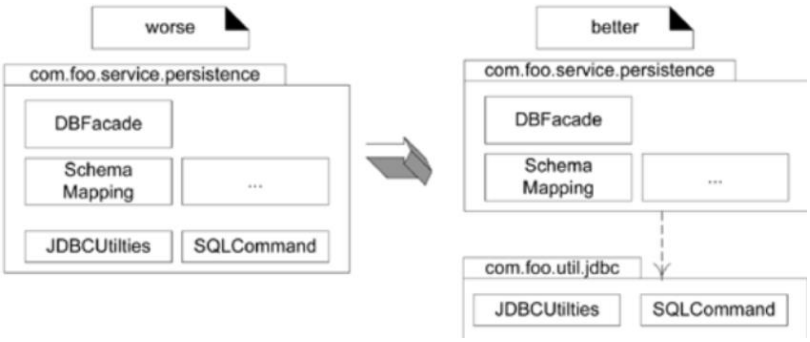
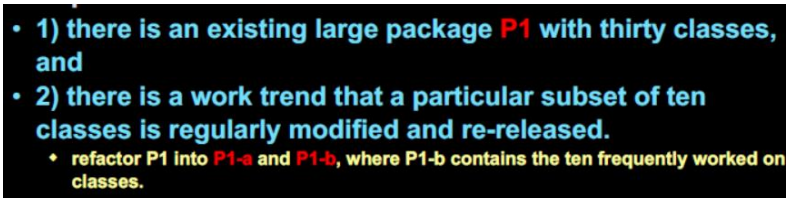
- **包设计原则**

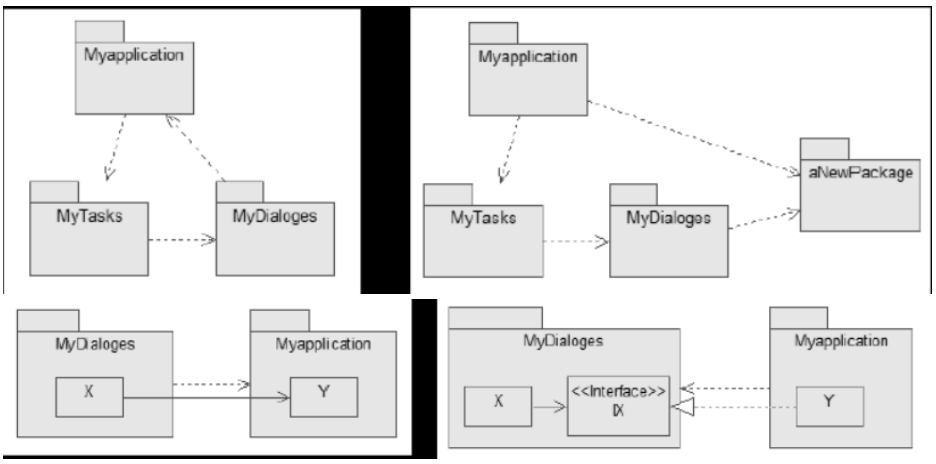
- 有关包的粒度 granularity，包的内聚性原则 cohesion，明确哪些类应该放置到同一个包
  - 重用-发布等价原则：重用粒度等价于发布粒度，这种粒度就是包
    - 一个包中的所有元素，即类和接口，会被当做单一的可重用单元看待
    - 多层应用水平分割：spring，EJB
  - 共同重用原则：同一个包中的所有类应该被一起重用，即只要重用了包中的任意一个类，就应该重用剩余的所有类
    - 类似于类设计的“接口隔离原则”，不要把用户并不会用到的类和需要用到的类放在同一个包里
    - The class in a package are reused together; If you reuse one of the classes in a package，you reuse them all



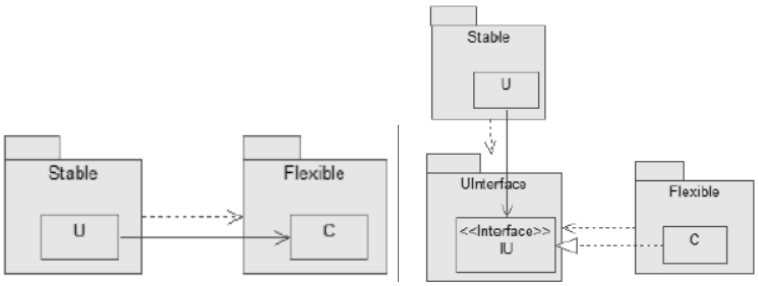
  - 共同封闭原则：软件的变更影响某个包的时候，会影响到这个包中的所有类
    - Classes in a package should be closed together against same kind of changes
    - Reduce widespread 普遍的 dependency on unstable packages



- 有关包的稳定性 stability，包的耦合性原则 coupling，明确包和包之间的依赖关系如何设计
  - 无环依赖原则：包之间的结构依赖是无环的
    - Solutions：将参与循环的类型分解成更小的包
    - Break the cycle with an interface



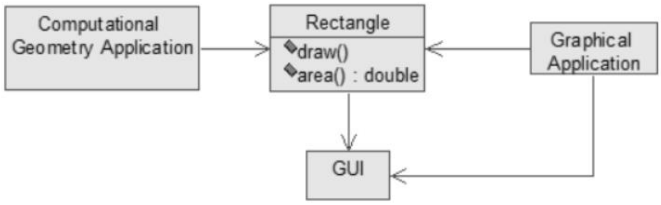
- 稳定依赖原则：每个包都只依赖比他更稳定的包
  - 稳定：不需要变更的包，对各种变更不做调整可以自适应

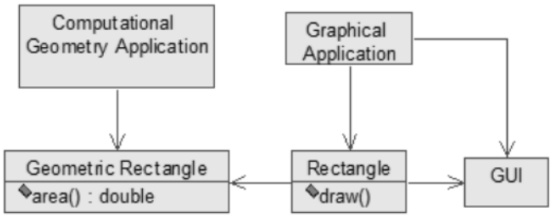


- 稳定抽象原则：最稳定的包应该是最抽象的包
  - SAP，SDP combined amount to the DIP for package

- 类设计原则

- 单一职责原则：每个类都只有一个职责
  - We define responsibility to be “a reason for change”



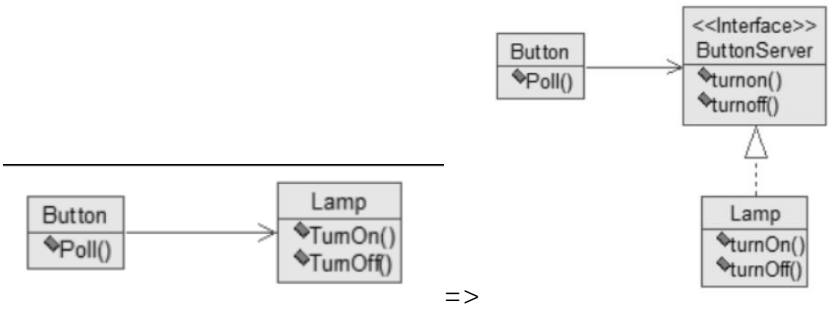


=>

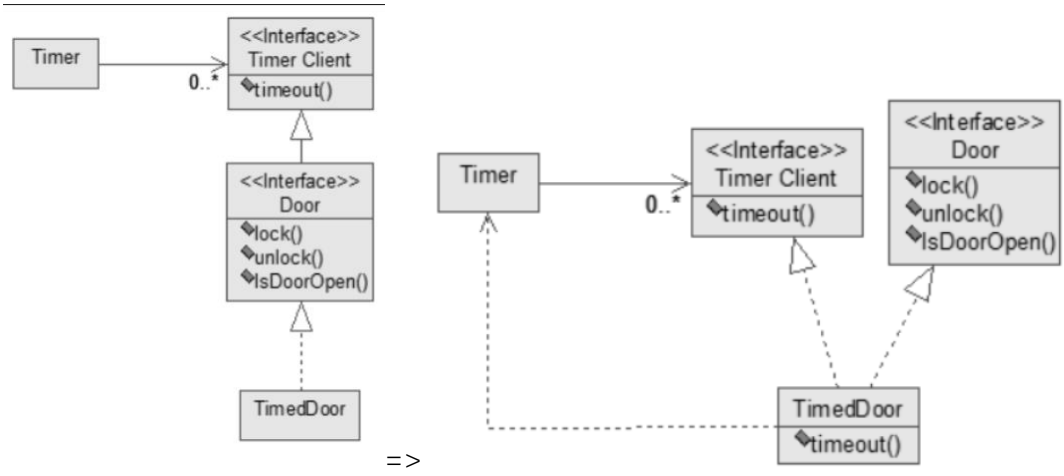
- 里氏替换原则：子类能够替换父类，意思不变；子类必须有父类方法的所有实现
  - Subtype must be substitutable for their base types
  - The objects in the same inherit architecture should have behavior feature

In real world , penguin and bird is a “is-a” relationship. If we design penguin is a subclass of bird class ,we violate LSP principle .because bird can fly( ) while penguin can not fly. So

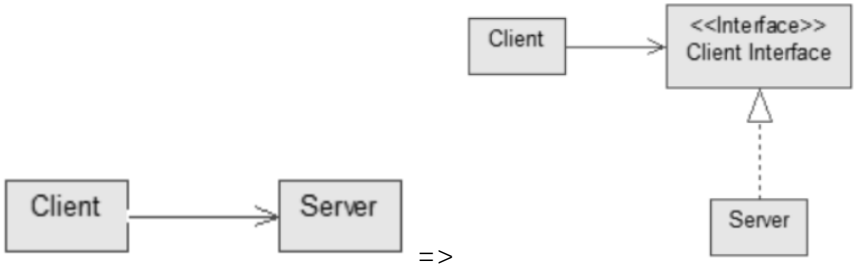
- 依赖倒置原则：类和类的依赖是通过接口来实现，接口的粒度应该如何控制没有说明； 高层模块不应该依赖于低层模块，他们都依赖于抽象；抽象不应该依赖于细节，细节必须依赖于抽象
  - 契约式设计：依赖接口的高层抽象
  - High-level modules should not depend on low-level modules(依赖abstractions)
  - Abstraction shouldn’t depend on details，details should depend on abstraction



- 接口隔离原则
  - 客户端不应该被迫依赖他们并不需要使用的接口，类与类的依赖关系应该建立在小的接口上
  - Clients should not forced to depend on methods that they don't use



- 开放-关闭原则
  - 软件实体，即类、模块、函数等，应该对扩展开放，对修改关闭

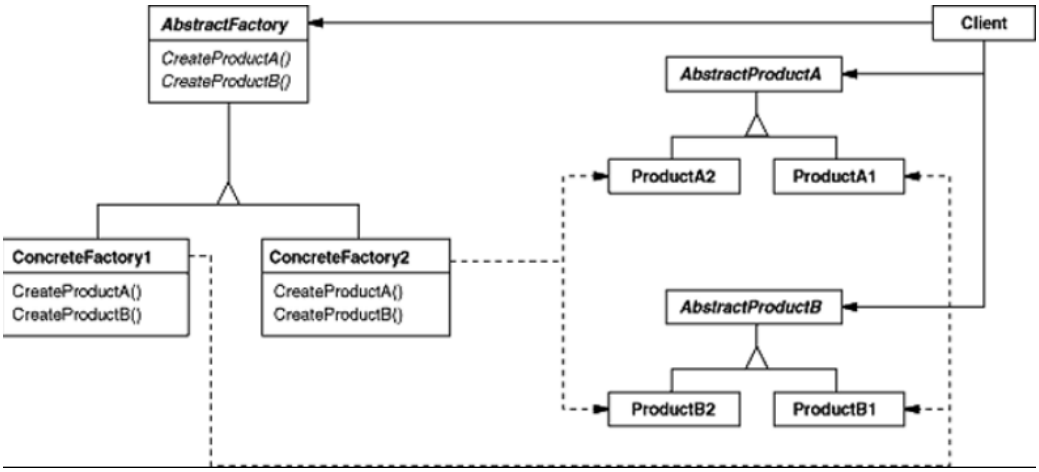


创建型设计模式

抽象工厂

抽象工厂模式提供了一个接口，用于创建一组相关或互相依赖的对象，同时并不要求必须指定它们的具体类。当系统中存在多组相关或互相依赖的对象，每一组对象都会被一同使用时，就可以使用该模式。

示例：AbstractFactory, User, GoldUser, SilverUser, Bid, GoldBid, SilverBid



使用情景：1. 系统的对象 product 的创建、构成 compose、代理 represent 要独立的时候

2. 系统需要被配置为 configure 多个不同类型的 product
3. 一组相关的对象需要设计成一起使用，需要强制这种约束
4. 希望提供 class library of product，只暴露出接口，不暴露实现

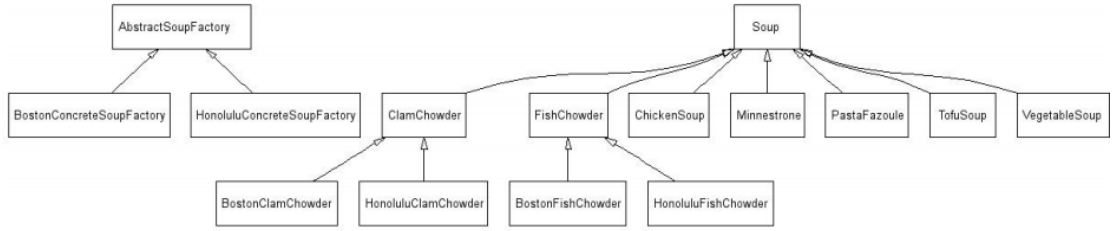
关键结论：1. Isolates 隔离 concrete classes

2. makes exchanging 交换 product family easy
3. promote 提升 consistency 一致性 among product
4. supporting new kinds of product is difficult

实现抽象工厂：1. Factories as singletons

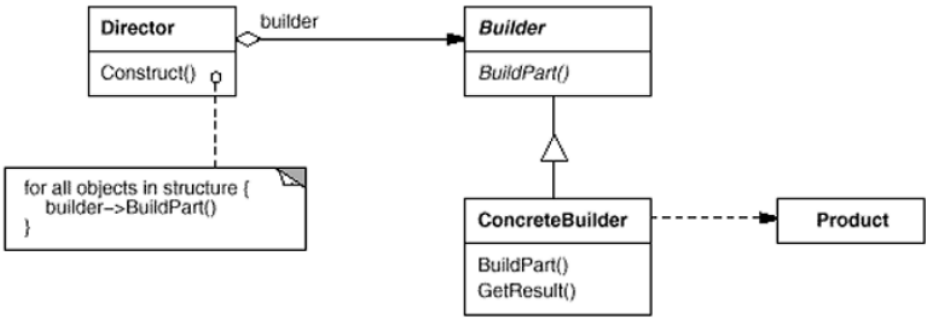
2. creating the products
3. defining extensible 可扩展的 factories

示例：



构建器

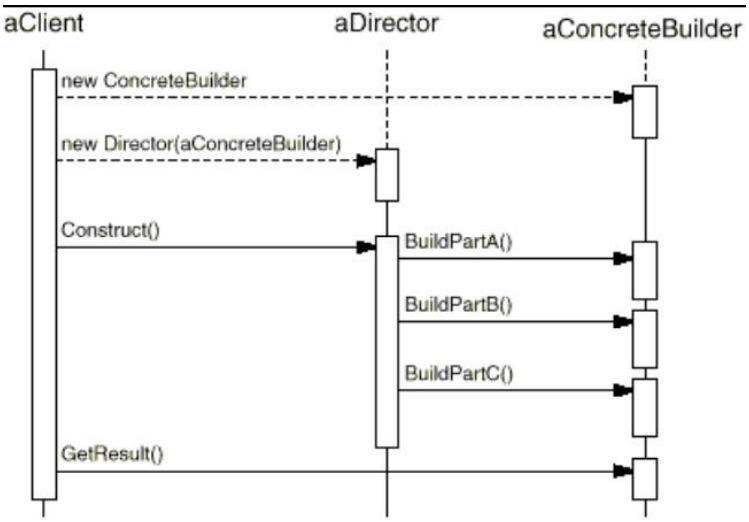
构建器将复杂对象的构建与其表示分离，以使得相同的构建过程可以创建不同的对象表示。当创建复杂对象的方式应该独立于创建其各个组成部分的方式和各部分的组装方式时，可以使用该设计模式。



使用情景：1. 创建复杂对象的算法需要独立于对象的组成 make up 和组装 assembled

2. 构建器必须对构建的对象遵循复杂的表示 representation

合作 collaboration：

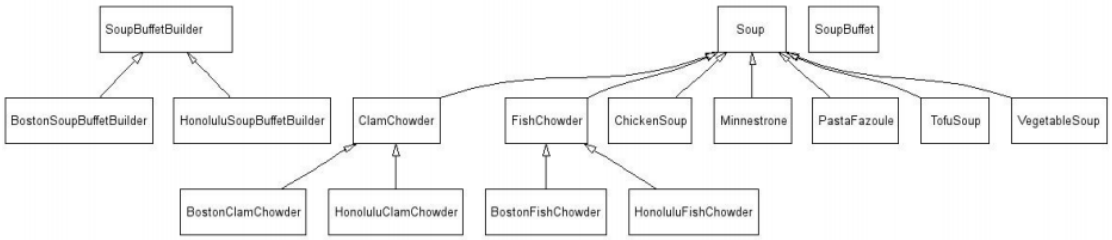


关键结论：1. Lets you vary 变化 a product's internal representation 内在表现

2. isolates code for construction and representation 构建和表现

3. gives you finer control over the construction process

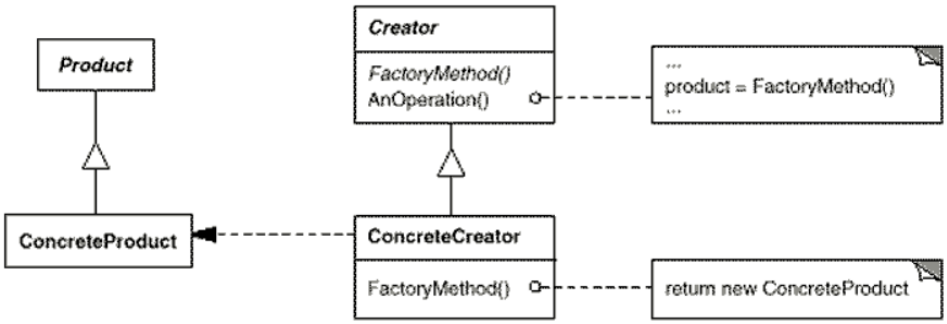
示例：SoupBuffet 自助餐 SoupBuffetBuilder,



工厂方法

工厂方法定义了创建对象的接口，但是需由其子类来确定需要实例化的具体类。当一个类无法预知其要创建的对象所属的具体类，从而将指定所需创建对象的具体类的职责交给子类，并由子类来具体实现创建该对象的逻辑时，就可以用该模式。

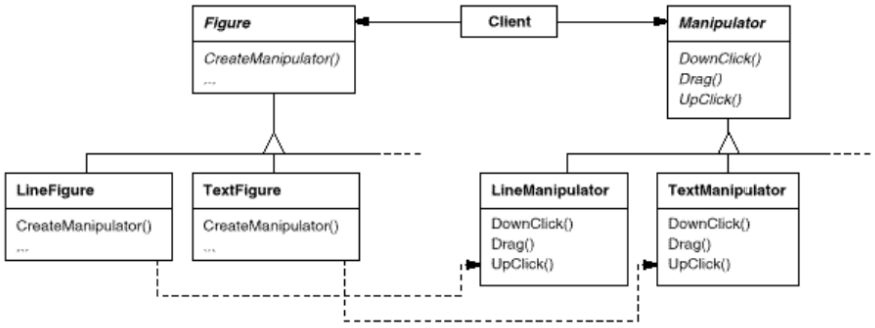
工厂方法和抽象工厂和联系十分紧密的两种模式，抽象方法关注的是创建一组关联对象，而工厂方法关注的是创建一个对象的具体过程。



使用情景：1. Class can't anticipate 预见 the class of objects it must create

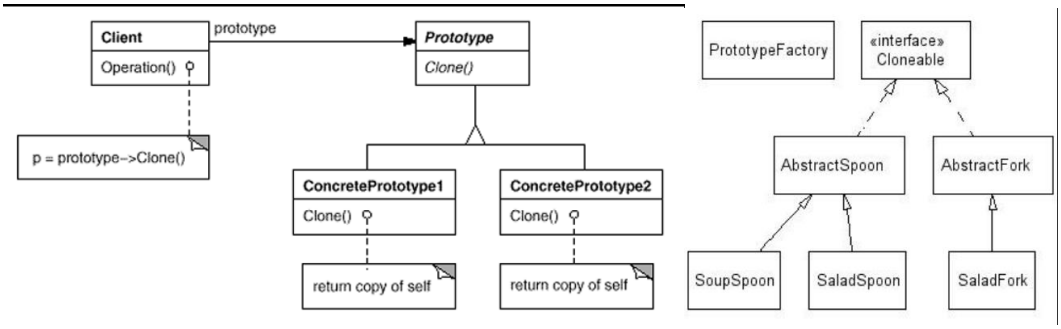
2. class wants its subclass to specify 明确指出 the objects it creates

3. class delegate 委派 responsibility to one of several helper subclass



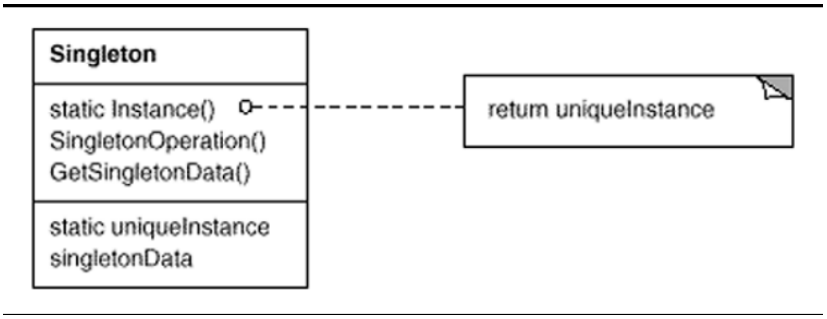
原型

在原型模式中，可以用原型实例来指定所要创建对象的类型，并通过复制原型所创建新的对象。觉不希望因为创建工厂类而导致在代码中产生了一个新的层次，或者对于一个类来说，其所有对象只有极少数的不同状态组合时，就可以采用原型模式。



单例

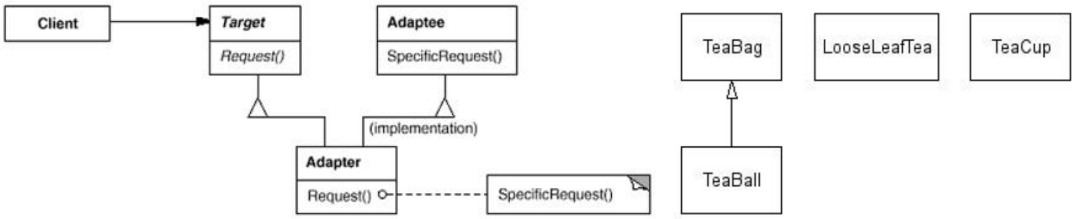
单例模式用于确保一个类只有一个实例，并提供对它的全局访问点。对于本身重量级的对象来说，应用单例模式对于节约计算资源就有重要的意义。



结构型设计模式

适配器 Adapter

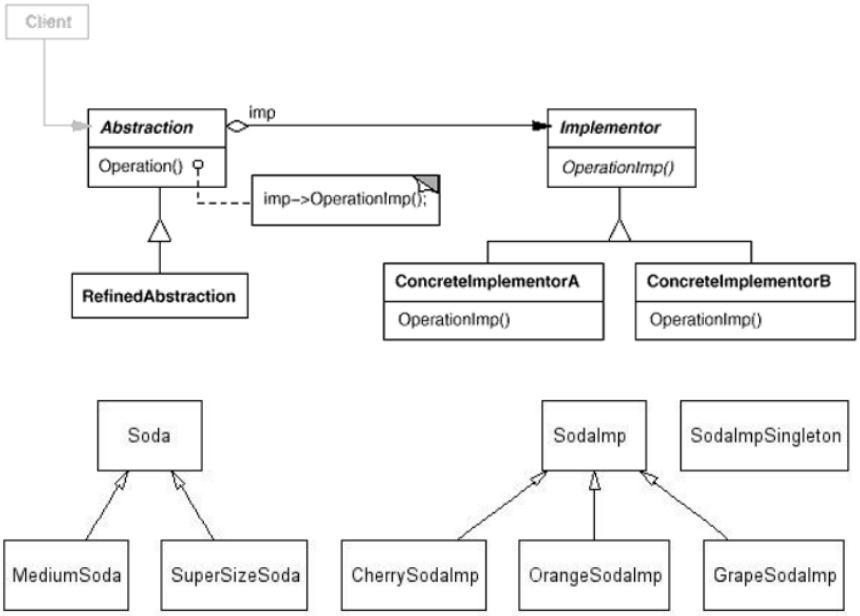
适配器模式用于将一个类的接口转换成客户期望的接口，使得不兼容的接口可以协作。当现有类的接口与期望的接口不匹配，或者希望设计的可复用类能够与当前无法预见到的类进行协作时，就可以使用适配器模式。而对象类型的适配器还可以将子类的接口适配成其父类的接口。



桥接 Bridge

桥接模式可以将类中的抽象部分与实现部分分离，并在运行时刻将他们连接起来，从而使两部分可以独立的变化。如果希望将一个类的抽象部分与实现部分解耦，而不是在编译时刻绑定，或者希望一个类的抽象部分和实现部分可以各自独立地通过子类化进行扩展，那么就可以使用桥接模式。

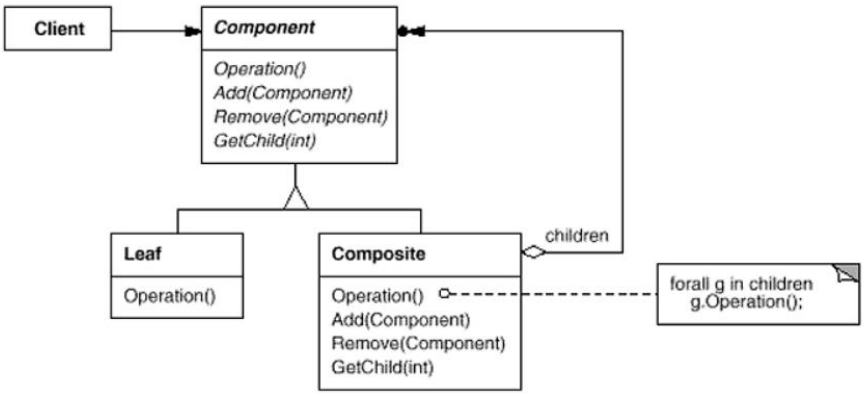
示例：User，Contact



组合 Composite

组合模式将对象组织成树状结构，以表示“部分和整体”的层次结构，使得用户可以以统一的方式处理单个对象和对象组合。当希望表示对象“部分与整体”层次结构，并希望用户能够忽略单个对象与对象组合的差异时，就可以使用该模式。

示例：用户对汽车的评价



装饰器

装饰器模式可以动态地将额外的职责添加到对象中，它提供了一种替代方案，可替代子类化实现对象功能 。 子类化相比，装饰器可以动态而透明地向单个对象添加额外的功能，而不会影响到其他对象，同时这些职责也可以动态地撤销。当子类化不符合要求时，就可以考虑使用该模式。

示例：一年中汽车附送的小礼品

外观 facade

外观模式为子系统中的一组接口提供了更高层的统一接口，使得该子系统更容易使用。当希望向用户提供一个简单的接口用于访问复杂子系统，或者希望对子系统分层时，就可以使用该模式。

一次性接受多参数（用户名，密码，竞拍物品，竞拍价格），在外观类中依次调用。

享元

享元模式可以通过共享有效地支持数量庞大的细粒度对象。有些应用中存在这样的对象，存储他们的高昂开销仅仅是因为数量庞大而产生的。这些对象的状态是由外部程序赋予的，并且其状态的不同取值数量有限，此时如果应用不区分对象的标识，而只是区分对象的状态，那么就可以用少数共享对象来替代大量的对象。

Token 对象 -> 即使设计人员将并发访问用户数量的上限设置为数百或上千，也只需要创建两个 token 对象

代理

代理模式提供了一个代理者，用于控制对被代理对象的访问。使用代理存在多种原因，报告提供对远程对象的本地表示，用于随需创建开销高昂的对象，对被代理对象进行保护，以及执行额外的行为等。

行为型设计模式

职责链

职责链模式可以避免请求发送者和接收者之间的耦合，使多个对象都有机会处理请求。这些接受对象串成链，请求沿这条链传递下去，直到被某个对象处理为止。除了希望多个对象都有机会处理请求之外，如果希望向一组对象发送请求，同时不希望明确指定接收者，或者需要动态指定处理请求的一组对象时，都可以使用职责链模式。

能处理就处理，不能处理就调用父亲的方法

拦截器

命令

命令模式将请求封装成对象，使得程序可以用不同的请求将客户参数化，将请求排队或者记入日志，以及支持可撤销的操作。通过使用命令模式，可以将调用操作的对象与执行操作的对象解耦，并且在添加新的命令时，并不会对现有命令产生影响。

解释器

解释器模式针对给定的语言定义及其文法表示，并提供使用该表示的解释器用于解释使用语言书写的句子。对于文法简单的语言，解释其模式是适用的、

文法复杂的语句需要使用特殊的工具来进行处理，解释器模式就显得不恰当了。由于解释器会构建语法树来解释句子，因此其效率会比较低。

自定义文法

迭代器 iterator

迭代器模式提供了一种无须暴露聚合对象底层就可以按顺序访问其各个元素的方式，通过这种方式，可以为不同的聚合对象提供统一的访问接口。

实现内部类：class A ， class A.B，内部类必须依附于一个具体的集合类

静态内部类的好处：在访问外部类的时候很方便，直接访问；不用打包在最后发布的版本里面，方便测试

中介器 mediator

在实现异步通信的时候。有一组对象互相要交互，所有的对象不直接交互，都跟中介进行交互。

中介器模式定义了一个中介对象，封装了一组对象之间的交互逻辑，使得他们不需要显式地互相引用，并且可以互相独立的改变交互方式，从而有利于实现松散耦合。

缺点：终结期可能会成为系统中的瓶颈；出现单一故障结点。

中介 setSuperTitleLowercase

把当前传进来的 title 转换成另一边

备忘录 memento

与 iterator 一样，也是内部类的实现方法。可以直接调用外部类的方法。

备忘录模式无需破坏封装，就可以捕获对象的内部状态，并将其保存在对象外部，使得该对象将来可以恢复到此状态。备忘录实际上是对象状态的快照，相对于直接提供获取对象状态的接口，他可以避免有可能产生的暴露实现细节和破坏封装等潜在的危险。

可以回滚内部的状态。

过滤恶意调价。

观察者 observer

有很多对象依赖于另外一个对象，当这个对象变化时，会有很多对象一起变化。

观察者模式定义了对象间的一对多依赖，当一个对象发生变化的时候，所有依赖于它的对象都会得到通知并自动更新。如果当一个对象发生变化是，它需要通知其他对象随之发生变化，但是无法预知到底需要通知多少以及哪些对象，就可以使用观察者模式。

推模式和拉模式：

被观察的对象要持有观察者的引用，监听器要能够调用被观察者

状态 state

对象会根据自身的状态决定自己的行为。

状态模式允许一个对象在其内部状态发生变化时改变其行为，就好像它所属的类发生了变化一样。当一个对象的行为依赖于其状态，而且在运行时必须根据不同的状态而改变行为时，就可以使用状态模式。

用户发送消息：队列满丢弃，队列不满加到队尾，有效的限制消息的数量，降低中介器的压力

策略 strategy

有一组算法很接近，未来这些算法根据不同的需求做切换，但是希望使用算法本身的代码不要有变化

策略模式定义了一个算法族，对每一个算法都进行了封装，使得他们可以相互替换，这使得这些算法可以独立于使用他们的客户端来进行改变。如果某个算法存在许多不同的辩题，条件不同时使用的算法也不同，就可以使用策略模式。

针对不同购车群体的 bid 方法

用户明确知道要创建哪一种策略

模板方法

访问者