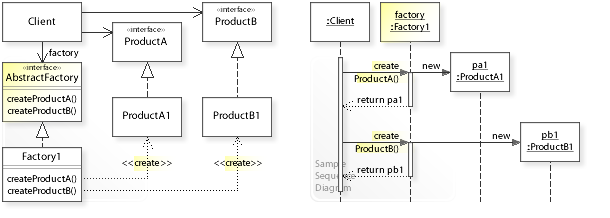
**The GoF Design Patterns Reference**

**GoF设计模式参考**

**一 Creational Patterns**

**1.ABSTRACT FACTORY**



ABSTRACT FACTORY设计模式的目的是：“提供一个接口，以创建相关或依赖对象的族，而不指定它们的具体类。”

ABSTRACT FACTORY设计模式解决了以下问题：

-类如何独立于它所需要的对象如何创建？

-如何创建不同的相关对象或依赖对象族？

一种不灵活的方法是直接在需要这些对象的类（客户端）中创建对象。这将类提交到特定的对象，并使得以后无法独立于（不更改）类来更改实例化。

ABSTRACT FACTORY设计模式描述了如何解决这类问题：

-提供一个接口，用于创建相关或依赖对象的族，而不指定它们的具体类：

AbstractFactory | createProductA(),createProductB(),…

-对象创建过程（例如，new ProductA1()）通过引用接口（委托给工厂对象)进行抽象： factory.createProductA（）。

客户机代码中不再有任何实例化具体类的东西。

优点：

•避免了对编译时实现的依赖关系。

–客户端没有直接实例化具体的类，而是将实例化委托给一个单独的factory对象。

•可确保创建一致的对象族。

–当应用程序支持创建多个相关对象族时，必须确保一起创建和使用一个相关对象族。

•使交换整个对象族很容易。

–因为一个factory对象封装了创建一个完整的对象族，所以可以通过交换factory对象来交换整个族。

缺点：

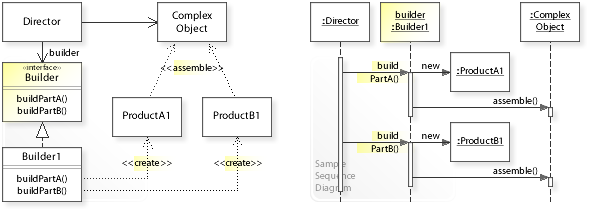
•需要扩展factory接口，以扩展对象族。

–必须扩展factory接口以扩展对象族（以支持新类型的对象）。

•引入了一个额外的间接级别。

–该模式通过引入一个额外的间接级别（客户端将实例化委托给一个单独的factory对象）来实现灵活性，这使客户端依赖于一个factory对象。

**2.BUILDER**



BUILDER设计模式的目的是：“将复杂对象的构造与其表示分开，以便相同的构建过程可以创建不同的表示。”

构建器设计模式解决了以下问题：

-类（相同的构造过程）如何创建复杂对象的不同表示？

•直接在类中创建和组装复杂对象的各个部分，使得不可能独立于（无需更改）类创建不同的表示。

•例如，创建物料清单对象（BOM）。一个类（相同的构造过程）应该有可能会为BOM创建不同的产品结构（表示）。

•生成器模式描述了如何解决此类问题：

-将复杂对象的构造与其表示分开—

-将复杂对象的创建封装在一个单独的生成器对象中。

–类可以通过委托给不同的构建器对象来创建复杂对象的不同表示。

优点：

避免了对编译时实现的依赖关系。

–客户端没有直接实例化具体的类，而是将实例化委托给一个单独的构建器对象。•简化了客户端。

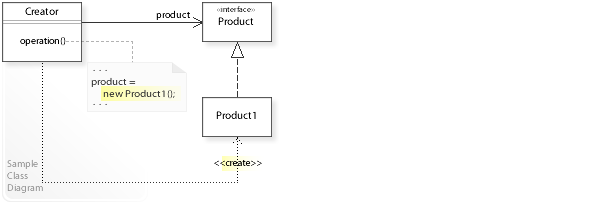
–因为客户端将创建复杂对象委托给构建器对象，因此它们更容易实现、更改、测试和重用。

缺点：

引入了一个额外的间接级别。

–该模式通过引入一个额外的间接级别（客户端将对象创建委托给一个单独的构建器对象）来实现灵活性，这使得客户端依赖于构建器对象。

**3.FACTORY METHOD**



FACTORY METHOD设计模式解决了以下问题：如何创建对象以使子类能够重新定义要实例化的类？类如何将实例化推迟到子类？

一种不灵活的方法是直接在需要（使用）该对象的类（Creator|操作（））中创建一个对象（新产品1（））。

这将类提交（配对）到一个特定的对象，并且使得不可能独立于（无需更改）类来更改实例化（要实例化的类）。

如果我们想创建一个对象，这样子类就可以重新定义创建对象的方式，那么我们就可以避免使用这种方法。

例如，设计需要（依赖于）其他对象的可重用类。一个可重用的类应该避免直接创建它所需要的对象（通常它不知道要实例化哪个类），这样该类的用户就可以编写子类来指定他们所需要的实例化。

优点：

避免了对实现的依赖关系。

–创建者类不直接实例化具体的类。

–它们将实例化延迟到子类（通过调用工厂方法），并且与实例化的具体类无关。

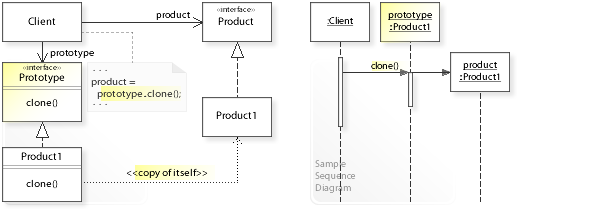
缺点：

避免了对实现的依赖关系。

–创建者类不直接实例化具体的类。

–它们将实例化延迟到子类（通过调用工厂方法），并且与实例化的具体类无关。

**4.PROTOTYPE**



原型设计模式的目的是：“指定要使用原型实例创建的对象的类型，并通过复制此原型来创建新的对象。”

原型设计模式解决了以下问题：

-如何创建对象，以便在运行时指定要创建的哪些对象？

–如何实例化动态加载的类？

原型模式描述了如何解决此类问题：

-指定要使用原型实例创建的对象类型，并通过复制此原型来创建新的对象。

–要作为原型，对象必须实现原型接口（克隆（））来复制自身。

–例如，实现克隆（）操作的产品1对象可以作为创建产品1对象的原型。

优点：

允许在运行时动态地添加和删除原型。

–“，这比其他创建模式更灵活一些，因为客户端可以在运行时安装和删除原型。”允许实例化动态加载的类。

–每个动态加载的类的实例都是自动创建的，并且可以存储在可用原型的注册表中。

–“客户端会在克隆注册表之前向注册表询问一个原型。我们称这个注册表为原型经理。”为工厂方法提供了一个灵活的替代方案。

–原型不需要子类来指定要实例化的类。

–原型将工作在任何工厂方法将，并有更多的灵活性。

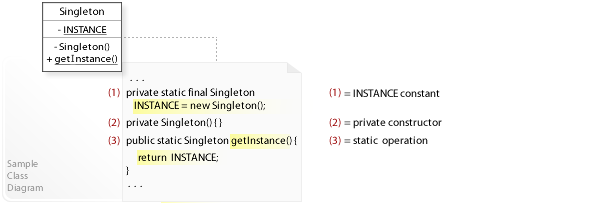
缺点：

可能会使克隆操作的实现变得困难。

–“原型模式中最困难的部分是正确地实现克隆操作。当对象结构包含循环引用时，这尤其棘手。”

-“明智地重写克隆体”

**5.SINGLETON**



SINGLETON设计模式的目的是：“确保一个类只有一个实例，并提供对它的全局访问点。”

单例设计模式解决了以下问题：

-如何确保一个类只有一个实例？

–如何在全局访问类的唯一实例？

例如，保存全局数据的系统对象（如数据库、文件系统、打印机后台处理程序或注册表）。必须确保这些对象只在系统中实例化一次，并且可以从系统的所有部分轻松地访问它们的唯一实例。

提醒一下：全局数据应保持在最低限度（主要由系统对象需要）。在面向对象的方法中，“很少或根本没有全局数据。”相反，数据应该被存储（封装）在那些主要工作的对象中，并在必要时将（作为参数）传递给其他对象。

优点：

可以控制对象的创建。

–get实例操作可以控制创建过程，例如，允许一个类的多个实例。

缺点：

使客户端依赖于具体的单例类。

–这将阻止客户端类的可重用性和可测试性。

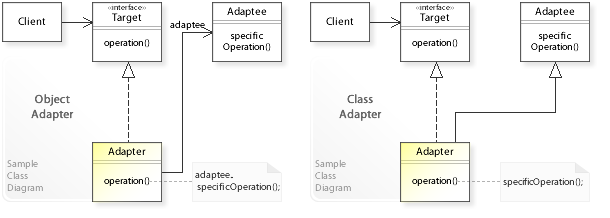
–“将类成为单例会使测试客户端变得困难，因为不可能用模拟实现代替单例，除非它实现作为其类型的接口。”

•可能会在多线程环境中导致问题。

–在多线程应用程序中，必须仔细地实现包含可变数据（即在创建对象后可以更改的数据）的单例数据（同步）。

**二 Structural Patterns**

**1.ADAPTER**



ADAPTER设计模式的目的是：“将类的接口转换为客户端期望的另一个接口”。ADAPTER允许类一起工作，否则不能因为不兼容的接口。”

ADAPTER设计模式解决了以下问题：

-如何重用没有客户端所需要的接口的类？

-具有不兼容接口的类如何一起工作？

通常，一个类（Adaptee）不能仅仅被重用，因为它的接口不符合客户端所需的接口（目标）。

ADAPTER模式描述了如何解决这些问题：

-将一个类的接口转换为客户端所期望的另一个接口。

定义一个单独的ADAPTER类，它将类（ADAPTER）的接口转换为客户端需要的另一个接口（目标）。

-需要目标接口的客户端可以通过ADAPTER来处理具有不兼容（而不是目标接口）接口的类。

优点：

·支持重用现有的功能。

-通常不能只重用现有的可重用对象，因为它的接口与客户端所依赖的接口不匹配。

-通过使用ADAPTER，客户端可以重用提供所需功能但不提供所需接口的现有对象。

·对象ADAPTER比类ADAPTER更灵活。

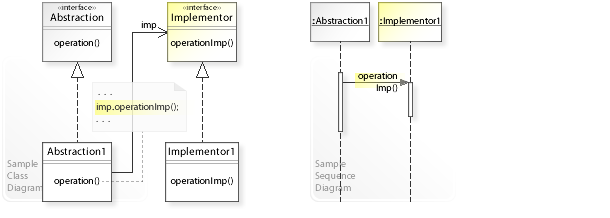
-类ADAPTER根据（通过继承）ADAPTER类实现目标接口。

-这将在编译时将类ADAPTER提交到一个ADAPTER类，并且不适用于其他ADAPTER类。

-此外，如果ADAPTER实现类属于其他应用程序，那么它们通常是隐藏的，不能被访问。

-另一方面，对象ADAPTER在运行时（通过委托）适配对象（独立于适配实现类）实现目标接口。

**2.BRIDGE**



BRIDGE的设计模式的目的是：“从其实现中分离出一个抽象，以便两者可以独立地变化。”

BRIDGE的设计模式解决了以下问题：

-抽象及其实现如何独立变化？

-如何在运行时选择和交换一个实现？

例如，一个支持不同硬件环境的可重用应用程序。为了使一个应用程序可以跨不同的硬件环境进行可移植，应该可以在运行时选择适当的特定于硬件的实现。

BRIDGE模式描述了如何解决这些问题：

-从其实现中分离一个抽象-为抽象（抽象）及其实现（实现者）定义单独的继承层次结构。抽象接口是根据（通过授权）一个实现者或对象来实现的。

优点：

提供了一个替代子类化的灵活的替代方案。

–继承是支持抽象实现的标准方法。

–但是通过继承，一个实现在编译时被绑定到它的抽象上，并且不能在运行时进行更改。

–此外，继承还需要为每个新的抽象扩展类（子类创建新的子类）。

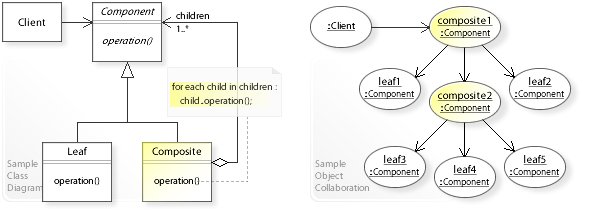
–Bridge便于在运行时动态地组合抽象对象和实现对象。

缺点：

引入了一个额外的间接级别。

–该模式通过引入一个额外的间接级别（抽象将实现委托给一个单独的实现者或对象）来实现灵活性，这使得抽象依赖于一个实现者或对象。

**3.COMPOSITE**



COMPOSITE设计模式的目的是：“将对象组合成树状结构，以表示部分-完整的层次结构。”COMPOSITE允许客户端统一地处理单个对象和对象的组合。”

COMPOSITE设计模式解决了以下问题：

-如何表示一个部分-完整的层次结构，以便客户机能够统一地处理单个对象和对象的组合？

树结构广泛面向对象系统，用于表示层次对象结构，如部分-整体层次结构。

树形结构由单个（叶）对象和子树（复合）对象组成。复合对象具有子对象，即“叶对象”或其他（较低级别的）复合对象。

COMPOSITE设计模式描述了如何解决这些问题：

-将对象组合成树状结构，以表示部分-整个层次结构。

-定义单独的复合对象，将整个层次结构中的对象组合成树结构。

-客户端通过一个公共组件接口来统一地处理叶对象和复合对象，这大大简化了客户端，并使它们更容易实现、更改、测试和重用。

优点：

•简化了客户端。

–客户端可以统一地处理层次结构中的所有对象，这大大简化了客户端代码。

•使添加新组件变得很容易。

–客户端引用了公共组件接口，并独立于其实现。

–这意味着，当添加新的复合类或叶类或扩展现有类时，不必更改客户端。

•允许在运行时动态地构建和更改复杂的层次结构。

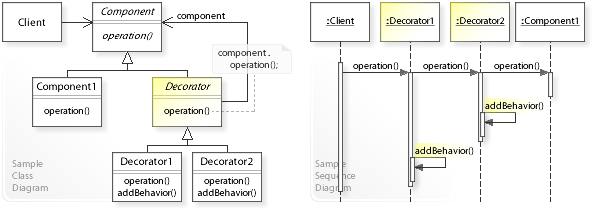
–该模式展示了如何在运行时应用递归组合来动态地构建复杂的分层对象结构。

缺点：

均匀性与类型之间的安全性。

–有两个主要的设计变体来实现与子级相关的操作：一致性设计和类型安全设计。–复合材料的图案强调均匀性，而不是类型的安全性。

**4.DECORATOR**



DECORATOR设计模式的目的是：“动态地向对象附加额外的责任”。DECORATOR为扩展功能提供了一种灵活的替代方案。”

装饰器设计模式解决了以下问题：

-如何将责任动态地添加到对象中？

–如何在运行时扩展对象的功能？

责任是指一个物体提供某种行为的义务。术语责任、行为和功能通常是可互换的。“有时我们想把责任添加到单个对象中，而不是添加到整个类中。例如，图形用户界面工具包应该允许您向任何用户界面组件添加边框等属性或滚动等行为。”

例如，可重用的GUI/Web对象（如按钮、菜单或树形小部件）。应该可以添加装饰物（如边框、滚动条等）。在运行时动态地使用基本的GUI/Web对象。”在装饰者图案中，装饰指的是任何为物体增加责任的东西。”

优点：

•提供了一个替代子类化的灵活的替代方案。

–装饰器提供了一个通过子类化的灵活的扩展功能的选择。

–它很容易组合（混合、排序、复制等）。通过与不同的装饰师协作而实现的功能。

–子类化将需要为每个新的功能组合创建一个新的子类。

•允许添加了一定数量的开放式功能。

–因为装饰器是装饰对象的透明框，它们可以递归嵌套，这允许开放数量的添加功能。

–客户端不知道它们是直接与对象一起工作还是通过其装饰器一起工作。

•简化了类。

–“无需尝试在一个复杂的、可定制的类中支持所有可预见的特性，您可以定义一个简单的类，并使用装饰器对象增量地添加功能。”

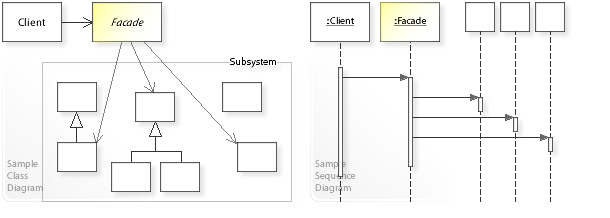
缺点：

不提供对对象标识的可靠性。

–装饰器对象是透明的，但与所装饰的对象不完全相同。

–因此，依赖于对象标识的应用程序不应该使用装饰器。

**5.FACADE**



FACADE设计模式的目的是：“为一个子系统中的一组接口提供一个统一的接口。”FACADE定义了一个更高级别的接口，使子系统更容易使用。”

Facade的设计模式解决了以下问题：

-如何为一个复杂的子系统提供一个简单的接口？

–如何避免客户端和子系统中的对象之间的紧密耦合？

一个复杂的子系统应该提供一个简化的（高级）视图，这对于大多数只需要基本功能的客户端来说足够好。

FACADE模式描述了如何解决这些问题：

-为一个子系统中的一组接口提供一个统一的接口：Facade | operation()

-子系统的客户端仅参考和了解（简单的）Facade接口，并且独立于子系统中的许多不同的接口，这减少了依赖关系，并使客户端更容易实现、更改、测试和重用。

优点：

•从子系统中剥离客户端。

–客户端通过处理一个立面对象来与子系统解耦。

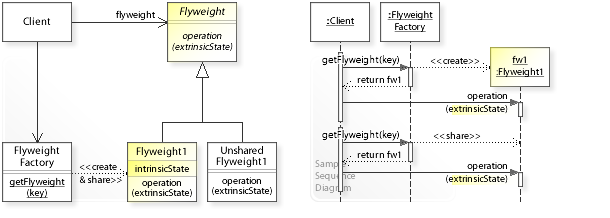
–客户端仅参考和了解简单的Facade接口，并且独立于复杂的子系统（松耦合）。–这使客户机更容易实现、更改、测试和重用。

•分解子系统。

–在为复杂系统进行分层时，立面可以为每个子系统定义单个入口点。

–子系统仅通过它们的外观相互协作，这减少和简化了子系统之间的依赖关系。

**6.FLYWEIGHT**



FLYWEIGHT设计模式的目的是：“使用共享来有效地支持大量的细粒度对象。”

FLYWEIGHT设计模式解决了以下问题：

-如何有效地支持大量细粒度对象？

例如，要在最好的级别上表示一个文本文档，文档中的每个字符都需要一个对象，这可能会导致大量的对象。

FLYWEIGHT模式描述了如何解决这些问题：

-使用共享有效地支持大量的细粒度对象。

-定义存储内在（不变）状态的飞行量对象。客户端共享飞行权重对象，并在调用飞行权重操作（飞行重量。操作（外部状态））时，在运行时动态地传递外部（变体）状态。

-内在状态是不变的（上下文独立的），因此可以共享。外部状态是可变的（依赖于上下文），因此不能共享，必须传入。

优点：

在最好的级别上启用抽象。

–少量物理创建的对象可以表示逻辑上不同的对象的开放式数量。

缺点：

引入了运行时成本。

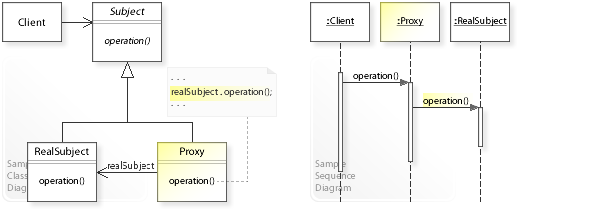
•–客户端负责在运行时动态地传递外部状态。

–每次执行蝇重操作时，存储/检索/计算外部状态都会影响内存使用和系统性能。•不提供对象标识的可靠性。

–相同的物理创建的对象表示许多逻辑上不同的对象。

–因此，依赖于对象标识的应用程序不应该使用蝇权重。

**7.PROXY**



PROXY设计模式的目的是：“为另一个对象提供一个代理或占位符，以控制对它的访问。”

PROXY设计模式解决了以下问题：

-如何控制对对象的访问？

–在访问一个对象时，如何提供其他功能？

例如，应该控制对敏感、昂贵的或远程对象的访问。

PROXY模式描述了如何解决这些问题：

-为其他对象提供代理或占位符，以控制对它的访问。定义一个单独的代理对象，它作为另一个对象（主题）的占位符。代理实现了主题接口，以便它可以在期望使用主题的任何地方充当占位符。

-通过代理对象来控制对（已存在的）对象的访问。

优点：

简化了客户端。

–代理向客户机隐藏了实现细节，这使它们更容易实现、更改、测试和重用。

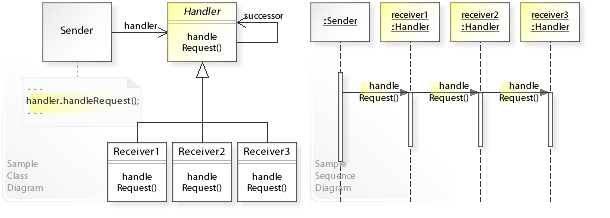
缺点：

代理与真实的主体是耦合的。

–一个代理实现了主题接口，并且（通常）可以直接访问具体的实际主题类。–“但是如果代理要实例化实际对象（比如在虚拟代理中），那么它们必须知道具体的类。”

**四 Behavioral Patterns**

**1.CHAIN OF RESPONSIBILITY**



CHAIN OF RESPONSIBILITY设计模式的目的是：“通过给多个对象一个处理请求的机会，避免将请求的发送者与接收者耦合起来。”将接收对象链连接起来，并沿着链传递请求，直到对象处理它。”

责任链设计模式解决了以下问题：

-如何避免将请求的发送器耦合到其接收器？

–多个对象如何处理一个请求？

请求是一个对象（发送方）对另一个对象（接收方）执行的操作。

一种不灵活的方法是直接在发送请求的类中实现一个请求。这将请求的发送器在编译时耦合到特定的接收器。

责任链模式描述了如何解决此类问题：

-链接接收对象，并将请求传递链直到对象处理为止。

–定义和链化可处理或转发请求的处理程序对象。这将导致一系列的对象有责任处理一个请求。

优点：

•从接收方中解密发送方。

–该模式通过将请求发送到一个处理程序链，从而将请求的发送者与特定的接收器（处理程序）解耦。

•使更改处理程序链变得很容易。

–处理程序链可以在运行时进行更改（处理程序可以添加到链中并从链中删除）。

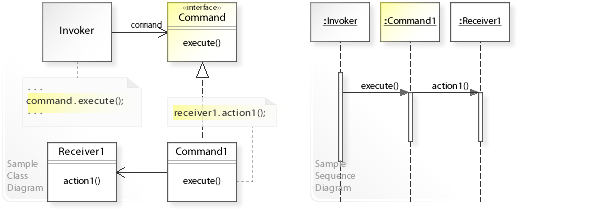
缺点：

后续链可以很复杂。

–如果没有现有的可以用来定义和维护后续链的对象结构，那么可能很难实现和维护该链（请参见实现）。

–“如果责任链已经是系统结构的一部分，那么它是分离发送方和接收方的好方法，并且其中一个对象可以处理请求。”

**2.COMMAND**



COMMAND设计模式的目的是：“将一个请求封装为一个对象，从而允许您使用不同的请求、队列或日志请求来参数化客户端，并支持不可执行的操作。”

COMMAND设计模式解决了以下问题：

—如何避免将请求的调用器耦合到请求中？

–如何用一个请求来配置一个对象？

请求是一个对象对另一个对象执行的操作。从更一般的角度来看，请求是要执行的任意操作。术语请求、消息、操作和方法通常就像执行、发出和发送请求一样可以互换。

COMMAND设计模式描述了如何解决此类问题：

-将请求封装为一个对象-定义单独的类（命令1，...），以实现（封装）请求，并定义一个公共接口（命令|执行（）），通过它可以执行请求。

优点：

使添加新命令变得容易。

–“添加新的命令很容易，因为你必须改变现有的类。使交换命令变得容易。

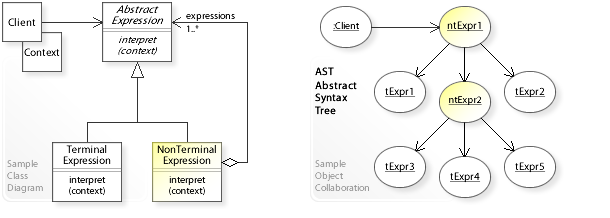
–对象可以配置所需的命令对象，更多，命令可以在运行时动态交换。

缺点：

额外的间接水平。

–命令通过引入一个额外的间接级别（调用者将一个请求委托给一个单独的命令对象）来实现灵活性，这使得调用者依赖于一个命令对象。

**3.INTERPRETER**



INTERPRETER设计模式的目的是：“给定一种语言，为其语法定义一种表示，并使用该表示来解释语言中的句子。”

INTERPRETER设计模式解决了以下问题：

-如何定义一种简单语言的语法，从而能够解释该语言中的句子？

术语和定义：

-一种语言是一组有效的句子。

–一个句子=语句=表达式。[…表达式是语句的核心组成部分[…表达式是由变量、操作符和方法调用组成的构造，它们根据语言的语法构造，计算结果为单个值。”-语法是一种正式描述一种语言的结构（语法）的方式。这是一系列的规则，解释者用来解释语言中的句子。最常见的语法符号是扩展的Backus-Naur形式（EBNF）。

INTERPRETER设计模式描述了如何解决这些问题：

-给定一种语言，定义一个表示语法，通过定义一个表达式类层次结构以及解释器使用表示解释句子的语言

-语言中的每一个句子是由一个抽象的语法树（AST）由表达式类的实例。一个句子可以通过召唤来解释。

优点：

•使改变语法变得很容易。

–解释器模式使用类层次结构来表示语法规则。

–客户端引用了抽象表达式接口，并独立于其实现。当添加新的终端或非终端表达式类时，–客户端不必进行更改。

•使添加新类型的解释操作更容易。

–“[……许多种操作都可以“解释”一个句子。通常，一个解释器被认为是在解释一个表达式并返回一个简单的结果，但也可以执行任何类型的操作。

–访问者模式可以用于定义新的解释操作类型，而不必更改现有的表达式类层次结构。

缺点：

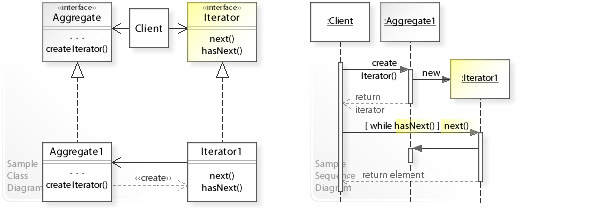
使表示复杂的语法变得困难。

–解释器模式使用（至少）一个类来表示每个语法规则。

–因此，对于复杂的语法，类层次结构会变得很大且难以维护。

–解析器生成器是一种替代方法。它们可以表示复杂的语法，而无需构建复杂的类层次结构。

**4.ITERATOR**



ITERATOR设计模式的目的是：“提供一种方法来按顺序访问聚合对象的元素，而不公开其底层表示。”

迭代器设计模式解决了以下问题：

-如何在不公开其底层表示的情况下访问和遍历聚合对象的元素？

例如，一个聚合对象，如列表、集合或其他类型的集合。应该可以访问和遍历集合中的元素，而不需要知道它的底层表示（数据结构）。

不可能公开聚合的表示，因为这会破坏它的封装。

优点：

•启用同时进行的遍历。

–可以在同一聚合体上进行多个遍历。

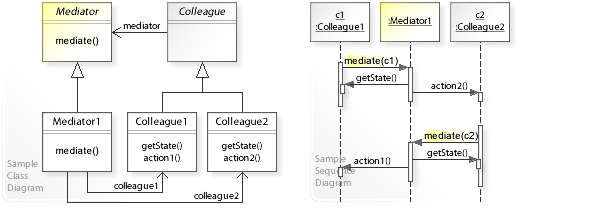
•简化了聚合接口。

–迭代器接口被分离出来，这简化了聚合接口。

•允许在运行时动态地更改遍历。

–“迭代器使更改遍历算法变得很容易：只需用不同的迭代器实例替换迭代器实例。”

**5.MEDIATOR**



MEDIATOR设计模式的目的是：“定义一个封装一组对象如何交互的对象。”中介通过阻止对象显式地引用彼此来促进松散耦合，它允许你独立地改变它们的交互。”

中介设计模式解决了以下问题：

-如何避免一组交互对象之间的紧密耦合？

–如何独立地改变一组对象之间的交互？

耦合是指物体相互依赖的程度。

–紧密耦合的对象很难实现、更改、测试和重用，因为它们依赖于（参考和了解）许多不同的对象。

–松散耦合的对象更容易实现、更改、测试和重用，因为它们对其他对象的依赖性很小。

中介器模式描述了如何解决此类问题：

-定义一个对象（中介器），它封装了一组对象如何交互。

–这种模式的关键思想是让对象通过一个控制和协调交互的中介对象间接地相互交互。这使得对象松散耦合，因为它们只依赖于（参考和知道）简单的中介接口。

优点：

•分解同事。

–同事们通过中介对象/接口间接地相互作用，彼此之间没有明确的知识。

–松散耦合的对象更容易实现、更改和重用。

•集中化了交互行为。

–中介封装（集中）交互行为，否则就会分布在交互的同事之间。

•使更改交互行为变得很容易。

–通过添加新的中介（子）类，可以独立于同事来改变交互行为。

缺点：

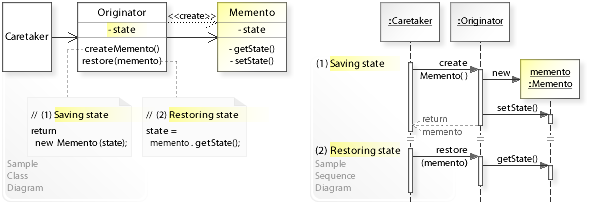
可以使中介物变得很复杂。

–因为中介封装（集中）了一组对象的交互行为，所以它可能会变得复杂。

–复杂性随着同事的复杂性和数量的增加而增加。

–“这可以使调解人本身成为一个难以维持的巨石。”

**6.MEMENTO**



MEMENTO设计模式的目的是：“在不违反封装的情况下，捕获和外部化对象的内部状态，以便对象以后可以恢复到此状态。”

Memento设计模式解决了以下问题：

-在不违反封装的情况下，如何捕获对象的内部状态并外部化，以便对象稍后恢复到此状态？

封装意味着隐藏对象的表示形式（数据结构，如字段、数组、集合等）。并在对象内部实现，以便不能从对象外部访问它们。

对象的内部状态表示所有内部数据结构加上它们的值。

这里的问题是保存对象的内部状态，而不使表示（数据结构）可见，并可从对象外部访问。

优点：

保留封装功能。

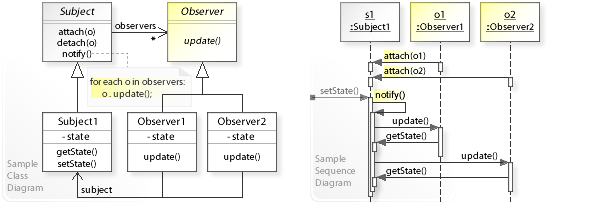
–对象的内部状态可以保存在外部（保存到另一个对象），而不违反封装（不允许内部可访问数据结构）。

缺点：

可能会引入运行时成本。

–使用大量的数据创建大量的备忘录可能会影响内存的使用和系统的性能。–“除非封装和恢复发起者状态很便宜，否则该模式可能不合适。”

**7.OBSERVER**



OBSERVER设计模式的目的是：“定义对象之间的一对多依赖关系，以便当一个对象改变状态时，其所有依赖关系都会自动得到通知和更新。”

观察者设计模式解决了以下问题：

-如何在不使对象紧密耦合的情况下定义对象之间的一对多依赖关系？

–一个对象如何通知其他对象的开放式数量？

耦合是指物体相互依赖的程度。

–紧密耦合的对象很难实现、更改、测试和重用，因为它们依赖于（参考和了解）许多不同的对象（具有不同的接口）。

–松散耦合的对象更容易实现、更改、测试和重用，因为它们对其他对象的依赖性很小。

观察者模式描述了如何解决这些问题：

-定义对象之间的一对多依赖关系，以便当一个对象改变状态时，其所有依赖关系将自动得到通知和更新。

–这种模式的关键思想是建立一个灵活的通知注册机制，当感兴趣的事件发生时，它会自动通知所有已注册的对象。

优点：

•从观察员的主题。

–主题仅参考并知道用于更新（同步）状态（更新（））的简单观察者界面。

–“因为主题和观察者并不是紧密耦合的，所以它们可以属于系统中的不同抽象层。松散耦合的对象更容易实现、更改、测试和重用。

•使添加/退出观察者变得很容易。

–观察者可以独立地添加到（动态地附加(o)）并从主题中退出。

–通常，观察员负责对一个主题进行注册和注销。

–受试者的唯一责任是持有观察者名单，并在其状态发生变化时通知（呼叫更新（））他们。

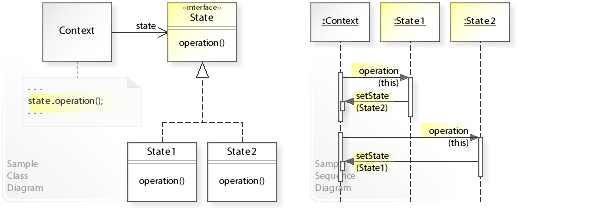
缺点：

可以使更新行为更加复杂。

–对主题的变化可能会导致观察者及其依赖对象的一系列更新。

–中介设计模式可应用于实现主体(s)和观察者之间的复杂依赖关系。

**8.STATE**



STATE设计模式的目的是：“允许对象在其内部状态更改时改变其行为。”这个对象似乎会改变它的类。”

状态设计模式解决了以下问题：当对象内部状态改变时如何改变其行为？

–如何定义特定于状态的行为，以便可以添加新的状态，并可以独立地改变现有状态的行为？

例如，订单处理系统中的一个销售订单对象。一个销售订单对象可以处于不同的状态之一。当它接收到请求时，它的行为根据其当前内部状态而不同。应该可以独立于（无需更改）销售订单类来添加新的状态和更改现有状态的行为。

优点：

•使添加新的状态变得容易。

–“因为所有特定于状态的代码都存在于一个状态子类中，所以通过定义新的子类，可以轻松地添加新的状态和转换。”避免了在状态之间切换的条件语句。

–上下文不是将在不同状态之间切换的硬编码多个/大型条件语句，而是将特定于状态的行为委托给其当前状态对象。

–“，它在代码上强加了结构，并使其意图更清晰。”确保了一致的状态。通过替换

–上下文的当前状态对象来替换

–上下文的状态。这可以避免不一致的内部状态。

•可以明确地进行状态转换。

–“为不同的状态引入不同的对象，使转换更加明确。”

缺点：

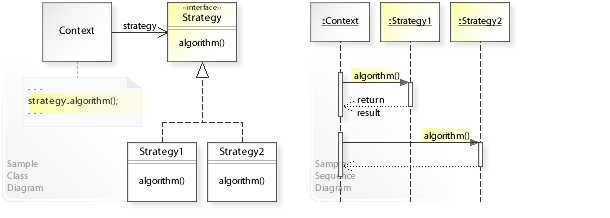
•可能需要扩展上下文接口。

–上下文接口可能必须进行扩展，以让状态对象改变上下文的状态。

•引入了一个额外的间接级别。

–State通过引入额外的间接级别（客户端委托给单独的状态对象）来实现灵活性，这使客户端依赖于状态对象。

**9.STRATEGY**



STRATEGY设计模式的目的是：“定义一系列算法，封装每一个算法，并使它们可互换。”STRATEGY允许算法独立于使用它的客户端而有所不同。”

策略设计模式解决了像这样的问题：

-如何在运行时为一个类配置一个算法，而不是直接实现一个算法？

–如何在运行时选择和交换一个算法？

术语算法通常被定义为一个以一些值作为输入，执行有限数量的步骤，并产生一些值作为输出的过程。从更一般的角度来看，算法是做适当的事情的任意代码。例如，在订单处理系统中计算价格。以不同的方式计算价格（取决于运行时条件，如客户类型、销量、产品数量等），应该可以在运行时选择正确的定价算法（定价“策略”）。

策略模式描述了如何解决这些问题：

-定义一系列算法，封装每个算法，-定义单独的类（策略1，策略2，...）来实现（封装）每个算法，并使它们可互换，并定义一个共同的接口（策略），通过它算法可以在运行时（内部）改变。

优点：

•避免了对编译时实现的依赖关系。

–客户端引用了一个接口（策略），并且独立于一个实现。

•为子类化提供了一个灵活的替代方案。

–子类化提供了一种更改类的算法的方法（在编译时）。当一个子类被实例化时，它的算法是固定的，并且不能在对象的生命周期内进行更改。

–策略提供了一种通过委托给不同的策略对象来更改对象的算法（在运行时）的方法。

•避免了在算法之间切换的条件语句。

–在不同算法之间切换的条件语句被委托给不同的策略对象所取代。包含许多条件语句的

–“代码通常表明需要应用策略模式。”

缺点：

•可以使通用的策略界面变得复杂起来。

–策略接口可能会变得复杂，因为它必须传递所有受支持的算法所需的数据（无论它们是简单的还是复杂的）。

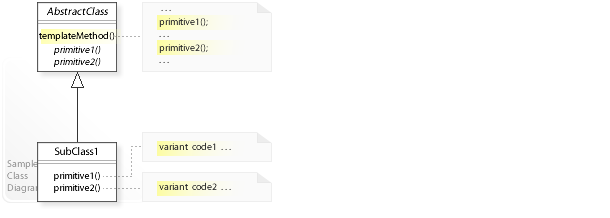
•要求客户了解策略的不同程度。

–“该模式有一个潜在的缺点，即客户必须在选择合适的策略之前才能了解策略的不同。客户可能会面临实施问题。”

–策略通过引入一个额外的间接级别（客户端将一个算法委托给一个单独的策略对象）来实现灵活性，这使得客户端依赖于一个策略对象。

–这“会使设计复杂化和/或花费你一些性能。设计模式只有在实际需要其提供的灵活性时才应该应用。”

**10.TEMPLATE METHOD**



TEMPLATE METHOD设计模式的目的是：“在操作中定义算法的骨架，将一些步骤延迟到子类。”模板方法允许子类不改变算法的某些步骤而不改变算法的结构。

模板方法设计模式解决了这样的问题：

-如何实现行为的不变部分，以便子类能够实现可变部分？

–子类如何在不改变行为结构的情况下重新定义行为的某些部分（算法的步骤）？

标准的方法是定义重新定义父类行为的子类。这使得不可能独立于（不复制）其他部分而只重新定义行为的某些部分。

模板方法模式描述了如何解决这些问题：

-在操作中定义算法的骨架，将一些步骤延迟到子类。

–为行为的每个变体部分定义一个原语操作（原语1（），原语2（），...）。

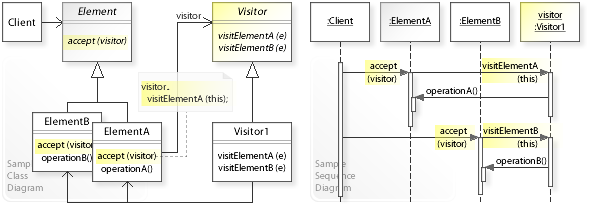
–模板方法（）通过实现不变部分和调用原语来定义行为的骨架（结构），以推迟实现变量部分到子类。

优点：

模板方法是代码重用的一种基本技术。它们在类库中特别重要，因为它们是分解库类中常见行为的方法。”

-模板方法的反转导致控制的反转，因为子类不再控制如何重新定义父类的行为。

**11.VISITOR**



VISITOR设计模式的目的是：“表示要对对象结构的元素执行的操作”。访问者允许您定义一个新的操作，而不改变它所操作的元素的类。”

访问者设计模式解决了以下问题：

-如何在不更改类的情况下为对象结构的类定义新的操作？

例如，表示技术设备组件的对象结构（材料清单）。许多不同的应用程序都使用对象结构，并且应该可以独立于（无需更改）对象结构的类来定义新的操作。

访问者模式描述了如何解决此类问题：

-表示要对对象结构的元素执行的操作。

–定义一个单独的访问者对象，它表示要对对象结构的元素执行的操作。

–为每个元素定义调度操作“接受（访问对象）”，以便“调度”（委派）客户端请求到“接受访问对象”。

–客户端遍历对象结构并在每个元素上调用接受（访问者）（通过传入访问者对象）。

优点：

•使添加新的操作变得容易。

–“您可以通过向访问者类层次结构中添加一个新的子类来创建一个新的操作。”允许跨继承层次结构访问不同类型的元素。

–访问者可以访问没有公共接口的元素，即，它可以访问不同类型的元素（ElementA、ElementB、...），这些元素不必通过继承进行关联。

•使积累状态变得容易。

–访问者使得在遍历对象结构时可以很容易地积累状态。

–它消除了将状态传递给执行累积的操作的需要。状态被累积并存储在访问者对象中。

缺点：

•需要扩展访问者接口以支持新的元素类。

–必须扩展访问者接口，以支持对象结构中的新元素类。

–因此，只有当对象结构稳定且不经常添加新的元素类时，才应该使用访问者模式。

•可能需要扩展元素接口。

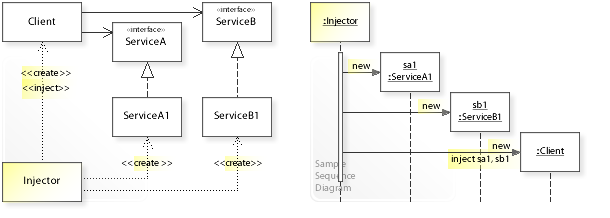
–元素接口可能必须扩展，让所有访问者完成他们的工作并访问所需的数据和功能。

•引入了额外的间接级别。

–该模式通过引入单独的访问者对象和双调度机制来实现灵活性，这可能会使设计复杂化。

**五 GoF设计模式更新**

**DEPENDENCY INJECTION**



依赖项注入设计模式的目的是：将对象创建与应用程序分开。依赖注入使应用程序与其对象的创建方式无关。

依赖性注入设计模式解决了以下问题：

-一个类如何独立于它所需要的对象的创建方式？

–如何在单独的配置文件中指定对象的创建方式？

一种不灵活的方法是直接在需要这些对象的类（客户端）中创建对象。这将类提交给对象的创建方式，并使得以后无法独立于类（不更改）类来更改实例化。

依赖注入模式描述了如何解决此类问题：

-将对象创建与应用程序分离：

-定义一个单独的注入器对象，用于创建和注入类所需的对象。

–类接受注入器对象所需要的对象，而不是直接创建对象。

优点：

•避免了对编译时实现的依赖关系。

–类在运行时注入它们的对象，并且独立于（不知道）哪些具体类是实例化的。

•大大简化了类。

–类在运行时自动注入它们的对象，而不必创建它们，这使得类更容易实现、更改、测试和重用。

•使更改应用程序的配置变得很容易。

–因为对象的创建方式是在单独的配置文件中定义的，因此可以通过使用不同的配置文件轻松地更改应用程序的配置。

•确保对象配置正确。–在使用构造函数注入时，会在使用对象之前先创建并注入对象的依赖关系。