目录

[一、 设计模式的目的 2](#_Toc296337583)

[二、 组织编目 3](#_Toc296337584)

[三、 创建型设计模式 5](#_Toc296337585)

[a) 抽象工厂（Abstract Factory） 5](#_Toc296337586)

[b) 生成器——Builder 6](#_Toc296337587)

[c) 工厂方法——Factory Method 7](#_Toc296337588)

[d) 原型模式——Prototype 8](#_Toc296337589)

[四、 结构型模式 10](#_Toc296337590)

[a) 适配器——Adapter 10](#_Toc296337591)

[b) 桥接——Bridge 11](#_Toc296337592)

[c) 组合——Composite 12](#_Toc296337593)

[d) 装饰——Decorator 13](#_Toc296337594)

[e) 外观——Facade 14](#_Toc296337595)

[f) 蝇量、享元——FlyWeight——128 15](#_Toc296337596)

[g) 代理——Proxy——137 17](#_Toc296337597)

[h) 综合讨论——144 18](#_Toc296337598)

[五、 行为模式 18](#_Toc296337599)

[a) 职责链——Chain of Responsibility 18](#_Toc296337600)

[b) 命令——Command 19](#_Toc296337601)

[c) 解释器——Interpreter 20](#_Toc296337602)

[d) 迭代器——Iterator 21](#_Toc296337603)

[e) 中介者——Mediator 22](#_Toc296337604)

[f) 备忘录——Memento 23](#_Toc296337605)

[g) 观察者——Observer 24](#_Toc296337606)

[h) 状态——State 25](#_Toc296337607)

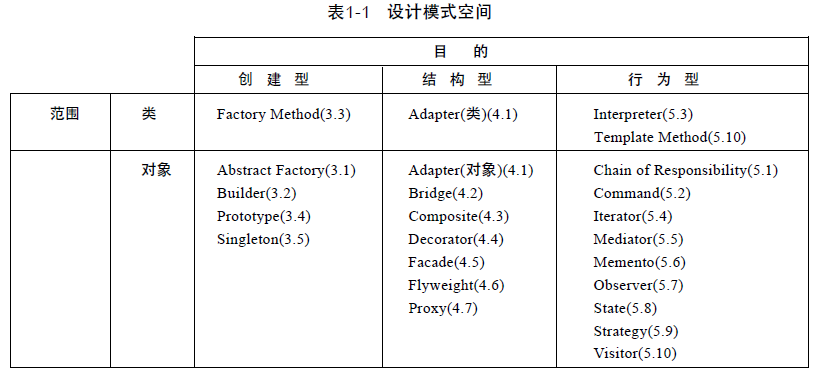
[i) 策略——Strategy——208 26](#_Toc296337608)

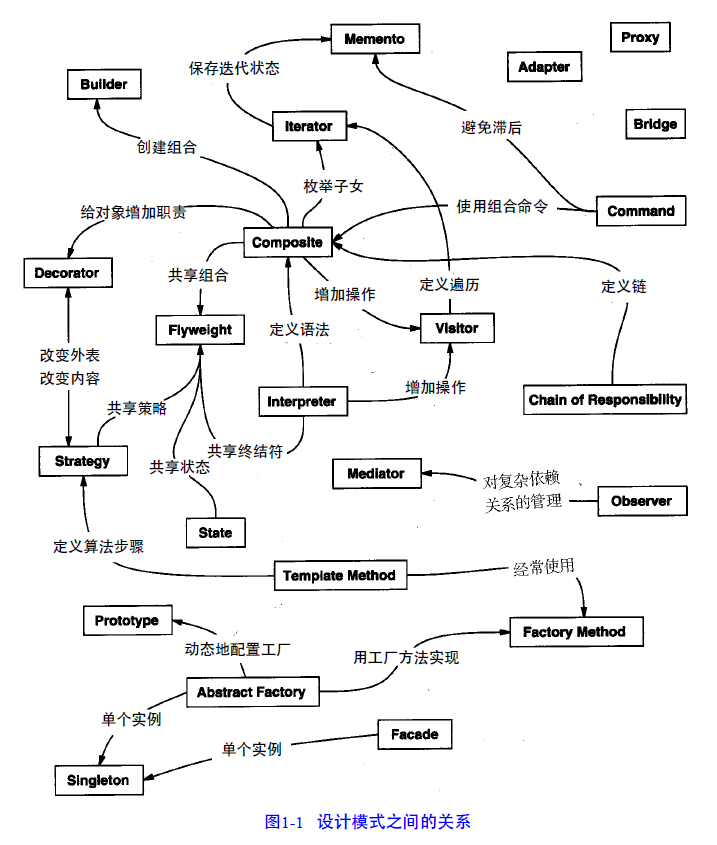
[j) 模板方法——Template Method 27](#_Toc296337609)

[k) 访问者——Visitor——218 28](#_Toc296337610)

[**六、** **词汇表** 30](#_Toc296337611)

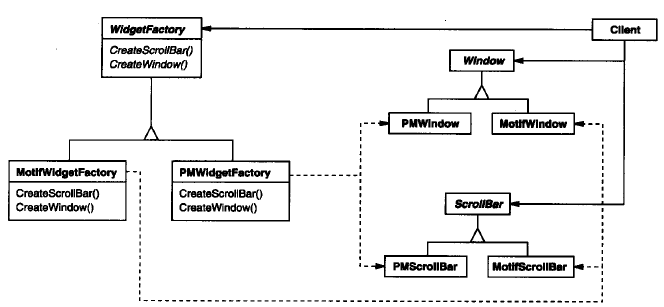
1. 设计模式的目的
   * + 1. Abstract Factory( 3 . 1 )：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。
       2. A d a p t er ( 4 . 1 )：将一个类的接口转换成客户希望的另外一个接口。A d a p t e r模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。
       3. B r i d g e(4. 2)：将抽象部分与它的实现部分分离，使它们都可以独立地变化。
       4. B u i l d e r(3 . 2)：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。
       5. Chain of Responsibility( 5 . 1 )：为解除请求的发送者和接收者之间耦合，而使多个对象都有机会处理这个请求。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它。
       6. C o m m a n d( 5 . 2 )：将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可取消的操作。
       7. C o m p o s i t e( 4 . 3 )：将对象组合成树形结构以表示“部分-整体”的层次结构。C o m p o s i t e使得客户对单个对象和复合对象的使用具有一致性。
       8. D e c o r a t o r( 4 . 4 )：动态地给一个对象添加一些额外的职责。就扩展功能而言， D e c o r a t o r模式比生成子类方式更为灵活。
       9. F a c a d e( 4 . 5 )：为子系统中的一组接口提供一个一致的界面， F a c a d e模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。
       10. Factory Method( 3 . 3 )：定义一个用于创建对象的接口，让子类决定将哪一个类实例化。Factory Method使一个类的实例化延迟到其子类。
       11. F l y w e i g h t( 4 . 6 )：运用共享技术有效地支持大量细粒度的对象。
       12. I n t e r p r e t e r( 5 . 3 )：给定一个语言, 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。
       13. I t e r a t o r( 5 . 4 )：提供一种方法顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。
       14. M e d i a t o r( 5 . 5 )：用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。
       15. M e m e n t o( 5 . 6 )：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保存的状态。
       16. b s e r v e r( 5 . 7 )：定义对象间的一种一对多的依赖关系,以便当一个对象的状态发生改变时,所有依赖于它的对象都得到通知并自动刷新。
       17. P r o t o t y p e( 3 . 4 )：用原型实例指定创建对象的种类，并且通过拷贝这个原型来创建新的对象。
       18. P r o x y( 4 . 7 )：为其他对象提供一个代理以控制对这个对象的访问。
       19. S i n g l e t o n( 3 . 5 )：保证一个类仅有一个实例，并提供一个访问它的全局访问点。
       20. S t a t e( 5 . 8 )：允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它所属的类。
       21. S t r a t e g y (5 . 9 )：定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。本模式使得算法的变化可独立于使用它的客户。
       22. Template Method( 5 . 1 0 )：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。
       23. Template Method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。
       24. Vi s i t o r( 5 . 11 )：表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。
2. 组织编目







1. 创建型设计模式
   1. 抽象工厂Abstract Factory
      1. 提供一个创建一些列相关或相互依赖对象的接口，而无需指定他们的具体类。
      2. 适用性
         1. 系统要独立于他的产品的创建，组合和表示
         2. 系统要有多个产品系列中的一个来配置
         3. 强调一些列相关的产品对象的设计，以便联合使用
         4. 提供一个产品类库，而只想显示他们的接口而不是实现
      3. 类图



* + 1. 分工
       1. A b s t r a c t F a c t o r y ( Wi d g e t F a c t o r y )

— 声明一个创建抽象产品对象的操作接口。

* + - 1. C o n c r e t e F a c t o r y ( M o t i f Wi d g e t F a c t o r y，P M Wi d g e t F a c t o r y )

— 实现创建具体产品对象的操作。

* + - 1. A b s t r a c t P r o d u c t ( Wi n d o w s，S c r o l l B a r )

— 为一类产品对象声明一个接口。

* + - 1. C o n c r e t e P r o d u c t ( M o t i f Wi n d o w，M o t i f S c r o l l B a r )

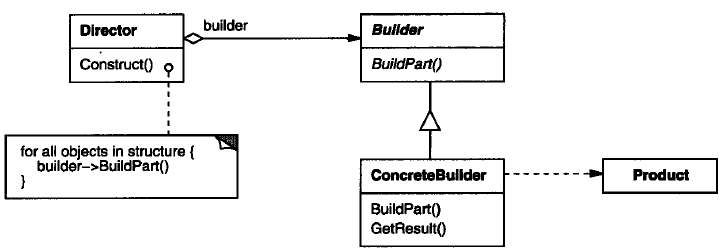
— 定义一个将被相应的具体工厂创建的产品对象。

— 实现A b s t r a c t P r o d u c t接口。

* + - 1. C l i e n t

— 仅使用由A b s t r a c t F a c t o r y和A b s t r a c t P r o d u c t类声明的接口

* 1. 生成器——Builder
     1. 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。
     2. 适用性
        1. 当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时。
        2. 当构造过程必须允许被构造的对象有不同的表示时。
     3. 类图



* + 1. 分工
       1. B u i l d e r（Te x t C o n v e r t e r）

— 为创建一个P r o d u c t对象的各个部件指定抽象接口。

* + - 1. C o n c r e t e B u i l d e r（A S C I I C o n v e r t e r、Te X C o n v e r t e r、Te x t Wi d g e t C o n v e r t e r）

— 实现B u i l d e r的接口以构造和装配该产品的各个部件。

— 定义并明确它所创建的表示。

— 提供一个检索产品的接口（例如， G e t A S C I I Te x t和G e t Te x t Wi d g e t）。

* + - 1. Director（RT F R e a d e r）

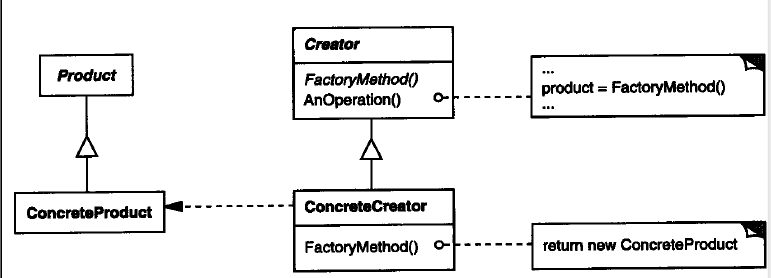
— 构造一个使用B u i l d e r接口的对象。

* + - 1. P r o d u c t（A S C I I Te x t、Te X Te x t、Te x t Wi d g e t）

— 表示被构造的复杂对象。C o n c r e t e B u i l d e r创建该产品的内部表示并定义它的装配过程。

— 包含定义组成部件的类，包括将这些部件装配成最终产品的接口。

* 1. 工厂方法——Factory Method
     1. 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。Factory Method使一个类的实例化延迟到其子类
     2. 适用性
        1. 当一个类不知道它所必须创建的对象的类的时候。
        2. 当一个类希望由它的子类来指定它所创建的对象的时候。
        3. 当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候。
     3. 类图



* + 1. 分工
       1. P r o d u c t( D o c u m e n t )

— 定义工厂方法所创建的对象的接口。

* + - 1. C o n c r e t e P r o d u c t（M y D o c u m e n t）

— 实现P r o d u c t接口。

* + - 1. C r e a t o r（A p p l i c a t i o n）

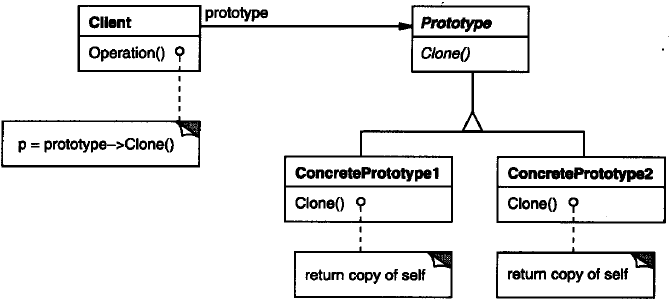
— 声明工厂方法，该方法返回一个P r o d u c t类型的对象。C r e a t o r也可以定义一个工厂方法的缺省实现，它返回一个缺省的C o n c r e t e P r o d u c t对象。

— 可以调用工厂方法以创建一个P r o d u c t对象。

* + - 1. C o n c r e t e C r e a t o r（M y A p p l i c a t i o n）

— 重定义工厂方法以返回一个C o n c r e t e P r o d u c t实例。

* 1. 原型模式——Prototype
     1. 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。
     2. 适用性
        1. 当要实例化的类是在运行时刻指定时，例如，通过动态装载；
        2. 为了避免创建一个与产品类层次平行的工厂类层次时；
        3. 当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。
     3. 类图



* + 1. 分工
       1. P r o t o t y p e（G r a p h i c）

— 声明一个克隆自身的接口。

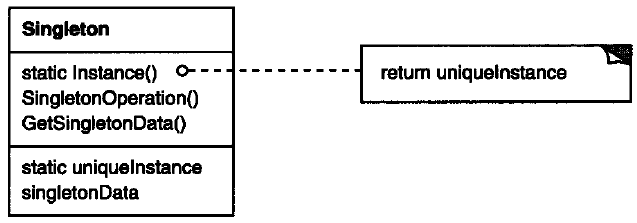
* + - 1. C o n c r e t e P r o t o t y p e（S t a ff、W h o l e N o t e、H a l f N o t e）

— 实现一个克隆自身的操作。

* + - 1. C l i e n t（G r a p h i c To o l）

— 让一个原型克隆自身从而创建一个新的对象。

* 1. 单件模式——Singleton
     1. 保证一个类仅有一个实例，并提供一个访问它的全局访问点。
     2. 适用性
        1. 当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时。
        2. 当这个唯一实例应该是通过子类化可扩展的，并且客户应该无需更改代码就能使用一个扩展的实例时。
     3. 类图



* + 1. 分工

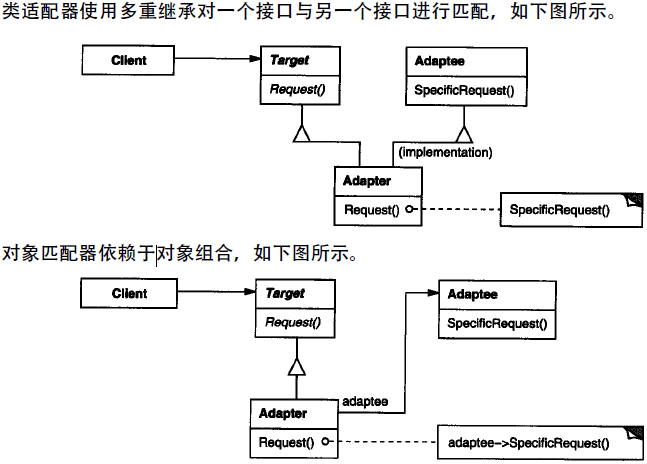
S i n g l e t o n

— 定义一个I n s t a n c e操作，允许客户访问它的唯一实例。I n s t a n c e是一个类操作（即S m a l l t a l k中的一个类方法和C + +中的一个静态成员函数）。

— 可能负责创建它自己的唯一实例。

* 1. 综合讨论

1. 结构型模式
   1. 适配器——Adapter
      1. 将一个类的接口转换成客户希望的另外一个接口。A d a p t e r模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。
      2. 适用性
         1. 你想使用一个已经存在的类，而它的接口不符合你的需求。
         2. 你想创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即那些接口可能不一定兼容的类）协同工作。
         3. （仅适用于对象A d a p t e r）你想使用一些已经存在的子类，但是不可能对每一个都进行子类化以匹配它们的接口。对象适配器可以适配它的父类接口。
      3. 类图



* + 1. 分工
       1. Ta r g e t ( S h a p e )

— 定义C l i e n t使用的与特定领域相关的接口。C l i e n t ( D r a w i n g E d i t o r )

— 与符合Ta rg e t接口的对象协同。

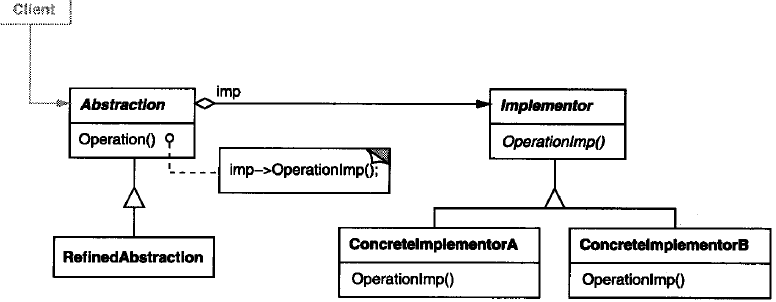
* + - 1. A d a p t e e ( Te x t Vi e w )

— 定义一个已经存在的接口，这个接口需要适配。

* + - 1. A d a p t e r ( Te x t S h a p e )

— 对A d a p t e e的接口与Ta rg e t接口进行适配

* 1. 桥接——Bridge
     1. 将抽象部分与它的实现部分分离，使它们都可以独立地变化。
     2. 适用性
        1. 你不希望在抽象和它的实现部分之间有一个固定的绑定关系。例如这种情况可能是因为，在程序运行时刻实现部分应可以被选择或者切换。
        2. 类的抽象以及它的实现都应该可以通过生成子类的方法加以扩充。这时B r i d g e模式使你可以对不同的抽象接口和实现部分进行组合，并分别对它们进行扩充。
        3. 对一个抽象的实现部分的修改应对客户不产生影响，即客户的代码不必重新编译。
        4. （C + +）你想对客户完全隐藏抽象的实现部分。在C + +中，类的表示在类接口中是可见的。
        5. 正如在意图一节的第一个类图中所示的那样，有许多类要生成。这样一种类层次结构说明你必须将一个对象分解成两个部分。R u m b a u g h称这种类层次结构为“嵌套的普化”（nested generalizations）。
        6. 你想在多个对象间共享实现（可能使用引用计数），但同时要求客户并不知道这一点。一个简单的例子便是C o p l i e n的S t r i n g类[ C o p 9 2 ]，在这个类中多个对象可以共享同一个字符串表示（ S t r i n g R e p）。
     3. 类图



* + 1. 分工
       1. Abstraction (Window)

— 定义抽象类的接口。

— 维护一个指向I m p l e m e n t o r类型对象的指针。

* + - 1. RefinedAbstraction (IconWi n d o w )

— 扩充由A b s t r a c t i o n定义的接口。

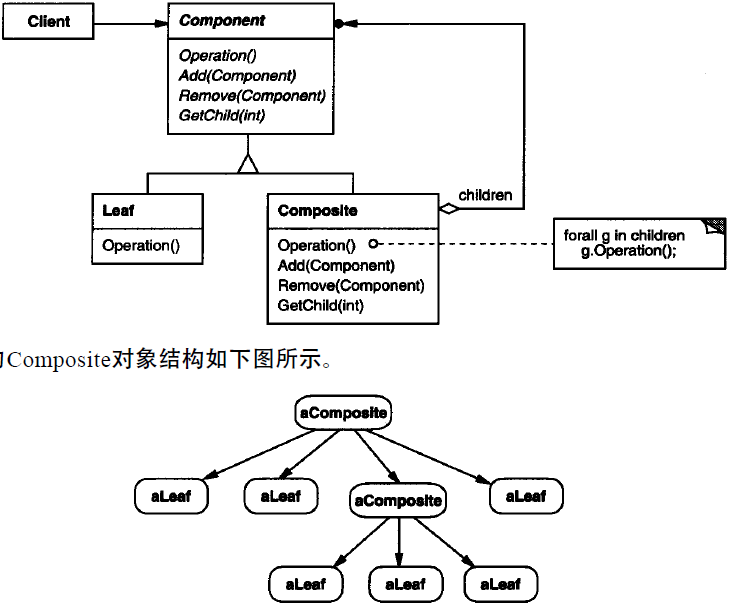
* + - 1. Implementor (Wi n d o w I m p )

— 定义实现类的接口，该接口不一定要与A b s t r a c t i o n的接口完全一致；事实上这两个接口可以完全不同。一般来讲， I m p l e m e n t o r接口仅提供基本操作，而A b s t r a c t i o n则定义了基于这些基本操作的较高层次的操作。

* + - 1. ConcreteImplementor (XwindowImp, PMWi n d o w I m p )

— 实现I m p l e m e n t o r接口并定义它的具体实现。

* 1. 组合——Composite
     1. 将对象组合成树形结构以表示“部分-整体”的层次结构。C o m p o s i t e使得用户对单个对象和组合对象的使用具有一致性。
     2. 适用性
        1. 你想表示对象的部分-整体层次结构。
        2. 你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。
     3. 类图



* + 1. 分工
       1. Component (Graphic)

— 为组合中的对象声明接口。

— 在适当的情况下，实现所有类共有接口的缺省行为。

— 声明一个接口用于访问和管理C o m p o n e n t的子组件。

—(可选)在递归结构中定义一个接口，用于访问一个父部件，并在合适的情况下实现它。

* + - 1. Leaf (Rectangle、L i n e、Te x t等)

— 在组合中表示叶节点对象，叶节点没有子节点。

— 在组合中定义图元对象的行为。

* + - 1. Composite (Picture)

— 定义有子部件的那些部件的行为。

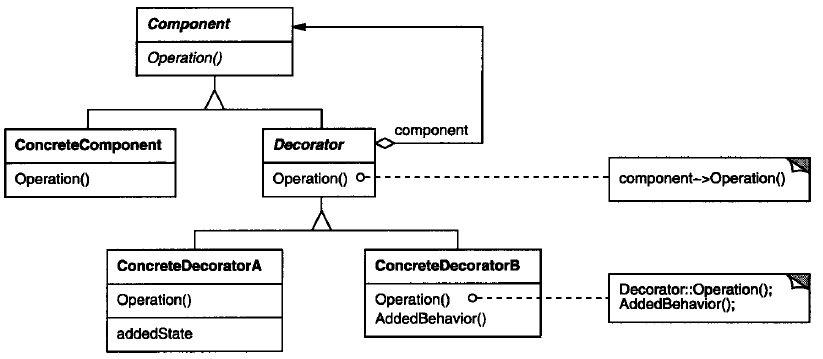
— 存储子部件。

— 在C o m p o n e n t接口中实现与子部件有关的操作。

* + - 1. Client

— 通过C o m p o n e n t接口操纵组合部件的对象。

* 1. 装饰——Decorator
     1. 动态地给一个对象添加一些额外的职责。就增加功能来说， D e c o r a t o r模式相比生成子类更为灵活。
     2. 适用性
        1. 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。
        2. 处理那些可以撤消的职责。
        3. 当不能采用生成子类的方法进行扩充时。一种情况是，可能有大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长。另一种情况可能是因为类定义被隐藏，或类定义不能用于生成子类。
     3. 类图



* + 1. 分工
       1. Component ( Vi s u a l C o m p o n e n t )

— 定义一个对象接口，可以给这些对象动态地添加职责。

* + - 1. C o n c r e t e C o m p o n e n t ( Te x t Vi e w )

— 定义一个对象，可以给这个对象添加一些职责。

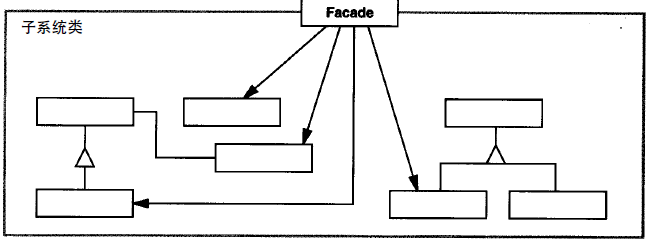
* + - 1. D e c o r a t o r

— 维持一个指向C o m p o n e n t对象的指针，并定义一个与C o m p o n e n t接口一致的接口。

* + - 1. C o n c r e t e D e c o r a t o r ( B o r d e r D e c o r a t o r, ScrollDecorator)

— 向组件添加职责。

* 1. 外观——Facade
     1. 为子系统中的一组接口提供一个一致的界面， F a c a d e模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。
     2. 适用性
        1. 当你要为一个复杂子系统提供一个简单接口时。子系统往往因为不断演化而变得越来越复杂。大多数模式使用时都会产生更多更小的类。这使得子系统更具可重用性，也更容易对子系统进行定制，但这也给那些不需要定制子系统的用户带来一些使用上的困难。F a c a d e可以提供一个简单的缺省视图，这一视图对大多数用户来说已经足够，而那些需要更多的可定制性的用户可以越过f a c a d e层。
        2. 客户程序与抽象类的实现部分之间存在着很大的依赖性。引入f a c a d e将这个子系统与客户以及其他的子系统分离，可以提高子系统的独立性和可移植性。
        3. 当你需要构建一个层次结构的子系统时，使用f a c a d e模式定义子系统中每层的入口点。如果子系统之间是相互依赖的，你可以让它们仅通过f a c a d e进行通讯，从而简化了它们之间的依赖关系。
     3. 类图



* + 1. 分工
       1. F a c a d e ( C o m p i l e r )

— 知道哪些子系统类负责处理请求。

— 将客户的请求代理给适当的子系统对象。

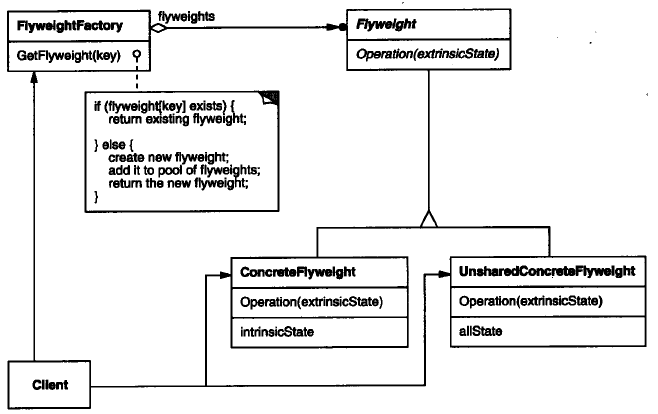
* + - 1. Subsystem classes ( S c a n n e r、P a r s e r、P r o g r a m N o d e等)

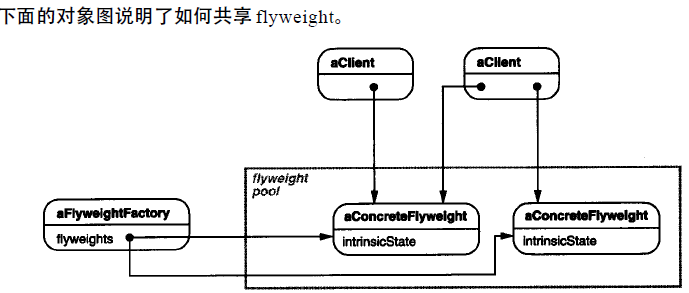
— 实现子系统的功能。

— 处理由F a c a d e对象指派的任务。

— 没有f a c a d e的任何相关信息；即没有指向f a c a d e的指针。

* 1. 蝇量、享元——FlyWeight——128
     1. 运用共享技术有效地支持大量细粒度的对象。
     2. 适用性
        1. 一个应用程序使用了大量的对象。
        2. 完全由于使用大量的对象，造成很大的存储开销。
        3. 对象的大多数状态都可变为外部状态。
        4. 如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象。
        5. 应用程序不依赖于对象标识。由于F l y w e i g h t对象可以被共享，对于概念上明显有别的对象，标识测试将返回真值。
     3. 类图





* + 1. 分工
       1. F l y w e i g h t ( G l y p h )

— 描述一个接口，通过这个接口f l y w e i g h t可以接受并作用于外部状态。

* + - 1. C o n c r e t e F l y w e i g h t( C h a r a c t e r )

— 实现F l y w e i g h t 接口， 并为内部状态（ 如果有的话） 增加存储空间。C o n c r e t e F l y w e i g h t对象必须是可共享的。它所存储的状态必须是内部的；即，它必须独立于C o n c r e t e F l y w e i g h t对象的场景。

* + - 1. U n s h a r e d C o n c r e t e F l y w e i g h t ( R o w, C o l u m n )

— 并非所有的F l y w e i g h t子类都需要被共享。F l y w e i g h t接口使共享成为可能，但它并不强制共享。在F l y w e i g h t对象结构的某些层次， U n s h a r e d C o n c r e t e F l y w e i g h t对象通常将C o n c r e t e F l y w e i g h t对象作为子节点（R o w和C o l u m n就是这样）。

* + - 1. F l y w e i g h t F a c t o r y

— 创建并管理f l y w e i g h t对象。

— 确保合理地共享f l y w e i g h t。当用户请求一个f l y w e i g h t时，F l y w e i g h t F a c t o r y对象提供

一个已创建的实例或者创建一个（如果不存在的话）。

* + - 1. Client

— 维持一个对f l y w e i g h t的引用。

— 计算或存储一个（多个） f l y w e i g h t的外部状态。

* 1. 代理——Proxy——137
     1. 为其他对象提供一种代理以控制对这个对象的访问和操作。
     2. 适用性
        1. 远程代理（ Remote Proxy ）

为一个对象在不同的地址空间提供局部代表。使用N X P r o x y类实现了这一目的。

* + - 1. 虚代理（Virtual Proxy）

根据需要创建开销很大的对象。在动机一节描述的I m a g e P r o x y就是这样一种代理的例子。

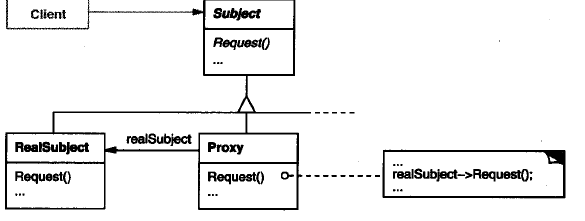
* + - 1. 保护代理（Protection Proxy）

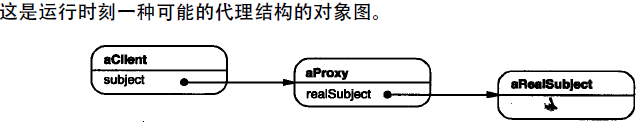
控制对原始对象的访问。保护代理用于对象应该有不同的访问权限的时候。

* + - 1. 智能指引（Smart Reference）

取代了简单的指针，它在访问对象时执行一些附加操作。它的典型用途包括：

* 对指向实际对象的引用计数，这样当该对象没有引用时，可以自动释放它)。
* 当第一次引用一个持久对象时，将它装入内存。
* 在访问一个实际对象前，检查是否已经锁定了它，以确保其他对象不能改变它。
  + 1. 类图





* + 1. 分工
       1. • P r o x y ( I m a g e P r o x y )

— 保存一个引用使得代理可以访问实体。若R e a l S u b j e c t和S u b j e c t的接口相同，P r o x y会引用S u b j e c t。

— 提供一个与S u b j e c t的接口相同的接口，这样代理就可以用来替代实体。

— 控制对实体的存取，并可能负责创建和删除它。

— 其他功能依赖于代理的类型：

* + - 1. • Remote Pro x y负责对请求及其参数进行编码，并向不同地址空间中的实体发送已编码的请求。
      2. • Vi rtual Pro x y可以缓存实体的附加信息，以便延迟对它的访问。例如，动机一节中提到的I m a g e P r o x y缓存了图像实体的尺寸。
      3. • P rotection Pro x y检查调用者是否具有实现一个请求所必需的访问权限。
      4. • S u b j e c t ( G r a p h i c )

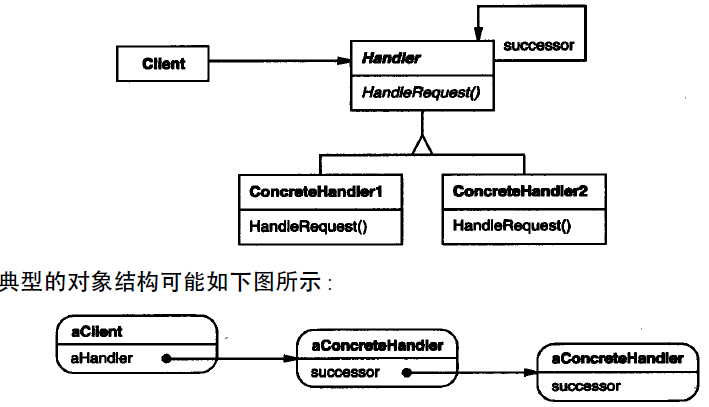
— 定义RealSubject 和P r o x y的共用接口，这样就在任何使用R e a l S u b j e c t的地方都可以使用P r o x y。

* + - 1. • R e a l S u b j e c t ( I m a g e )

— 定义P r o x y所代表的实体。

* 1. 综合讨论——144

1. 行为模式
   1. 职责链——Chain of Responsibility
      1. 使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。
      2. 适用性
         1. 有多个的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。
         2. 你想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。
         3. 可处理一个请求的对象集合应被动态指定。
      3. 类图



* + 1. 分工
       1. H a n d l e r（如H e l p H a n d l e r）

— 定义一个处理请求的接口。

— （可选） 实现后继链。

* + - 1. C o n c r e t e H a n d l e r（如P r i n t B u t t o n和P r i n t D i a l o g）

— 处理它所负责的请求。

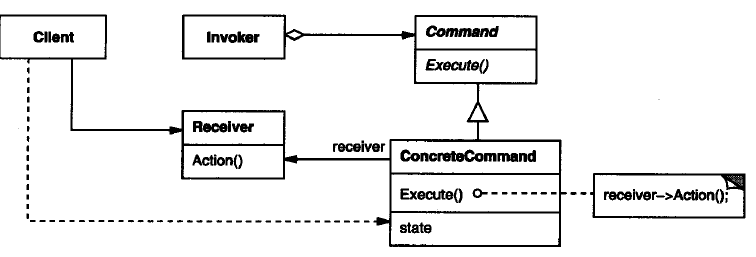
— 可访问它的后继者。

— 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。

* + - 1. C l i e n t

— 向链上的具体处理者( C o n c r e t e H a n d l e r )对象提交请求。

* 1. 命令——Command
     1. 将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤消的操作。
     2. 适用性
        1. 像上面讨论的M e n u I t e m对象那样，抽象出待执行的动作以参数化某对象。你可用过程语言中的回调（c a l l b a c k）函数表达这种参数化机制。所谓回调函数是指函数先在某处注册，而它将在稍后某个需要的时候被调用。C o m m a n d模式是回调机制的一个面向对象的替代品。
        2. 在不同的时刻指定、排列和执行请求。一个C o m m a n d对象可以有一个与初始请求无关的生存期。如果一个请求的接收者可用一种与地址空间无关的方式表达，那么就可将负责该请求的命令对象传送给另一个不同的进程并在那儿实现该请求。
        3. 支持取消操作。C o m m a n d的E x c u t e操作可在实施操作前将状态存储起来，在取消操作时这个状态用来消除该操作的影响。C o m m a n d接口必须添加一个U n e x e c u t e操作，该操作取消上一次
        4. E x e c u t e调用的效果。执行的命令被存储在一个历史列表中。可通过向后和向前遍历这一列表并分别调用U n e x e c u t e和E x e c u t e来实现重数不限的“取消”和“重做”。
        5. 支持修改日志，这样当系统崩溃时，这些修改可以被重做一遍。在C o m m a n d接口中添加装载操作和存储操作，
     3. 类图



* + 1. 分工
       1. C o m m a n d

— 声明执行操作的接口。

* + - 1. C o n c r e t e C o m m a n d ( P a s t e C o m m a n d，O p e n C o m m a n d )

— 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E x e c u t e。

* + - 1. C l i e n t ( A p p l i c t i o n )

— 创建一个具体命令对象并设定它的接收者。

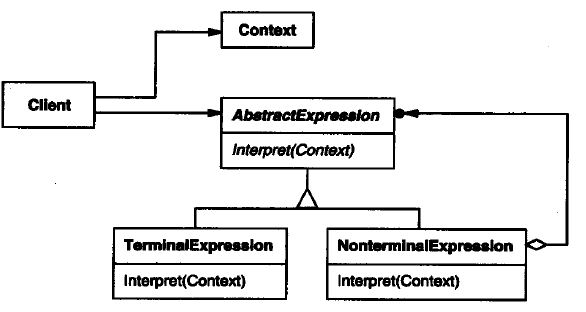
* + - 1. Invoker ( M e n u I t e m )

— 要求该命令执行这个请求。

* + - 1. R e c e i v e r ( D o c u m e n t，A p p l i c a t i o n )

— 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。

* 1. 解释器——Interpreter
     1. 给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。
     2. 适用性
        1. 当有一个语言需要解释执行, 并且你可将该语言中的句子表示为一个抽象语法树时，可使用解释器模式。而当存在以下情况时该模式效果最好：
        2. 该文法简单对于复杂的文法, 文法的类层次变得庞大而无法管理。此时语法分析程序生成器这样的工具是更好的选择。它们无需构建抽象语法树即可解释表达式, 这样可以节省空间而且还可能节省时间。
        3. 效率不是一个关键问题最高效的解释器通常不是通过直接解释语法分析树实现的, 而是首先将它们转换成另一种形式。例如，正则表达式通常被转换成状态机。但即使在这种情况下, 转换器仍可用解释器模式实现, 该模式仍是有用的。
     3. 类图



* + 1. 分工
       1. a l E x p r e s s i o n (终结符表达式，如L i t e r a l E x p r e s s i o n )

— 实现与文法中的终结符相关联的解释操作。

— 一个句子中的每个终结符需要该类的一个实例。

* + - 1. N o n t e r m i n a l E x p r e s s i o n (非终结符表达式，如AlternationExpression, epetition-Expression, SequenceExpressions)

— 对文法中的每一条规则R ::= R1R2. . . Rn都需要一个N o n t e r m i n a l E x p r e s s i o n类。

— 为从R1到Rn的每个符号都维护一个A b s t r a c t E x p r e s s i o n类型的实例变量。

— 为文法中的非终结符实现解释( I n t e r p r e t )操作。解释( I n t e r p r e t )一般要递归地调用表示R1到Rn的那些对象的解释操作。

* + - 1. C o n t e x t（上下文）

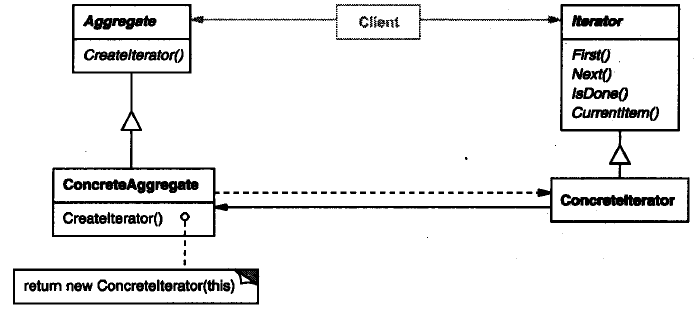
— 包含解释器之外的一些全局信息。

* + - 1. C l i e n t（客户）

— 构建(或被给定) 表示该文法定义的语言中一个特定的句子的抽象语法树。该抽象语法树由N o n t e r m i n a l E x p r e s s i o n和Te r m i n a l E x p r e s s i o n的实例装配而成。

— 调用解释操作。

* 1. 迭代器——Iterator
     1. 提供一种方法顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。
     2. 适用性
        1. 访问一个聚合对象的内容而无需暴露它的内部表示。
        2. 支持对聚合对象的多种遍历。
        3. 为遍历不同的聚合结构提供一个统一的接口(即, 支持多态迭代)。
     3. 类图



* + 1. 分工
       1. I t e r a t o r（迭代器）

— 迭代器定义访问和遍历元素的接口。

* + - 1. C o n c r e t e I t e r a t o r（具体迭代器）

— 具体迭代器实现迭代器接口。

— 对该聚合遍历时跟踪当前位置。

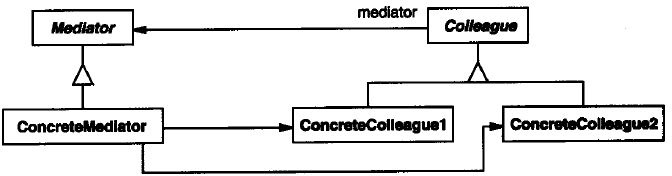
* + - 1. A g g r e g a t e（聚合）

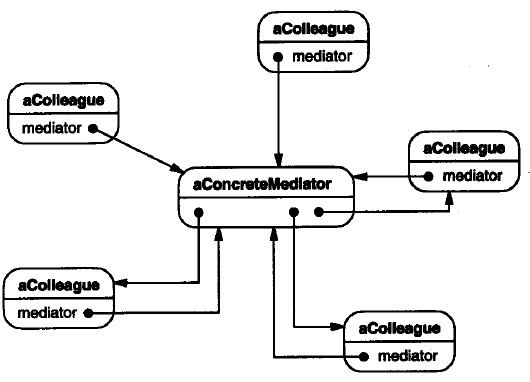
— 聚合定义创建相应迭代器对象的接口。

* + - 1. ConcreteAggregate（具体聚合）

— 具体聚合实现创建相应迭代器的接口，该操作返回C o n creteIterator的一个适当的实例。

* 1. 中介者——Mediator
     1. 用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。
     2. 适用性
        1. 一组对象以定义良好但是复杂的方式进行通信。产生的相互依赖关系结构混乱且难以理解。
        2. 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。
        3. 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。
     3. 类图





* + 1. 分工
       1. M e d i a t o r(中介者，如D i a l o g D i r e c t o r )

— 中介者定义一个接口用于与各同事（ C o l l e a g u e）对象通信。

* + - 1. C o n c r e t e M e d i a t o r(具体中介者，如F o n t D i a l o g D i r e c t o r )

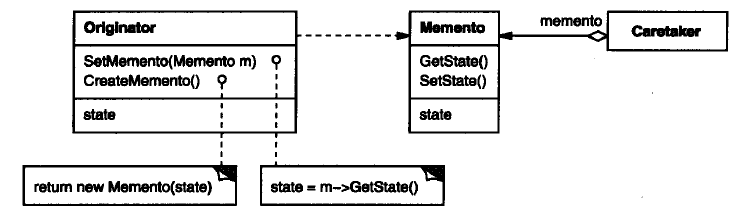
— 具体中介者通过协调各同事对象实现协作行为。

— 了解并维护它的各个同事。

* + - 1. Colleague class(同事类，如ListBox, EntryField)

— 每一个同事类都知道它的

* 1. 备忘录——Memento
     1. 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。
     2. 适用性
        1. 必须保存一个对象在某一个时刻的(部分)状态, 这样以后需要时它才能恢复到先前的状态。
        2. 如果一个用接口来让其它对象直接得到这些状态，将会暴露对象的实现细节并破坏对象的封装性。
     3. 类图



* + 1. 分工
       1. M e m e n t o(备忘录，如S o l v e r S t a t e )

— 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( c a r e t a k e r )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。

* + - 1. O r i g i n a t o r(原发器，如C o n s t r a i n t S o l v e r )

— 原发器创建一个备忘录,用以记录当前时刻它的内部状态。

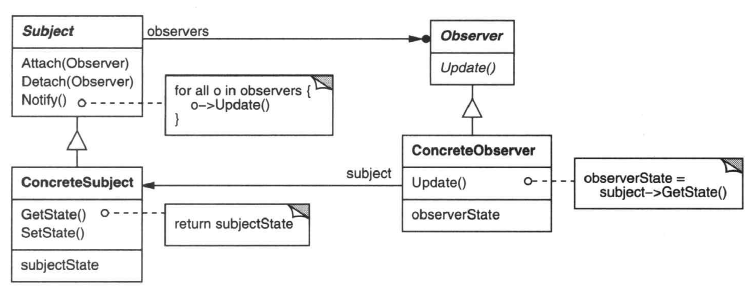
— 使用备忘录恢复内部状态.。

* + - 1. C a r e t a k e r(负责人，如undo mechanism)

— 负责保存好备忘录。

— 不能对备忘录的内容进行操作或检查。

* 1. 观察者——Observer
     1. 定义对象间的一种一对多的依赖关系,当一个对象的状态发生改变时, 所有依赖于它的对象都得到通知并被自动更新。
     2. 适用性
        1. 当一个抽象模型有两个方面, 其中一个方面依赖于另一方面。将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。
        2. 当对一个对象的改变需要同时改变其它对象, 而不知道具体有多少对象有待改变。
        3. 当一个对象必须通知其它对象，而它又不能假定其它对象是谁。换言之, 你不希望这些对象是紧密耦合的。
     3. 类图



* + 1. 分工
       1. S u b j e c t（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。

* + - 1. O b s e r v e r（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。

* + - 1. C o n c r e t e S u b j e c t（具体目标）

— 将有关状态存入各C o n c r e t e O b s e r v e r对象。

— 当它的状态发生改变时, 向它的各个观察者发出通知。

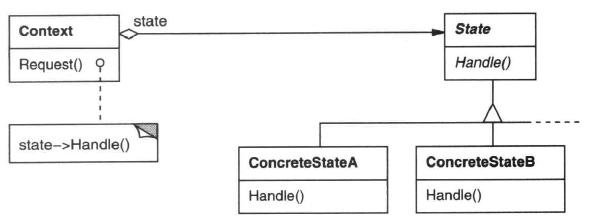
* + - 1. C o n c r e t e O b s e r v e r（具体观察者）

— 维护一个指向C o n c r e t e S u b j e c t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O b s e r v e r的更新接口以使自身状态与目标的状态保持一致。

* 1. 状态——State
     1. 允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。
     2. 适用性
        1. 一个对象的行为取决于它的状态, 并且它必须在运行时刻根据状态改变它的行为。
        2. 一个操作中含有庞大的多分支的条件语句，且这些分支依赖于该对象的状态。这个状态通常用一个或多个枚举常量表示。通常, 有多个操作包含这一相同的条件结构。S t a t e模式将每一个条件分支放入一个独立的类中。这使得你可以根据对象自身的情况将对象的状态作为一个对象，这一对象可以不依赖于其他对象而独立变化。
     3. 类图



* + 1. 分工
       1. C o n t e x t(环境，如T C P C o n n e c t i o n )

— 定义客户感兴趣的接口。

— 维护一个C o n c r e t e S t a t e子类的实例，这个实例定义当前状态。

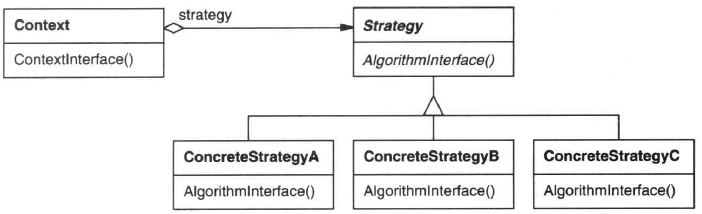
* + - 1. S t a t e(状态，如T C P S t a t e )

— 定义一个接口以封装与C o n t e x t的一个特定状态相关的行为。

* + - 1. ConcreteState subclasses(具体状态子类，如TCPEstablished, TCPListen, TCPClosed)

— 每一子类实现一个与C o n t e x t的一个状态相关的行为。

* 1. 策略——Strategy——208
     1. 定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。
     2. 适用性
        1. 许多相关的类仅仅是行为有异。“策略”提供了一种用多个行为中的一个行为来配置一个类的方法。
        2. 需要使用一个算法的不同变体。例如，你可能会定义一些反映不同的空间/时间权衡的算法。当这些变体实现为一个算法的类层次时,可以使用策略模式。
        3. 算法使用客户不应该知道的数据。可使用策略模式以避免暴露复杂的、与算法相关的数据结构。
        4. 一个类定义了多种行为, 并且这些行为在这个类的操作中以多个条件语句的形式出现。将相关的条件分支移入它们各自的S t r a t e g y类中以代替这些条件语句。
     3. 类图



* + 1. 分工
       1. S t r a t e g y(策略，如C o m p o s i t o r )

— 定义所有支持的算法的公共接口。C o n t e x t使用这个接口来调用某C o n c r e t e S t r a t e g y定义的算法。

* + - 1. C o n c r e t e S t r a t e g y(具体策略，如S i m p l e C o m p o s i t o r, Te X C o m p o s i t o r, ArrayCompositor)

— 以S t r a t e g y接口实现某具体算法。

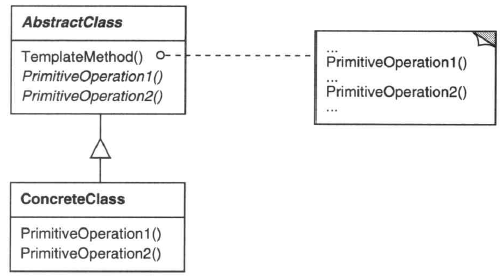
* + - 1. C o n t e x t(上下文，如C o m p o s i t i o n )

— 用一个C o n c r e t e S t r a t e g y对象来配置。

— 维护一个对S t r a t e g y对象的引用。

— 可定义一个接口来让S t a t e g y访问它的数据。

* 1. 模板方法——Template Method
     1. 定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。Te m p l a t e M e t h o d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。
     2. 适用性
        1. 一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。
        2. 各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。首先识别现有代码中的不同之处，并且将不同之处分离为新的操作。最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。
        3. 控制子类扩展。模板方法只在特定点调用“ h o o k”操作（参见效果一节），这样就只允许在这些点进行扩展。
     3. 类图



* + 1. 分工
       1. A b s t r a c t C l a s s（抽象类，如A p p l i c a t i o n）

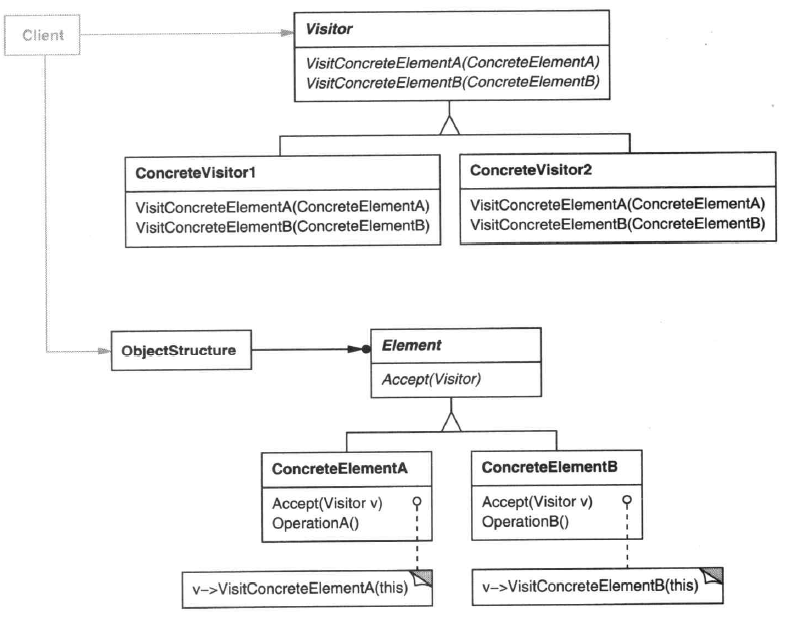
— 定义抽象的原语操作（ primitive operation），具体的子类将重定义它们以实现一个算法的各步骤。

— 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A b s t r a c t C l a s s或其他对象中的操作。

* + - 1. C o n c r e t e C l a s s（具体类，如M y A p p l i c a t i o n）

— 实现原语操作以完成算法中与特定子类相关的步骤。

* 1. 访问者——Visitor——218
     1. 表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。
     2. 适用性
        1. 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。
        2. 需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而你想避免让这些操作“污染”这些对象的类。Vi s i t o r使得你可以将相关的操作集中起来定义在一个类中。当该对象结构被很多应用共享时，用Vi s i t o r模式让每个应用仅包含需要用到的操作。
        3. 定义对象结构的类很少改变，但经常需要在此结构上定义新的操作。改变对象结构类需要重定义对所有访问者的接口，这可能需要很大的代价。如果对象结构类经常改变，那么可能还是在这些类中定义这些操作较好。
     3. 类图



* + 1. 分工
       1. Vi s i t o r（访问者，如N o d e Vi s i t o r）

— 为该对象结构中C o n c r e t e E l e m e n t的每一个类声明一个Vi s i t操作。该操作的名字和特征标识了发送Vi s i t请求给该访问者的那个类。这使得访问者可以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。

* + - 1. C o n c r e t e Vi s i t o r（具体访问者，如Ty p e C h e c k i n g Vi s i t o r）

— 实现每个由Vi s i t o r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。C o n c r e t e Vi s i t o r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。

* + - 1. E l e m e n t（元素，如N o d e）

— 定义一个A c c e p t操作，它以一个访问者为参数。

* + - 1. C o n c r e t e E l e m e n t（具体元素，如A s s i g n m e n t N o d e，Va r i a b l e R e f N o d e）

— 实现A c c e p t操作，该操作以一个访问者为参数。

* + - 1. O b j e c t S t r u c t u r e（对象结构，如P r o g r a m）

— 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（参见C o m p o s i t e（4 . 3））或是一个集合，如一个列表或一个无序集合。

* + 1. 行为模式综合讨论

1. **词汇表**
   * 抽象类（abstract class） 一种主要用来定义接口的类。抽象类中的部分或全部操作被延迟到其子类中实现。抽象类不能实例化。
   * 抽象耦合（abstract coupling） 若类A维护一个指向抽象类B的引用，则称类A抽象耦合于B。我们之所以称之为抽象耦合乃是因为A指向的是一个对象的类型，而不是一个具体对象。
   * 抽象操作（abstract operation） 一种声明了型构（ s i g n a t u r e）而没有实现的操作。在C + +中，抽象操作对应于纯虚成员函数。
   * 相识关系（acquaintance relationship） 如果一个类指向另一个类，则这两个类之间有相识关系。
   * 聚合对象（aggregate object） 一种包含子对象的对象。这些子对象称为聚合对象的部分，而聚合对象对它们负责。
   * 聚合关系（aggregation relationship） 聚合对象与其部分之间的关系。类为其对象（例如，聚合对象）定义这种关系。
   * 黑箱复用（black-box reuse） 一种基于对象组合的复用方式。这些被组合的对象之间并不开放各自的内部细节，因此被比作“黑箱”。
   * 类（c l a s s） 类定义对象的接口和实现。它规定对象的内部表示，定义对象可实施的操作。
   * 类图（class diagram） 类图描述类及其内部结构和操作，以及类间的静态关系。
   * 类操作（class operation） 以类而不是单独的对象为目标的操作。在C + +中，类操作称为静态成员函数。
   * 具体类（concrete class） 不含抽象操作的类。它可以实例化。
   * 构造器（c o n s t r u c t o r） 在C + +中，一种系统自动调用的用来初始化新对象实例的操作。
   * 耦合（c o u p l i n g） 软件构件之间相互依赖的程度。
   * 委托（d e l e g a t i o n） 一种实现机制，即一个对象把发给它的请求转发/委托给另一个对象。而受托对象将代表原对象执行请求的操作。
   * 设计模式（design pattern） 设计模式针对面相对象系统中重复出现的设计问题，提出一个通用的设计方案，并予以系统化的命名和动机解释。它描述了问题、解决方案、在什么条件下使用该解决方案及其效果。它还给出了实现要点和实例。该解决方案是解决该问题的一组精心安排的通用的类和对象，再经定制和实现就可用来解决特定上下文中的问题。
   * 析构器（d e s t r u c t o r） 在C + +中，一种系统自动调用的用来清理（ f i n a l i z e）即将被删除的对象的操作。
   * 动态绑定（dynamic binding） 在运行时刻才将一个请求与一个对象及其一个操作关联起来。在C + +中，只有虚函数可动态绑定。
   * 封装（e n c a p s u l a t i o n） 其结果是将对象的表示和实现隐藏起来。在对象之外，看不到其内部表示，也不能直接对其进行访问。操作（ o p e r a t i o n）是访问和修改对象表示的唯一途径。
   * 框架（f r a m e w o r k） 一组相互协作的类，形成某类软件的一个可复用设计。框架将设计划分为一组抽象类，并定义它们各自的责任及相互之间的合作，以此来指导体系结构级的设计。开发者通过继承框架中的类和组合其实例来定制该框架以生成特定的应用。
   * 友类（friend class） 在C + +中，A为B的友类是指A对B中的操作和数据有与B本身一样的访问权限。
   * 继承（i n h e r i t a n c e） 两个实体间的一种关系，其中一实体乃是基于另一实体而定义的。类继承以一个或多个父类为基础定义一个新类，这个新类继承了其父类的接口和实现，被称为子类（ C + +）或派生类。类继承包含了接口继承和实现继承。接口继承以一个或多个已有接口为基础定义新的接口；实现继承以一个或多个已有实现为基础定义新的实现。
   * 实例变量（instance variable） 定义部分对象表示的数据。C + +中使用的术语是数据成员。
   * 交互图（interaction diagram） 展示对象间请求流程的一种示意图。
   * 接口（i n t e r f a c e） 一个对象所有操作定义的型构的集合。接口刻划了一个对象可响应的请求的集合。
   * 元类（m e t a c l a s s） 在S m a l l t a l k中，类也是对象。元类是类对象的类。
   * 混入类（mixin class） 一种被设计为通过继承与其他类结合的类。混入类通常是抽象类。
   * 对象（o b j e c t） 一个封装了数据及作用于这些数据的操作的运行实体。
   * 对象组合（object composition） 组装和组合一组对象以获得更复杂的行为。
   * 对象图（object diagram） 描述运行时刻特定对象结构的示意图。
   * 对象引用（object reference） 用于标识另一对象的一个值。
   * 操作（o p e r a t i o n） 对象的数据仅能由其自身的操作来存取。对象受到请求时执行操作。在C + +中，操作称为成员函数，而S m a l l t a l k使用术语“ 方法”。
   * 重定义（o v e r r i d i n g） 在一个子类中重定义（从父类继承下来的）操作。
   * 参数化类型（parameterized type） 一种含有未确定成分类型的类型。在使用时，将未确定类型处理成参数。在C + +中，参数化类型称为模板（t e m p l a t e）。
   * 父类（parent class） 被其他类继承的类。S m a l l t a l k又称之为超类（s u p e r c l a s s），C + +中又称之为基类（base class），有时又称为祖先类（ancestor class）。
   * 多态（p o l y m o r p h i s m） 在运行时刻接口匹配的对象能互相替换的能力。
   * 私有继承（private inheritance） 在C + +中，一种仅出于实现目的的继承。
   * 协议（p r o t o c o l） 接口概念的扩展，包含指明可允许的请求序列。
   * 接收者（r e c e i v e r） 一个请求的目标对象。
   * 请求（r e q u e s t） 一个对象当受到其他对象的请求时执行相应的操作。通常请求又称为消息。
   * 型构（s i g n a t u r e） 一个操作的型构定义了它的名称、参数和返回值。
   * 子类（s u b c l a s s） 继承了另一个类的类。在C + +中，子类又称为派生类（derived class）。
   * 子系统（s u b s y s t e m） 一组相互协作的类形成的一个相对独立的部分，完成一定的功能。
   * 子类型（s u b t y p e） 如果一个类型的接口包含另一类型的接口，则前一类型称为后一类型的子类型。
   * 超类型（s u p e r t y p e） 为其他类型继承的父类型。
   * 工具箱（t o o l k i t） 一组提供实用功能的类，但它们并不包含任何具体应用的设计。
   * 类型（t y p e） 一个特定接口的名称。
   * 白箱复用（white-box reuse） 一种基于类继承的复用。子类复用父类的接口和实现，但它也可能存取其父类的其他私有部分。