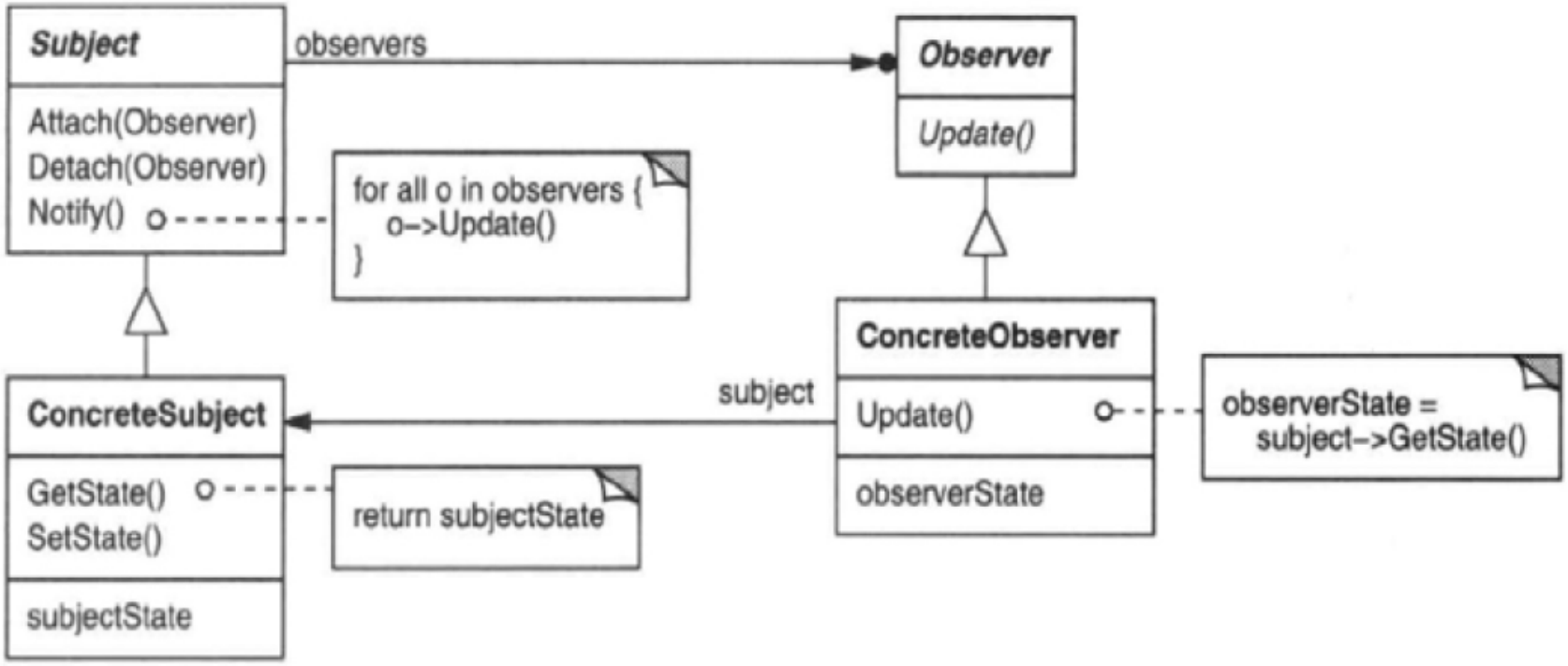
- 一组对象之间的通信方式过于复杂，产生的相互依赖关系结构混乱且难以理解。
- 一个对象引用其他很多对象并且直接与这些对象通信,导致难以复用该对象。
- 想定制一个分布在多个类中的行为，而又不想生成太多的子类。

20）观察者模式Observer

- 在软件构建过程中，我们需要为某些对象建立一种“通知依赖关系”---一个对象(目标对象)的状态发生改变，所有的依赖对象(观察者对象)都将得到通知。如果这样的依赖关系过于紧密，将使软件不能很好地抵御变化。
- 举例：在ATM取款，当取款成功后，以手机、邮件等方式进行通知。
- 观察者模式定义了对象间一对多的关系，当一个对象的状态变化时，所有依赖它的对象都得到通知并且自动地更新。



观察者模式的结构

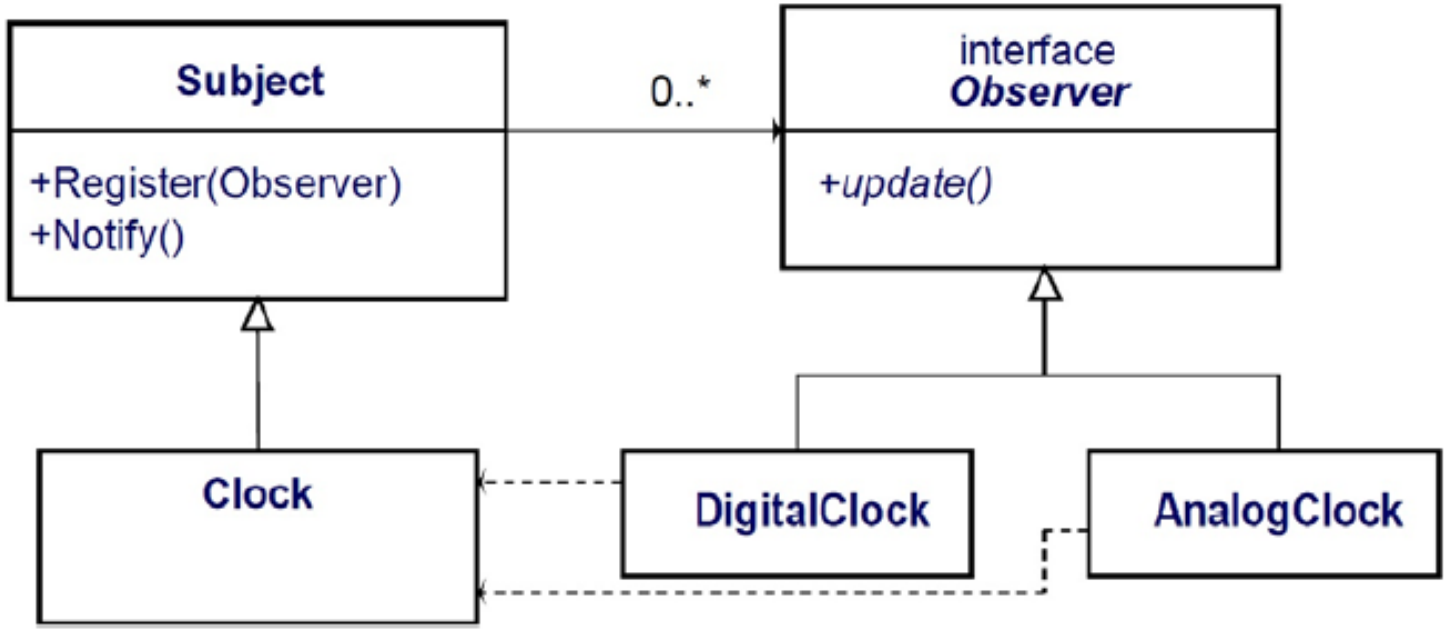


- 一个目标可以有任意数目的依赖它的观察者。
- 一旦目标的状态发生改变, 所有的观察者都得到通知。作为对这个通知的响应，每个观察者都将查询目标以使其状态与目标的状态同步。
- 这种交互也称为发布 - 订阅(publish-subscribe) 。目标是通知的发布者，观察者是通知的订阅者。

- Subject(目标)
  - 目标知道它的观察者。可以有任意多个观察者观察同一个目标。
  - 提供注册和删除观察者对象的接口。
- Observer(观察者)
  - 为那些在目标发生改变时需获得通知的对象定义一个更新接口。
- ConcreteSubject(具体目标)
  - 将有关状态存入各ConcreteObserver对象。
  - 当它的状态发生改变时，向它的各个观察者发出通知。
- ConcreteObserver(具体观察者)
  - 维护一个指向ConcreteSubject对象的引用。
  - 存储有关状态，这些状态应与目标的状态保持一致。
  - 实现Observer的更新接口以使自身状态与目标的状态保持一致。

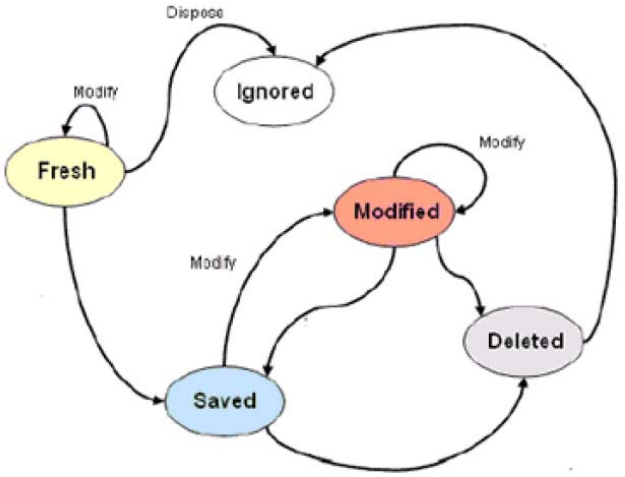
举例：时钟

- 系统内有一个计时器（模型，没有界面），这个计时器驱动两个形式的时钟界面，一个数字时钟，一个模拟时钟

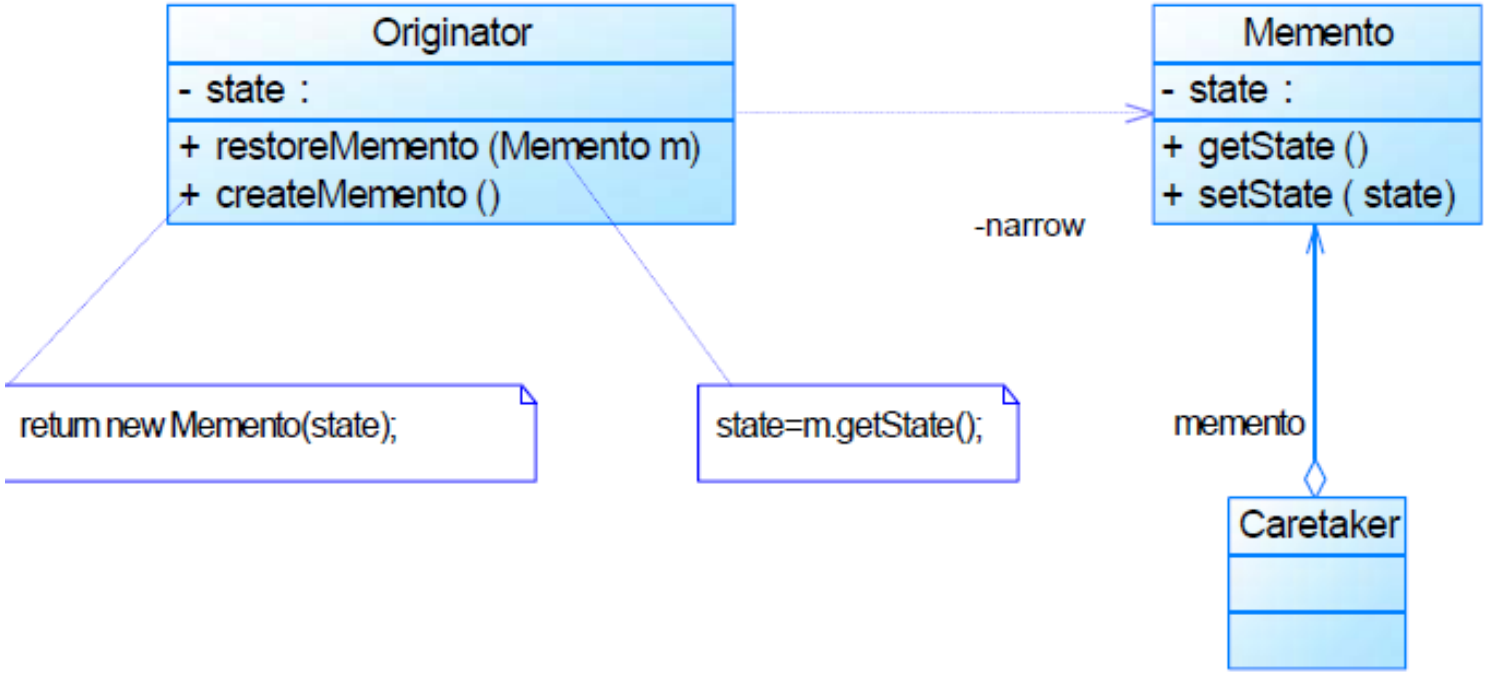


21）备忘录模式Memento

- 对象状态的变化无端，如何回溯恢复对象在某个点的状态?
  - 例如: 用户取消不确定的操作;
  - 当数据库某事务中一条数据操作语句执行失败时，整个事务将进行回滚操作，系统回到事务执行之前的状态。
- 如果使用一些公用接口来让其他对象得到对象的状态，便会暴露对象的细节实现。
- 备忘录模式在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。



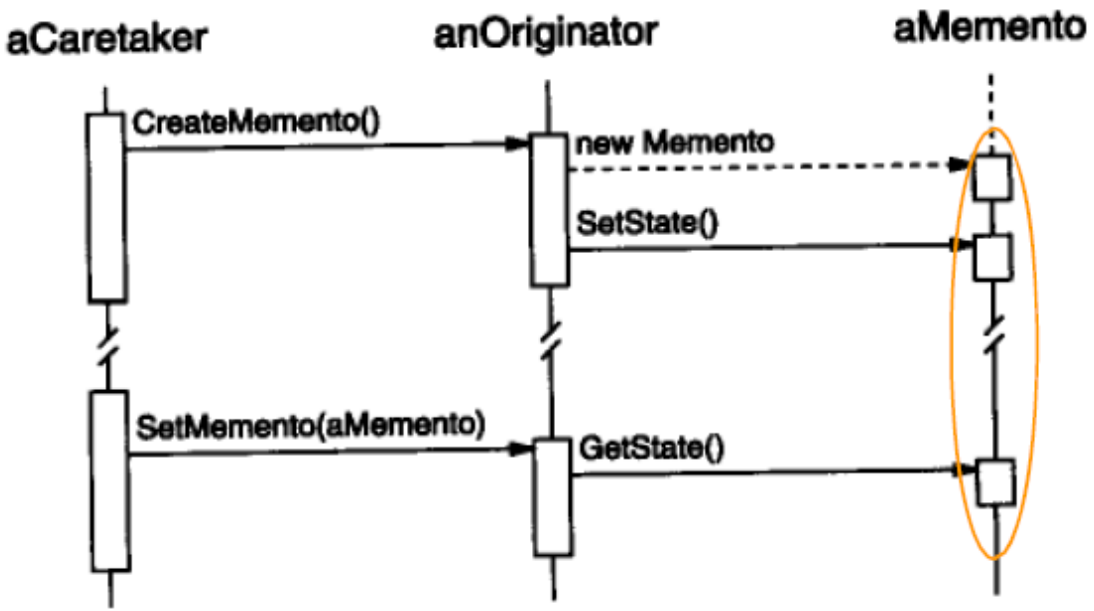
备忘录模式的结构



- 一个备忘录(memento)是一个对象, 它存储另一个对象在某个瞬间的内部状态，而后者称为备忘录的原发器(originator)。当需要设置原发器的检查点(checkpoint) 时, 取消操作机制会向原发器请求一个备忘录。原发器用描述当前状态的信息初始化该备忘录。只有原发器可以向备忘录中存取信息，备忘录对其他的对象“不可见”。

- Memento (备忘录)
  - 备忘录存储原发器对象的内部状态。原发器根据需要决定备忘录存储原发器的哪些内部状态。
  - 防止原发器以外的其他对象访问备忘录。备忘录实际上有两个接口，管理者 (caretaker) 只能看到备忘录的窄接口—它只能将备忘录传递给其他对象。相反，原发器能够看到一个宽接口，允许它访问返回到先前状态所需的所有数据。理想的情况是只允许生成本备忘录的那个原发器访问本备忘录的内部状态。
- Originator (原发器)
  - 原发器创建一个备忘录，用以记录当前时刻它的内部状态。
  - 使用备忘录恢复内部状态。
- Caretaker (管理器)
  - 负责保存好备忘录
  - 不能对备忘录的内容进行操作或检查

- 管理器向原发器请求一个备忘录，保留一段时间后，将其送回给原发器。

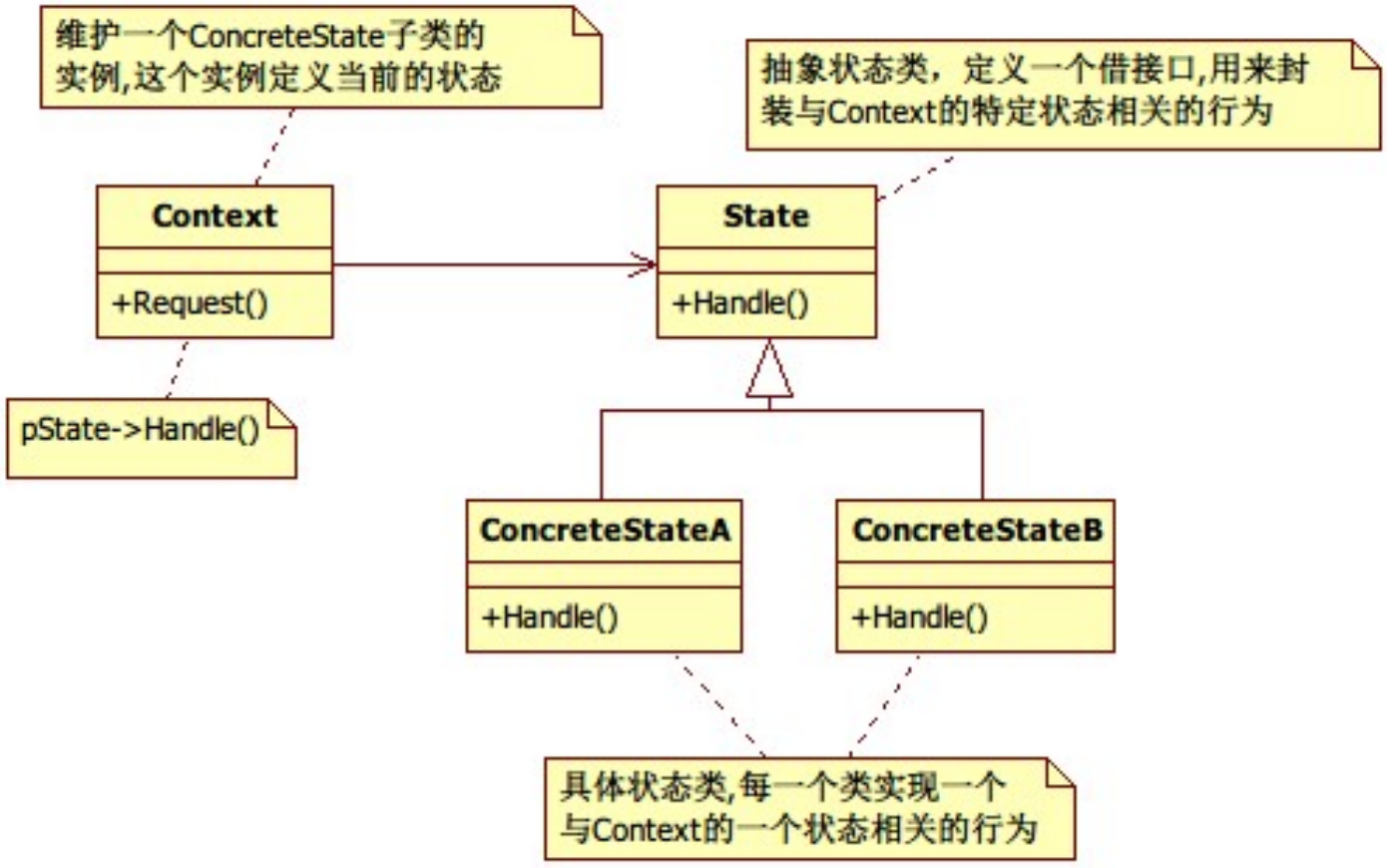


- 备忘录是被动的，只有创建备忘录的原发器会对它的状态进行赋值和检索。

22）状态模式State

- 在软件构建过程中，某些对象的状态如果改变，其行为也随之发生改变，如文档处于只读状态，其支持的行为和读写状态支持的行为就可能完全不同。
- 如何在运行时根据对象的状态来透明地改变对象的行为？但又不会为对象操作和状态转化之间引入紧耦合？
- 状态模式允许一个对象在其内部状态改变时改变它的行为。

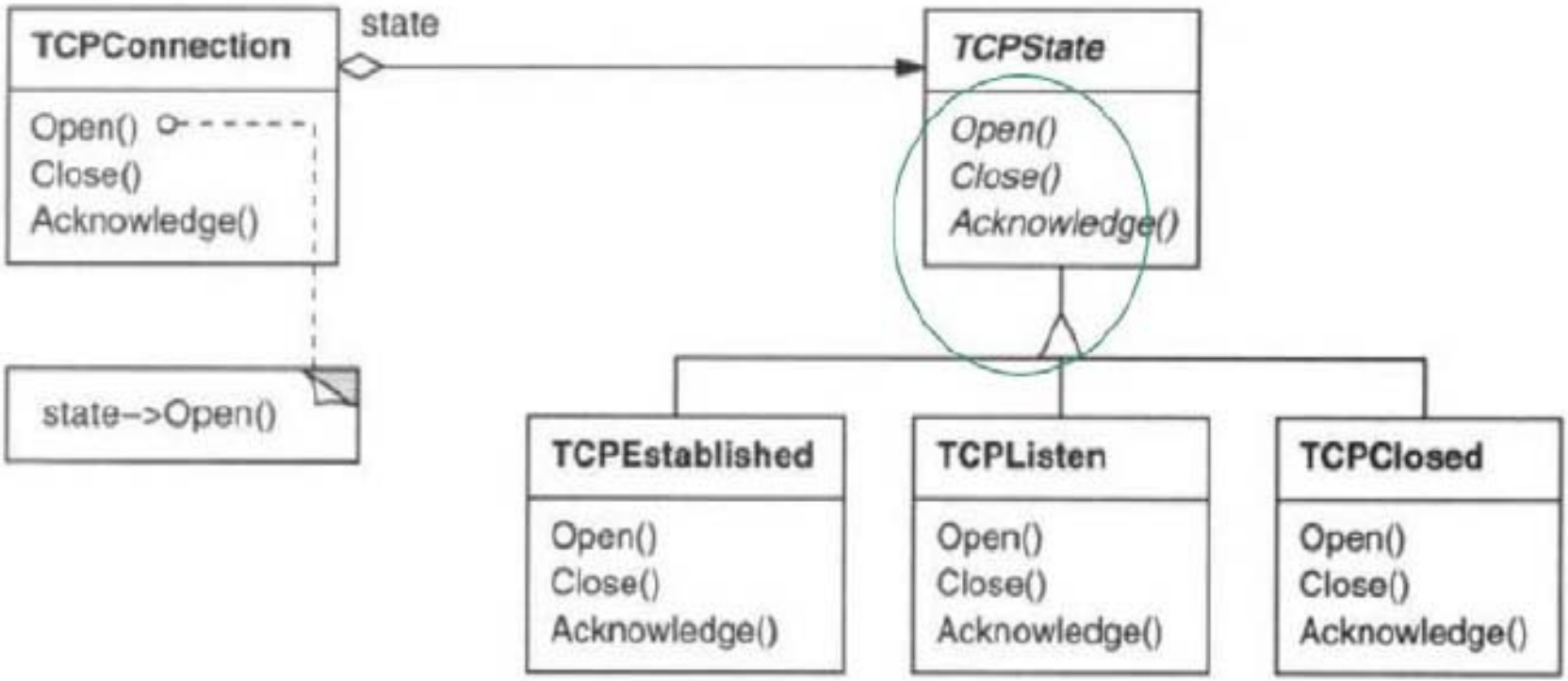
状态模式的结构


维护一个ConcreteState子类的
实例,这个实例定义当前的状态
抽象状态类，定义一个借接口,用来封
装与Context的特定状态相关的行为
Context
+Request()
State
+Handle()
pState->Handle()
ConcreteStateA
+Handle()
ConcreteStateB
+Handle()
具体状态类,每一个类实现一个
与Context的一个状态相关的行为

- 在状态模式结构中需要理解环境类与抽象状态类的作用：
  - 环境类实际上就是拥有状态的对象，环境类有时候可以充当状态管理器(State Manager)的角色，可以在环境类中对状态进行切换操作。
  - 抽象状态类可以是抽象类，也可以是接口，不同状态类就是继承这个父类的不同子类，状态类的产生是由于环境类存在多个状态，同时还满足两个条件：这些状态经常需要切换，在不同的状态下对象的行为不同。因此可以将不同对象下的行为单独提取出来封装在具体的状态类中，使得环境类对象在其内部状态改变时可以改变它的行为，对象看起来似乎修改了它的类，而实际上是由于切换到不同的具体状态类实现的。由于环境类可以设置为任一具体状态类，因此它针对抽象状态类进行编程，在程序运行时可以将任一具体状态类的对象设置到环境类中，从而使得环境类可以改变内部状态，并且改变行为。

举例：TCP

- 考虑一个表示网络连接的类TCPconnection。一个TCPconnection对象的状态处于若干个不同的状态之一：
  - 连接已建立
  - 正在监听
  - 连接已关闭
- 当一个TCPconnection对象收到其他对象的请求时，它根据自身的当前状态作出不同的反应。如：一个Open请求的结果依赖于该连接是处于连接已关闭状态还是连接已建立状态。State模式描述了TCPconnection如何在每一种状态下表现出不同的行为。



- TcpConnection 类持有一个状态对象，亦即TcpState 的一个子类的实例，来表征TCP连接的现在状态。TcpConnection类把所有与状态有关的请求都委派给它的状态对象。TcpConnection 使用它的TcpState 的子类实例来执行特定的连接状态所对应的操作。

实现要点

- State模式将所有与一个特定状态相关的行为都放入一个State的子类对象中，在对象状态切换时，切换相应的对象；但同时维持State的接口，这样实现了具体操作与状态转换之间的解耦。
- 为不同的状态引入不同的对象使得状态转换变得更加明确，而且可以保证不会出现状态不一致的情况，因为转换是原子性的----即要么彻底转换过来，要么不转换。
- 如果State对象没有实例变量，那么各个上下文可以共享同一个State对象，从而节省对象开销。

Strategy模式与State模式

- 策略用来处理算法方式变化，而状态则是处理状态变化。
- 对于模式的选择反映出设计者对结构的理解。
  - 此刻你把它视为一种状态，如果将来你发觉用Strategy能更好的说明你的意图，你可以重构它。
  - 这两种模式在结构上是相似，都是通过继承来实现的。所以如果重构变化会较小。
  - 这两种模式之间有时候没有区别，譬如在面向连结的TCPConnection例子中，不同的状态可能具有不同的方法。但是一个无连结的P2P底层协议中，所有状态需要处理的就是一个processXML方法，这时候，你可以叫它状态或者策略都可以。
  - Strategy模式用来处理用户算法方案的变化。这些算法方案之间一般来说没有状态变迁，并且总是从几个算法中间选取一个。

23）访问者模式Visitor

- 在软件构建过程中，由于需求的改变，某些类层次结构中常常需要增加新的行为(方法)，如果直接在基类中做这样的更改，将会给子类带来很繁重的变更负担，甚至破坏原有设计。

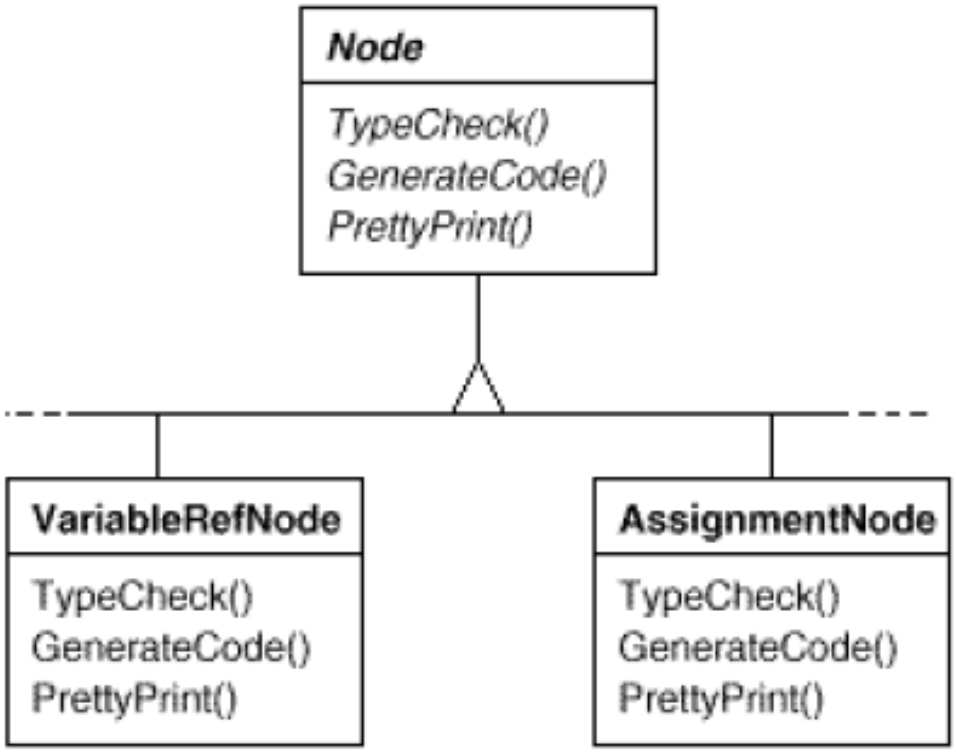
- 如何在不改变类层次结构的前提下，在运行时根据需要透明地为类层次结构上的各个类动态添加新的操作，从而避免上述问题？

- 访问者模式把这样的操作包装到一个独立的对象(visitor)中，然后在遍历过程中把此对象传递给被访问的元素。

示例：编译器

- 考虑一个编译器，它将源程序表示为一个抽象语法树。该编译器需在抽象语法树上实施某些操作以进行“静态语义”分析，例如检查是否所有的变量都已经被定义了，它也需要生成代码，因此它可能要定义许多操作以进行类型检查、代码优化、流程分析，检查变量是否在使用前被赋初值，等等。此外，还可使用抽象语法树进行优美格式打印、程序重构以及对程序进行多种度量。
- 这些操作大多要求对不同的结点进行不同的处理。如对代表赋值语句的结点的处理就不同于对代表变量或算术表达式的结点的处理。因此有用于赋值语句的类，有用于变量访问的类，还有用于算术表达式的类，等等。

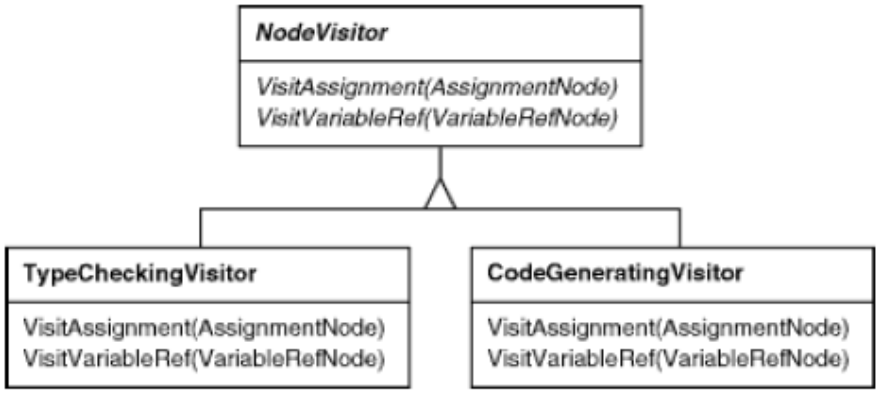
不用visitor的compiler例子

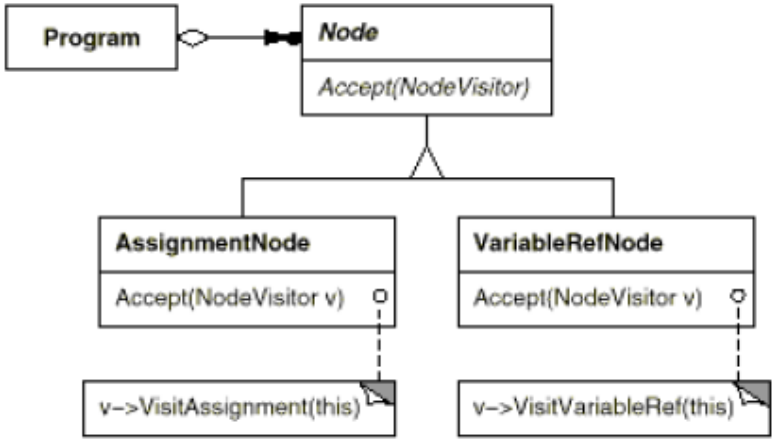


- 将所有这些操作分散到各种结点类中会导致整个系统难以理解、维护和修改。
- 将类型检查代码和优美格式打印代码或流程分析代码放在一起，会产生混乱。
- 增加新的操作将重新编译所有的类。

如何才能独立增加新的操作，并使这些结点类独立于作用于其上的操作?

使用visitor的compiler例子

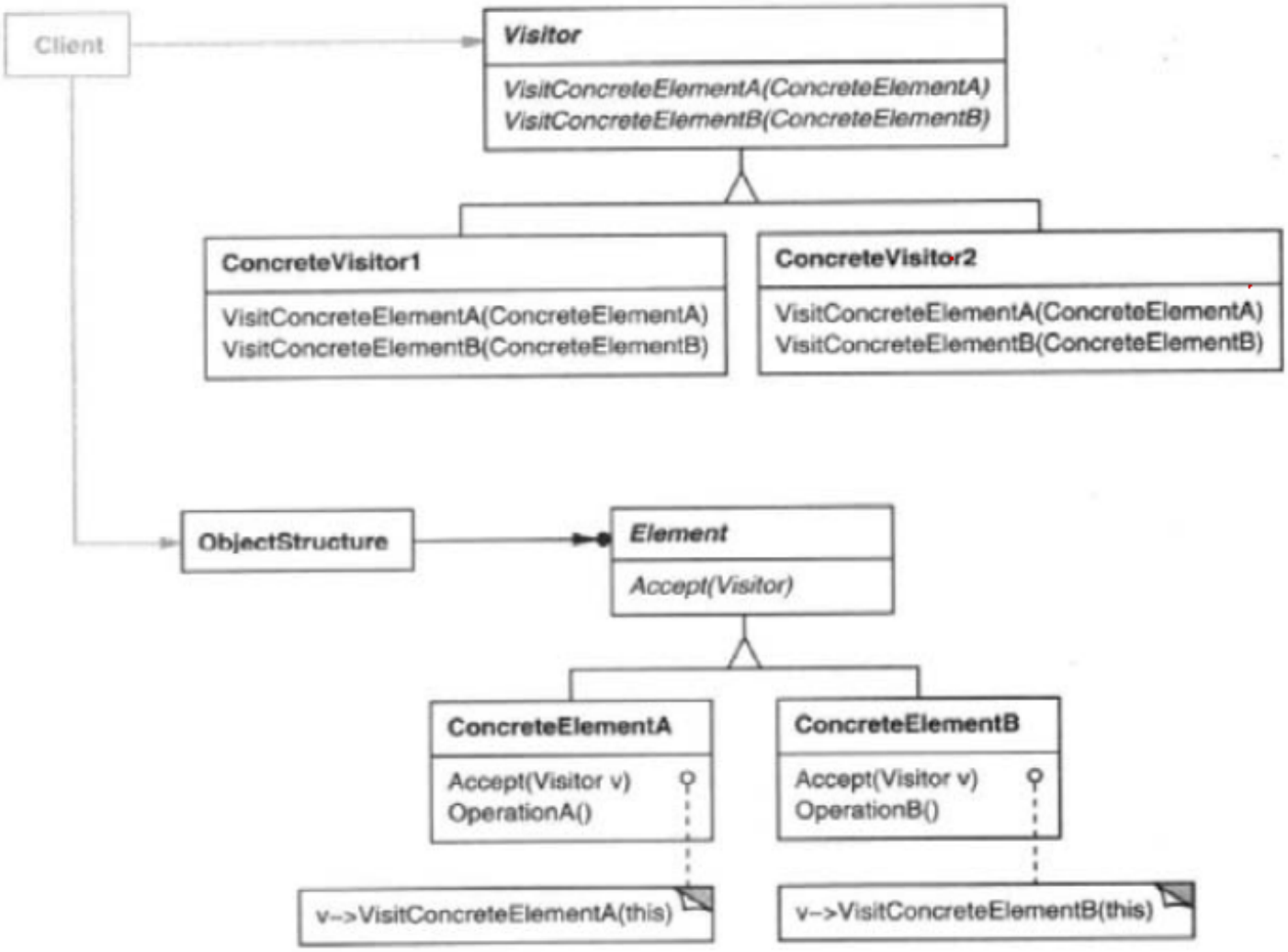




两个类层次！

- 使用Visitor模式，必须定义两个类层次：
  - 一个对应于接受操作的元素(node层次)
  - 一个对应于定义对元素的操作的访问者(node visitor层次)
- 给访问者类层次增加一个新的子类即可创建一个新的操作。只要该编译器接受的语法不改变(即不需要增加新的node子类)，可以简单定义新的node visitor子类以增加新的功能。

访问者模式的结构


Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

- Visitor(抽象访问者)
  - 为该对象结构中ConcreteElement的每一个类声明一个Visit操作。该操作的名字和特征标识了发送Visit请求给该访问者的那个类。这使得访问者可确定正被访问元素的具体的类。这样访问者就可以通过该元素的特定接口直接访问它。
- ConcreteVisitor(具体访问者)
  - 实现每个由Visitor声明的操作。每个操作实现本算法的一部分，而该算法片断乃是对应于结构中对象的类。ConcreteVisitor为该算法提供了上下文并存储它的局部状态。这一状态常常在遍历该结构的过程中累积结果。
- Element(抽象元素)
  - 定义一个Accept操作，它以一个访问者为参数。
- ConcreteElement(具体元素)
  - 实现Accept操作，该操作以一个访问者为参数。
- ObjectStructure(对象结构)
  - 能枚举它的元素。
  - 可以提供一个高层的接口以允许该访问者访问它的元素。
  - 可以是一个复合或是一个集合。

Summary: Behavioral Patterns

- Strategy、Iterator、Mediator、State、Command
  - 用一个对象来封装某些经常变化的特性，比如算法(Strategy)、交互协议(Mediator)、状态(State)、行为、遍历方法(Iterator)
- Mediator、Observer
  - Observer建立起subject和observer之间的松耦合连接
  - Mediator把约束限制集中起来，进行中心控制
- Command、Chain of Responsibility、Interpreter
  - Command模式侧重于命令的总体管理
  - Chain of Responsibility侧重于命令被正确处理
  - Interpreter用于复合结构中操作的执行过程

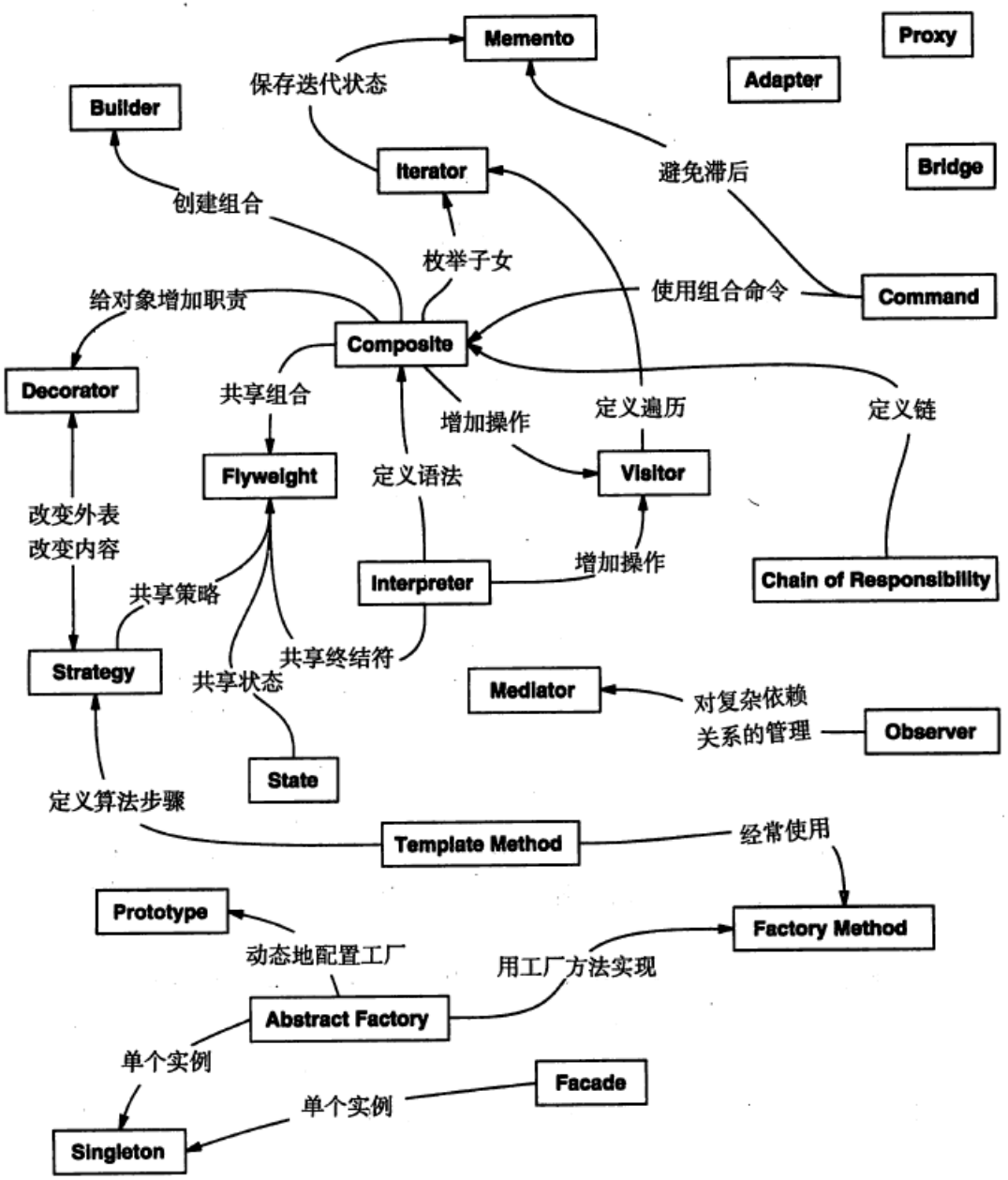
Summary: Behavioral Patterns

- Interpreter模式注重封装特定领域变化，支持领域问题的频繁变化
- Template Method模式注重封装算法结构，支持算法子步骤变化
- Chain of Responsibility模式注重封装对象责任，支持责任的变化
- Command模式注重将请求封装为对象，支持请求的变化
- Iterator模式注重封装集合对象内部结构，支持集合的变化
- Strategy模式注重封装算法，支持算法的变化
- Mediator模式注重封装对象间的交互，支持对象交互的变化
- Memento模式注重封装对象状态变化，支持状态保存/恢复
- Observer模式注重封装对象通知，支持通信对象的变化
- State模式注重封装与状态行为相关的行为，支持状态的变化
- Visitor模式注重封装对象操作变化，支持在运行时为类层次结构动态添加新的操作。

设计模式是“封装变化”方法的最佳阐释

- 无论是创建型模式、结构型模式还是行为型模式，归根结底都是寻找软件中可能存在的“变化”，然后利用抽象的方式对这些变化进行封装。
- 由于抽象没有具体的实现，就代表了一种无限的可能性，使得其扩展成为了可能。
- 利用设计模式，通过封装变化的方法，可以最大限度的保证软件的可扩展性。
- 在设计之初预见某些变化，在一定程度上避免未来存在的变化为软件架构带来的灾难性伤害，就能用很小的代价来应对剧烈的变化。

设计模式间的关系


Memento
Proxy
保存迭代状态
Adapter
Builder
Iterator
避免滞后
Bridge
创建组合
枚举子女
给对象增加职责
使用组合命令
Command
Composite
Decorator
共享组合
增加操作
定义遍历
定义链
Flyweight
定义语法
Visitor
改变外表
改变内容
共享策略
Interpreter
增加操作
Chain of Responsibility
Strategy
共享状态
共享终结符
Mediator
对复杂依赖
关系的管理
Observer
State
定义算法步骤
Template Method
经常使用
Prototype
Factory Method
动态地配置工厂
用工厂方法实现
Abstract Factory
单个实例
Facade
单个实例
Singleton