***1.单例模式（Singleton Pattern）***

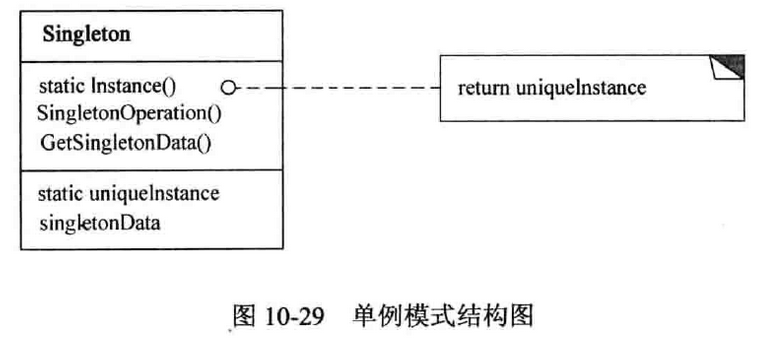
确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。

● 要求生成唯一序列号的环境；

● 在整个项目中需要一个共享访问点或共享数据，例如一个Web页面上的计数器，并确保是线程安全的；

● 创建一个对象需要消耗的资源过多，如要访问IO和数据库等资源；

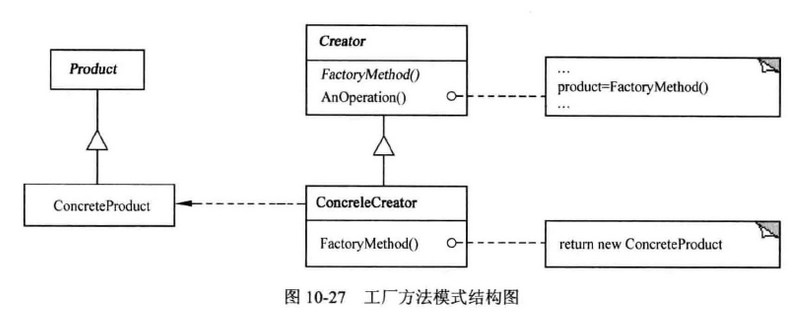
● 需要定义大量的静态常量和静态方法（如工具类）的环境，可以采用单例模式（当然，也可以直接声明为static的方式）。



***2.工厂模式（Factory pattern）***

定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法使一个类的实例化延迟到其子类。创建零件是它的主要职责，组装顺序不是它关心的。

● jdbc连接数据库，硬件访问，降低对象的产生和销毁。

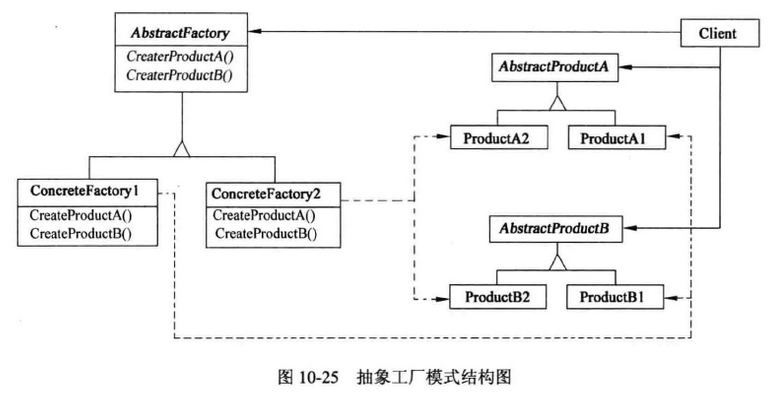


***3.抽象工厂模式（Abstract Factory Pattern）***

为创建一组相关或相互依赖的对象提供一个接口，而且无须指定它们的具体类。

● 一个对象族（或是一组没有任何关系的对象）都有相同的约束；

● 涉及不同操作系统的时候，都可以考虑使用抽象工厂模式。



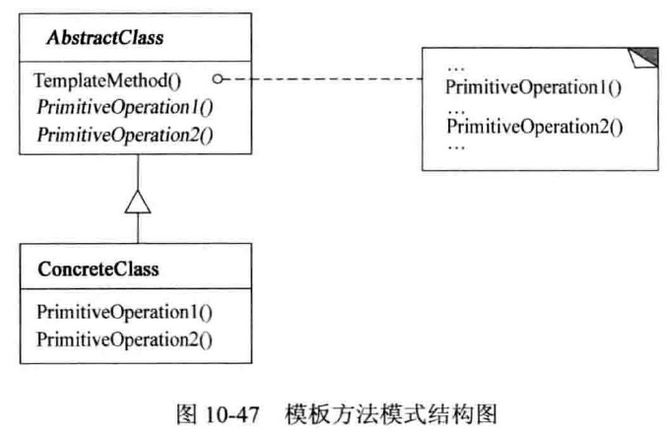
***4.模板方法模式（Template Method Pattern）***

定义一个操作中的算法的框架，而将一些步骤延迟到子类中。使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

● 多个子类有公有的方法，并且逻辑基本相同时。

● 重要、复杂的算法，可以把核心算法设计为模板方法，周边的相关细节功能则由各个子类实现。

● 重构时，模板方法模式是一个经常使用的模式，把相同的代码抽取到父类中，然后通过钩子函数（见“模板方法模式的扩展”）约束其行为。



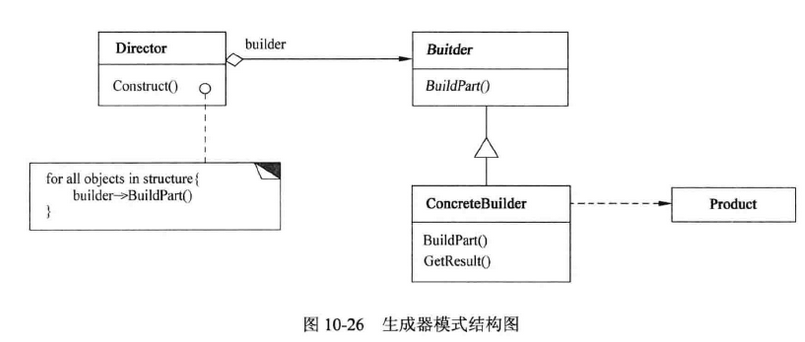
***5.建造者模式（Builder Pattern）***

将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。顺序不同产生的对象也不同。

● 相同的方法，不同的执行顺序，产生不同的事件结果时，可以采用建造者模式。

● 多个部件或零件，都可以装配到一个对象中，但是产生的运行结果又不相同时，则可以使用该模式。

● 产品类非常复杂，或者产品类中的调用顺序不同产生了不同的效能，这个时候使用建造者模式非常合适。



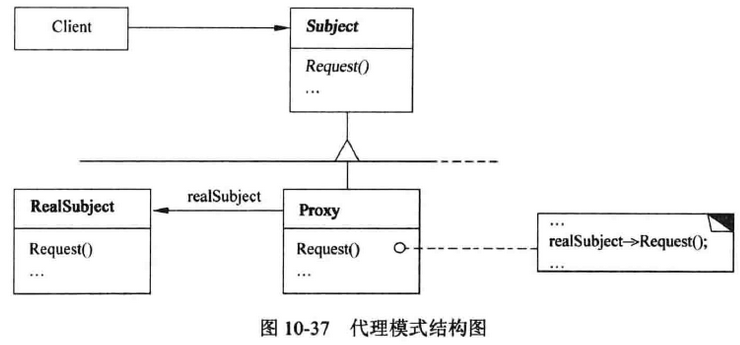
***6.代理模式（Proxy Pattern）***

为其他对象提供一种代理以控制对这个对象的访问。首要条件：被代理的类必须要实现一个接口。

● 当我们想要隐藏某个类时，可以为其提供代理类

● 当一个类需要对不同的调用者提供不同的调用权限时，可以使用代理类来实现

● 当我们要扩展某个类的某个功能时，可以使用代理模式，在代理类中进行简单扩展



***7.原型模式（Prototype Pattern）***

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。实现Cloneable接口，重写clone（）方法。

● 资源优化场景

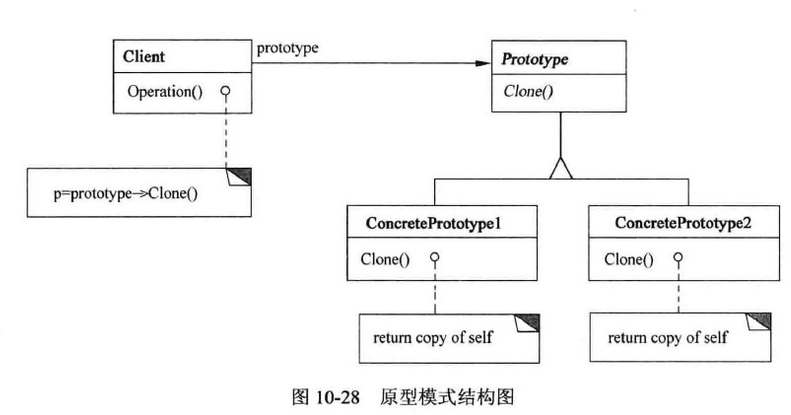
类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等。

● 性能和安全要求的场景

通过new产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式。

● 一个对象多个修改者的场景

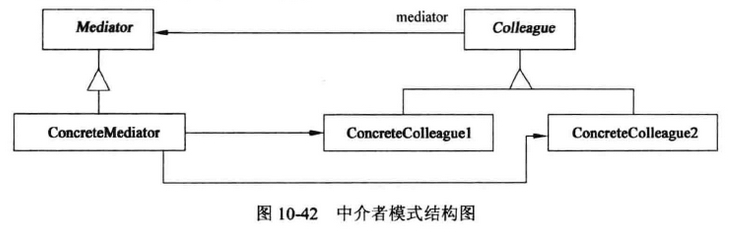
一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用。



***8.中介者模式（Mediator Pattern）***

用一个中介对象封装一系列的对象交互，中介者使各对象不需要显示地相互作用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

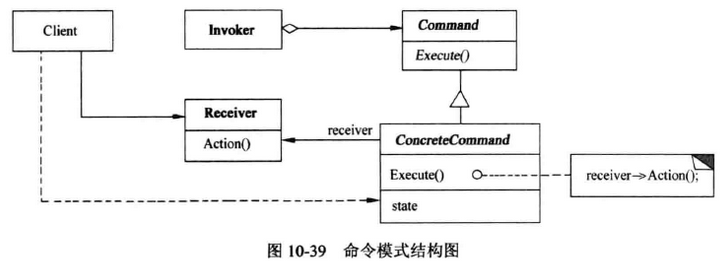
● 式适用于多个对象之间紧密耦合的情况，紧密耦合的标准是：在类图中出现了蜘蛛网状结构，即每个类都与其他的类有直接的联系。



***9.命令模式（Command Pattern）***

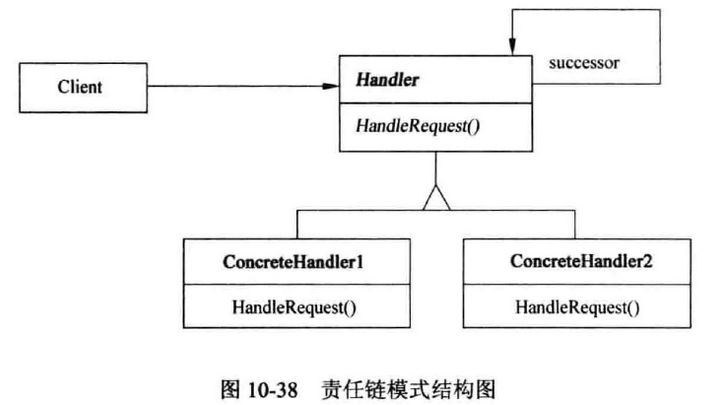
将一个请求封装成一个对象，从而让你使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。

● 认为是命令的地方就可以采用命令模式，触发－反馈机制的处理等。



***10.责任链模式（Chain of Responsibility）***

使多个对象都有机会处理请求，从而避免了请求的发送者和接受者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有对象处理它为止。



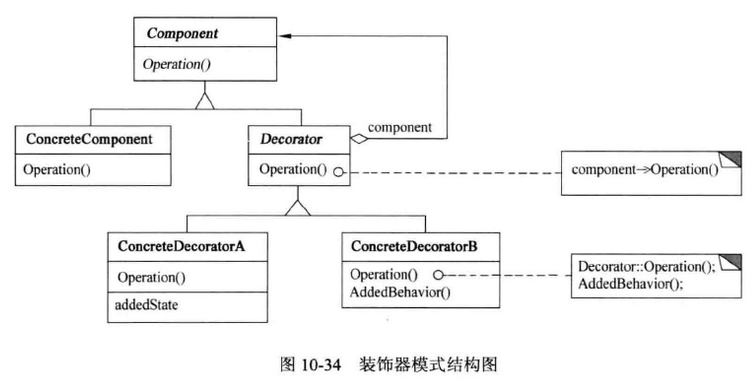
***11.装饰模式（Decorator Pattern）***

动态地给一个对象添加一些额外的职责。就增加功能来说，装饰模式相比生成子类更为灵活。

● 需要扩展一个类的功能，或给一个类增加附加功能。

● 需要动态地给一个对象增加功能，这些功能可以再动态地撤销。

● 需要为一批的兄弟类进行改装或加装功能，当然是首选装饰模式。



***12.策略模式（Strategy Pattern）***

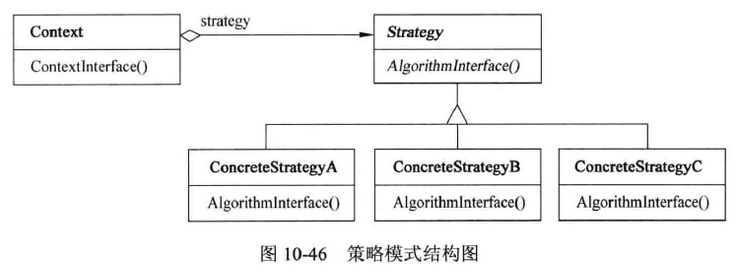
定义一组算法，将每个算法都封装起来，并且使它们之间可以互换。

● 多个类只有在算法或行为上稍有不同的场景。

● 算法需要自由切换的场景。

● 需要屏蔽算法规则的场景。

● 具体策略数量超过4个，则需要考虑使用混合模式



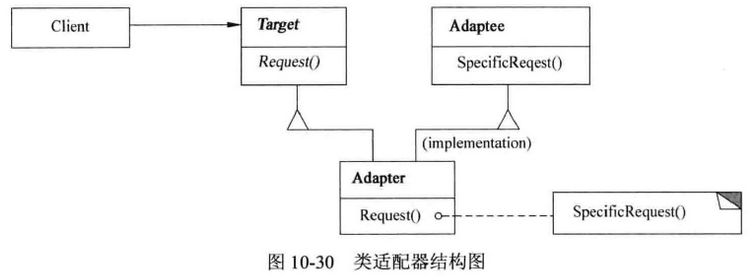
***13.适配器模式（Adapter Pattern）***

将一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

● 你有动机修改一个已经投产中的接口时，适配器模式可能是最适合你的模式。

● 比如系统扩展了，需要使用一个已有或新建立的类，但这个类又不符合系统的接口，使用适配器模式。

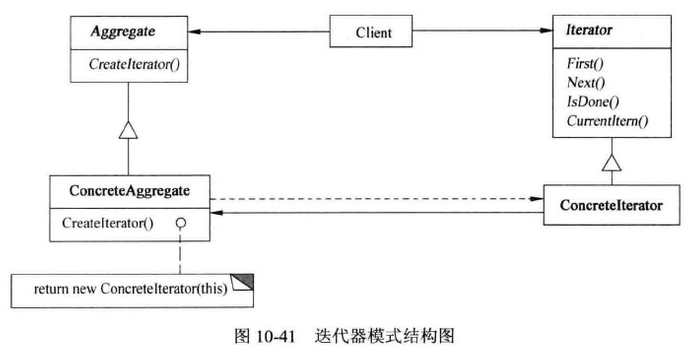
● 详细设计阶段不要考虑使用适配器模式，使用主要场景为扩展应用中。



***14.迭代器模式（Iterator Pattern）***

它提供一种方法访问一个容器对象中各个元素，而又不需暴露该对象的内部细节。

● 使用java自带的迭代器就已经满足我们的需求了。



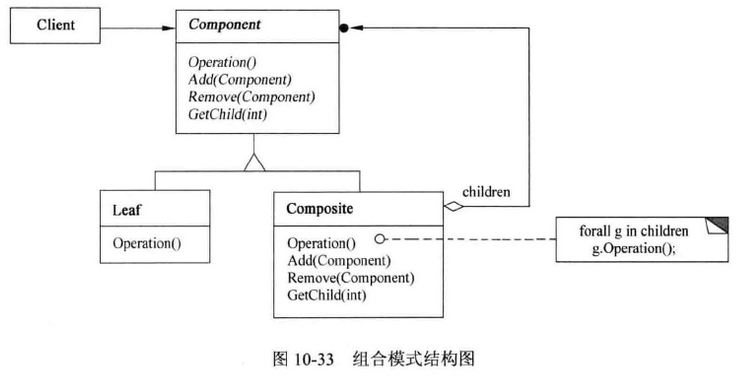
***15.组合模式（Composite Pattern）***

将对象组合成树形结构以表示“部分-整体”的层次结构，使得用户对单个对象和组合对象的使用具有一致性。

● 维护和展示部分-整体关系的场景，如树形菜单、文件和文件夹管理。

● 从一个整体中能够独立出部分模块或功能的场景。

● 只要是树形结构，就考虑使用组合模式。



***16.观察者模式（Observer Pattern）***

定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。

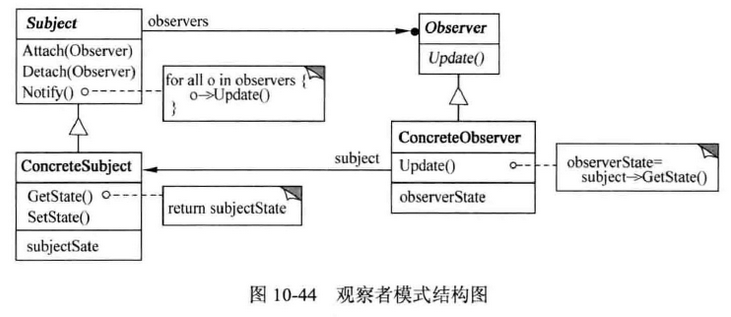
● 关联行为场景。需要注意的是，关联行为是可拆分的，而不是“组合”关系。

● 事件多级触发场景。

● 跨系统的消息交换场景，如消息队列的处理机制。

● 在一个观察者模式中最多出现一个对象既是观察者也是被观察者，也就是说消息最多转发一次（传递两次）。

● 观察者比较多，而且处理时间比较长，采用异步处理来考虑线程安全和队列的问题。



***17.门面模式（Facade Pattern）***

要求一个子系统的外部与其内部的通信必须通过一个统一的对象进行。门面模式提供一个高层次的接口，使得子系统更易于使用。

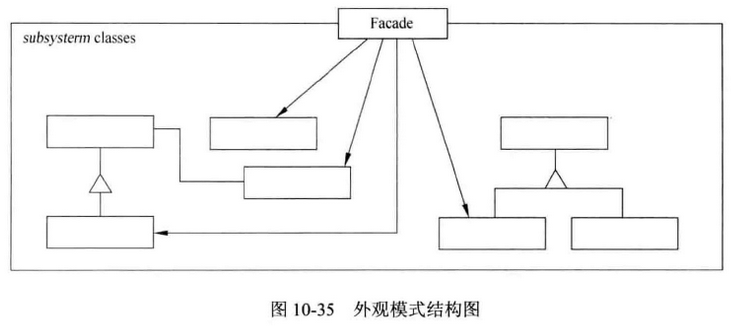
● 为一个复杂的模块或子系统提供一个供外界访问的接口

● 子系统相对独立——外界对子系统的访问只要黑箱操作即可

● 预防低水平人员带来的风险扩散

● 一个子系统可以有多个门面

● 门面不参与子系统内的业务逻辑



***18.备忘录模式（Memento Pattern）***

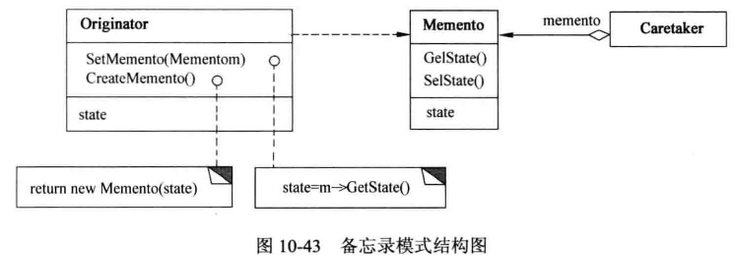
在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。

● 需要保存和恢复数据的相关状态场景。

● 提供一个可回滚（rollback）的操作。

● 需要监控的副本场景中。

● 数据库连接的事务管理就是用的备忘录模式。

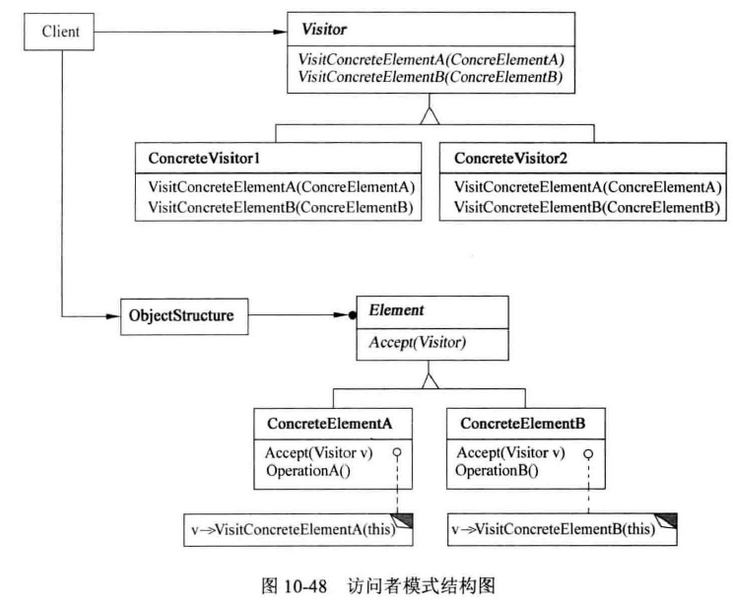


***19.访问者模式（Visitor Pattern）***

封装一些作用于某种数据结构中的各元素的操作，它可以在不改变数据结构的前提下定义作用于这些元素的新的操作。

● 一个对象结构包含很多类对象，它们有不同的接口，而你想对这些对象实施一些依赖于其具体类的操作，也就说是用迭代器模式已经不能胜任的情景。

● 需要对一个对象结构中的对象进行很多不同并且不相关的操作，而你想避免让这些操作“污染”这些对象的类。



***20.状态模式（State Pattern）***

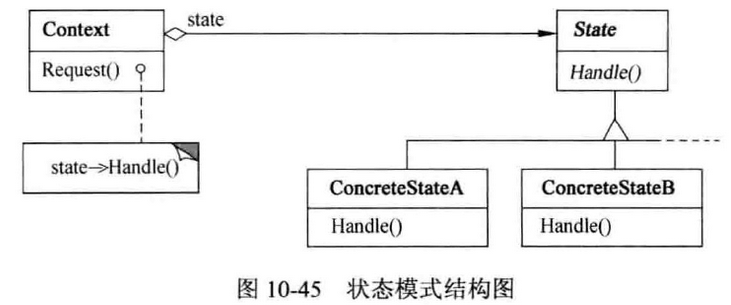
当一个对象内在状态改变时允许其改变行为，这个对象看起来像改变了其类。

● 行为随状态改变而改变的场景

这也是状态模式的根本出发点，例如权限设计，人员的状态不同即使执行相同的行为结果也会不同，在这种情况下需要考虑使用状态模式。

● 条件、分支判断语句的替代者

● 状态模式适用于当某个对象在它的状态发生改变时，它的行为也随着发生比较大的变化，也就是说在行为受状态约束的情况下可以使用状态模式，而且使用时对象的状态最好不要超过5个。

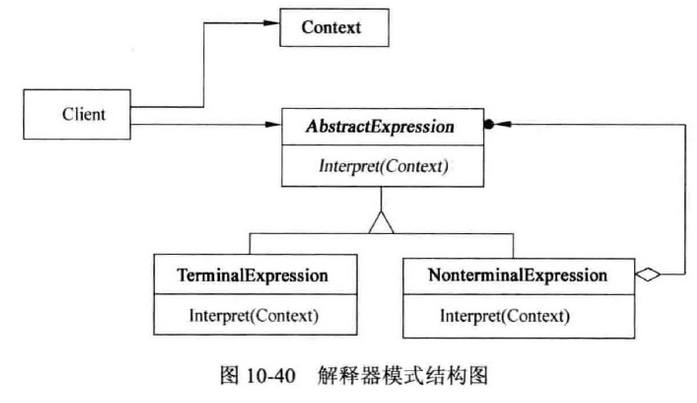


***21.解释器模式（Interpreter Pattern）***

给定一门语言，定义它的文法的一种表示，并定义一个解释器，该解释器使用该表示来解释语言中的句子。

● 重复发生的问题可以使用解释器模式

● 一个简单语法需要解释的场景



***22.享元模式（Flyweight Pattern）***

使用共享对象可有效地支持大量的细粒度的对象。

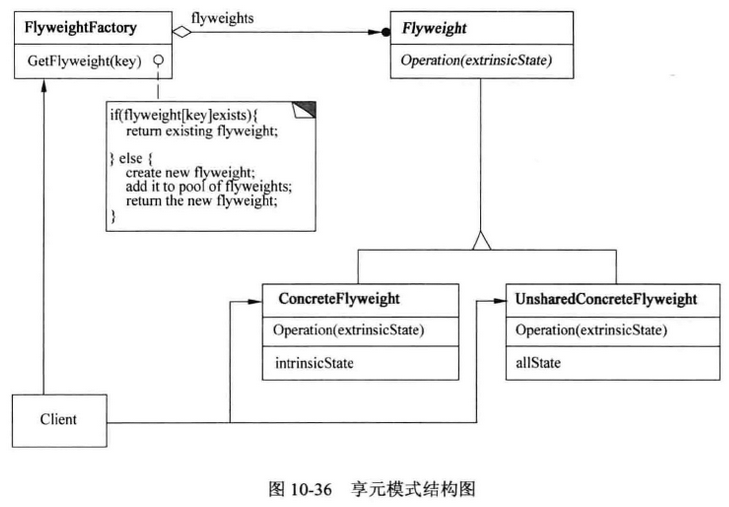
● 系统中存在大量的相似对象。

● 细粒度的对象都具备较接近的外部状态，而且内部状态与环境无关，也就是说对象没有特定身份。

● 需要缓冲池的场景。

● 享元模式是线程不安全的，只有依靠经验，在需要的地方考虑一下线程安全，在大部分场景下不用考虑。对象池中的享元对象尽量多，多到足够满足为止。

● 性能安全：外部状态最好以java的基本类型作为标志，如String，int，可以提高效率。



**23.桥梁模式（Bridge Pattern）**

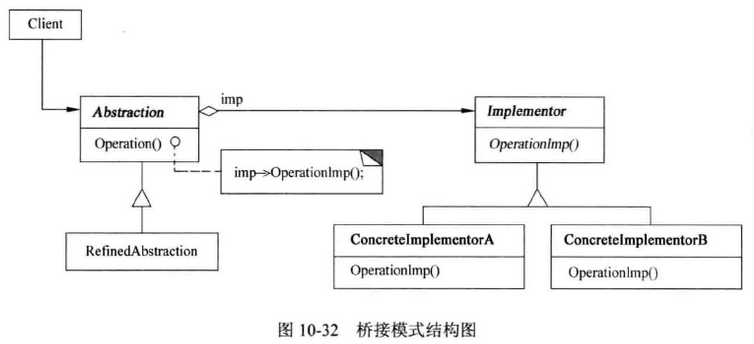
将抽象和实现解耦，使得两者可以独立地变化。

● 不希望或不适用使用继承的场景

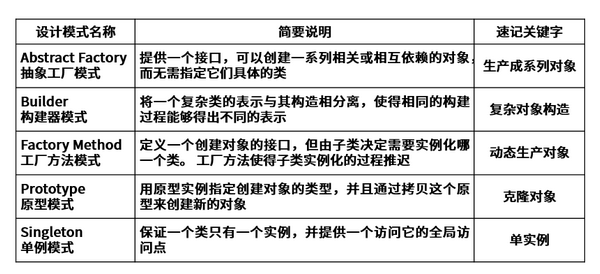
● 接口或抽象类不稳定的场景

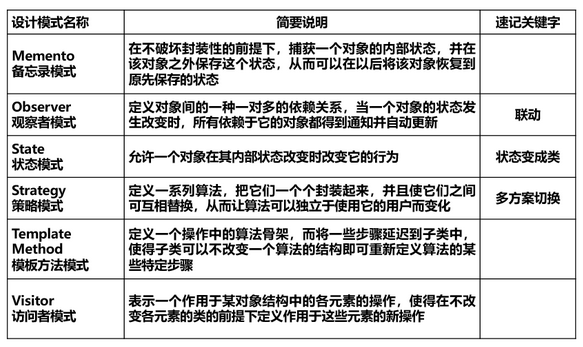
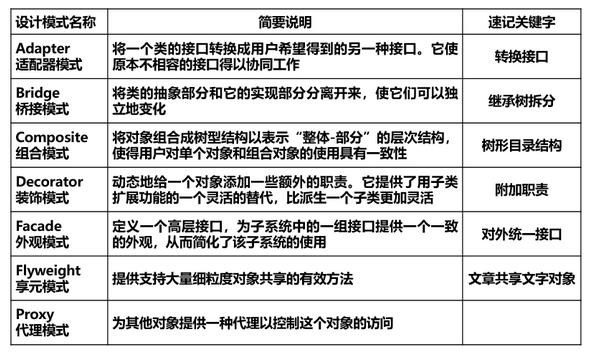
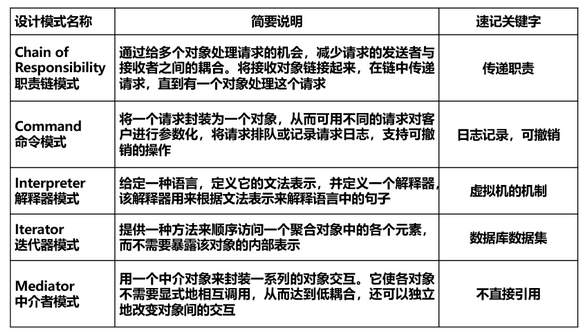
● 重用性要求较高的场景

● 发现类的继承有N层时，可以考虑使用桥梁模式。桥梁模式主要考虑如何拆分抽象和实现。









设计原则：

●Single Responsibility Principle：单一职责原则

单一职责原则有什么好处：

● 类的复杂性降低，实现什么职责都有清晰明确的定义；

● 可读性提高，复杂性降低，那当然可读性提高了；

● 可维护性提高，可读性提高，那当然更容易维护了；

●变更引起的风险降低，变更是必不可少的，如果接口的单一职责做得好，一个接口修改只对相应的实现类有影响，对其他的接口无影响，这对系统的扩展性、维护性都有非常大的帮助。

ps：接口一定要做到单一职责，类的设计尽量做到只有一个原因引起变化。

单一职责原则提出了一个编写程序的标准，用“职责”或“变化原因”来衡量接口或类设计得是否优良，但是“职责”和“变化原因”都是不可度量的，因项目而异，因环境而异。

● Liskov Substitution Principle：里氏替换原则

定义：Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

（所有引用基类的地方必须能透明地使用其子类的对象。）

通俗点讲，只要父类能出现的地方子类就可以出现，而且替换为子类也不会产生任何错误或异常，使用者可能根本就不需要知道是父类还是子类。但是，反过来就不行了，有子类出现的地方，父类未必就能适应。

定义中包含的四层含义：

1.子类必须完全实现父类的方法

2.子类可以有自己的个性

3.覆盖或实现父类的方法时输入参数可以被放大

如果父类的输入参数类型大于子类的输入参数类型，会出现父类存在的地方，子类未必会存在，因为一旦把子类作为参数传入，调用者很可能进入子类的方法范畴。

4. 覆写或实现父类的方法时输出结果可以被缩小

父类的一个方法的返回值是一个类型T，子类的相同方法（重载或覆写）的返回值为S，那么里氏替换原则就要求S必须小于等于T，也就是说，要么S和T是同一个类型，要么S是T的子类。

● Interface Segregation Principle：接口隔离原则

接口分为两种：

实例接口（Object Interface）：Java中的类也是一种接口

类接口（Class Interface）： Java中经常使用Interface关键字定义的接口

隔离：建立单一接口，不要建立臃肿庞大的接口；即接口要尽量细化，同时接口中的方法要尽量少。

接口隔离原则与单一职责原则的不同：接口隔离原则与单一职责的审视角度是不相同的，单一职责要求的是类和接口职责单一，注重的是职责，这是业务逻辑上的划分，而接口隔离原则要求接口的方法尽量少。

● Dependence Inversion Principle：依赖倒置原则

原始定义：

①高层模块不应该依赖低层模块，两者都应该依赖其抽象；

②抽象不应该依赖细节（实现类）；

③细节应该依赖抽象。

依赖倒置原则在java语言中的体现：

①模块间的依赖通过抽象发生，实现类之间不发生直接的依赖关系，其依赖关系是通过接口或抽象类产生的；

②接口或抽象类不依赖于实现类；

③实现类依赖接口或抽象类。

依赖的三种写法：

①构造函数传递依赖对象（构造函数注入）

②Setter方法传递依赖对象（setter依赖注入）

③接口声明依赖对象（接口注入）

使用原则：

依赖倒置原则的本质就是通过抽象（接口或抽象类）使各个类或模块的实现彼此独立，不互相影响，实现模块间的松耦合，我们怎么在项目中使用这个规则呢？只要遵循以下的几个规则就可以：

①每个类尽量都有接口或抽象类，或者抽象类和接口两者都具备

②变量的表面类型尽量是接口或者是抽象类

③任何类都不应该从具体类派生（只要不超过两层的继承是可以忍受的）

④尽量不要复写基类的方法

⑤结合里氏替换原则使用

●Open Closed Principle：开闭原则

定义：软件实体应该对扩展开放，对修改关闭。

其含义是说一个软件实体应该通过扩展来实现变化，而不是通过修改已有的代码来实现变化。

软件实体：项目或软件产品中按照一定的逻辑规则划分的模块、抽象和类、方法。

变化的三种类型：

①逻辑变化

只变化一个逻辑，而不涉及其他模块，比如原有的一个算法是a\*b+c，现在需要修改为a\*b\*c，可以通过修改原有类中的方法的方式来完成，前提条件是所有依赖或关联类都按照相同的逻辑处理。

②子模块变化

一个模块变化，会对其他的模块产生影响，特别是一个低层次的模块变化必然引起高层模块的变化，因此在通过扩展完成变化时，高层次的模块修改是必然的。

③可见视图变化

可见视图是提供给客户使用的界面，如JSP程序、Swing界面等，该部分的变化一般会引起连锁反应（特别是在国内做项目，做欧美的外包项目一般不会影响太大）。可以通过扩展来完成变化，这要看我们原有的设计是否灵活。