**GRASP：是General Responsibility Assignment Software Patterns的缩写，这些模式不是设计模式，而是对象设计的基本原则，关注对象设计最重要的方面——分配职责给类，并不强调体系结构的设计；**

|  |  |
| --- | --- |
| 模式名称 | 问题，解决方案，优缺点 |
| 信息专家模式 | 问题：对象设计和职责分配的一般原则是什么？ 解决方案：将职责分配给拥有履行一个职责所必需信息的类－－即信息专家。（也就是将职责分配给一个类，这个类必须拥有履行这个职责所需要的信息。）  优点：维护信息影藏，支持高内聚低耦合，  缺点：让类变得复杂 |
| 创建者模式 | 问题：谁应该负责产生类的实例（对应于GoF设计模式系列里的“工厂模式”）  解决方案：如果符合下面的一个或多个条件，则将创建类A实例的职责分配给类B.  .类B聚合类A的对象。（prefer）  .类B包含类A的对象。（prefer）  .类B记录类A对象的实例。  .类B密切使用类A的对象。  .类B初始化数据并在创建类A的实例时传递给类A（类B是创建类A实例的一个专家)。  在以上情况下，类B是类A对象的创建者。  优点：  支持低耦合：将创建实例的职责分配给需要对象引用的类  降低依赖：通过自己创建对象避免了依赖其它类帮他们创建对象 |
| 控制器模式 | 问题：谁处理一个系统事件？  解决方案：当类代表下列一种情况时，为它分配处理系统事件消息的职责。  .代表整个系统、设备或子系统（外观控制器）。  .代表系统事件发生的用例场景（用例或回话控制器）。  专门设计一个类处理事件：  1（功能）针对业务 或overall organization(a façade controller).（集中式）  2（系统）针对系统 (a façade controller).（集中式）  3（角色）针对模块 （a role controller）（近似集中式）  4（用例）针对一个用例 (a use case controller)（近似分散式）  优点：  使外部事件源和内部事件处理器不相互依赖对方的类型和行为  缺点： |
| 低耦合 | 问题：如何支持低依赖性以及增加重用性？  解决方案：分配职责时使（不必要的）耦合保持为最低。  OO中耦合的种类：  Y是X的属性  X的方法中有Y（参数，局部变量，返回值）  X是Y的子孙类  X实现Y接口  优点：类更易维护，易复用，将change本地化 |
| 高内聚 | 问题：如何让复杂性可管理？  解决方案：分配职责时使内聚保持为最高。  内聚的不同程度：  非常低内聚:一个类的职责包括很多功能  低内聚: 一个类职责包含一个功能中的复杂的任务  高内聚. 一个类在一个功能上有适中的职责，和其他类协作完成任务。  优点：类易维护，易理解，支持低耦合，支持复用 |
| 多态模式 | 问题：当行为随类型变化而变化时谁来负责处理这些变化？  解决方案：当类型变化导致另一个行为或导致行为变化时，应用多态操作将行为的职责分配到引起行为变化的类型。  优点：  更简单可靠，对比复杂的选择逻辑（判断语句）  更易添加额外的行为  在设计中增加类的数量  使代码更易理解 |
| 纯虚构模式 | 问题：当不想破坏高内聚和低耦合的设计原则时，谁来负责处理这些变化？  解决方案：将一组高内聚的职责分配给一个虚构的或处理方便的“行为”类，它并不是问题域中的概念，而是虚构的事务，以达到支持高内聚、低耦合和重用的目的。  典型适用场景：  将representation 从 model中分离出去  将platforms(facilities) 从 model中分离出去  分离复杂的行为  分离复杂的数据结构  优点：  支持高内聚：相关职责被封装成一个类来处理一系列相关任务。  支持复用because of the presence of fine grained Pure Fabrication classes. |
| 间接性 | 问题：如何分配职责以避免直接耦合？  解决方案：分配职责给中间对象以协调组件或服务之间的操作，使得它们不直接耦合。  优点：与可变性低耦合，支持复用 |
| 受保护变化模式 | 问题：如何分配职责给对象、子系统和系统，使得这些元素中的变化或不稳定的点不会对其他元素产生不利影响？  解决方案：找出预计有变化或不稳定的元素，为其创建稳定的“接口”而分配职责。  类型：  Information Hiding  Data driven (configuration files)  Service lookup (runtime registration)  Interpreter-Driven(generalize module)  Reflective or Meta-Level Designs (Component replace)  Uniform Access (adherence to protocols)  LSP (polymorphism)  Law of Demeter (restrict communication paths) |

# 专家模式（expert）

**最基本的问题——谁负责得知某些信息；解决方法——将这个职责分配给需要信息来完成自己职责的类；维护信息封装，低耦合高内聚，可能使类变得很复杂**

谁都负责自己的信息的统计工作，也就是，你只负责你本身拥有的信息的处理，只有对自己了解的信息，你才是专家，才能拥有职责。

Sale 负责总价钱

SalesLineItem 负责每一个Line的总价钱

ProductSpecification 负责每一个产品的价格

1) 低耦合：分配职责时保证低耦合，即降低依赖并增加复用性

2) 高内聚：分配职责后保证模块内部的内聚性很高，将复杂度控制在可管理的范围之内。不可能完全消除时序内聚、过程内聚、通信内聚，必须将这些内聚的拥有者变成转发者。一个对象仅实现单一的职责，而不实现复杂的职责组合

3) 专家模式：（促进高内聚）解决面向对象设计中的职责分配问题，同时保证对信息的封装。分配职责要看对象所拥有的信息，谁拥有哪方面的信息，谁就负责哪方面的职责，并且一个对象只做这一件事情

4) 创建者模式：（促进低耦合）通过考虑对象间的关系，来决定某个对象该由谁来创建。如果某个对象与其他对象已经有聚合/包含关系，则这个对象应该由聚合/包含它的对象来创建。这样就不需要依赖第三方，不会增加新的耦合

5) 控制者模式：对象协作设计中的一种风格，解决“处理系统事件”这一职责的分配问题

1. 如果一个程序可以接收外部事件（如GUI事件），它需要在事件源和事件处理模块之间加一个事件管理模块，从而将这二者解耦

2. 控制者（可以是外观控制者，或者一个纯虚构对象）接受所有的外界请求并转发，很可能形成集中式风格

3. Controller肯定会跟两边对象都有很严重的耦合，并且由于它负责转发很多信息，所以它的内聚度不高

6) 多态模式：处理基于类型的变更情况。当遇到根据对象的类型来选择系统行为的情况时，需要使用多态的方法调用，而不是通过if/else语句来做选择。

7) 纯虚构模式：（促进高内聚）额外创建一个虚构的类，将一些相关的、内聚度很高的职责放入这个实际不存在的类中，从而促进高内聚、低耦合、可复用。比如将数据库的处理方法放在一个虚构的类中，相关的还有与显示、平台（硬件）相关的处理代码，以及复杂的行为和数据结构，都可以放在虚构的类中

8) 间接模式：在需要解耦的两个对象之间增加一个中间对象（并将可能引发耦合的职责分配到这个对象中），防止它们直接耦合，从而提高可复用性。在filter-pipe方法中，filter中间的pipe就是间接的对象；从详细设计的角度来考虑，适配器模式就是加了一个间接对象来适配

9) 保护差异模式：识别对象的可能变更和不稳定性，设计一个稳定的接口来应对将来可能发生的变化，由稳定的接口来承担对象的访问职责

# 创建者模式（creator）

**问题——谁负责创建某类的新实例；解决方法——赋予B创建A的职责如果（B聚合A，B包含A，B记录A的实例，B密切使用A，B有创建A的初始数据），其中第一条或第二条最好；类拥有创建需要引用的对象的职责来减少Connection以降低耦合，自己创建实例可以避免依赖于其他类为条目创建实例**

谁拥有创建的信息，谁包含被创建对象，谁统计被创建对象的信息，谁负责创建对象。

Sale负责创建SalesLineItem,需要数量作为参数，因为只有Sale知道需要多少个数量的Item。

# 低耦合（low Coupling）

**问题——低依赖度、高复用性；解决方案——分配职责保持低耦合；类易于维护和复用，变更本地化**

一个类型A和类型B之间的耦合关系有如下几种形式：

B是A的属性

A中的一个方法里面使用到B类型，或者以B类型作为参数

B是A的子类，或者间接的子类

A是一个接口，而B实现了A

三个类

Payment ，Post ， Sale

Sale负责创建payment，因为sale知道总价钱。

# 高内聚（high Cohesion）

**问题——使得复杂性在可管理的范围内；解决方案——分配职责保持高内聚，一个功能领域的中等规模职责，与其他类协作完成任务；类易于维护和理解，增益低耦合，职责粒度适中故而为复用提供支持；**

内聚种类：

非常低内聚：一个类负责不同功能区不同的事情

低内聚：一个类负责相同功能区里面不同的事情

高内聚：一个类负责相同功能区里面一些职责，并和其他类一起完成该功能。

* + **Rule of thumb（首要原则）**

# 控制器 (controller)

**问题——谁负责处理系统事件；解决方案——如果程序接收不是从图形接口来的外部源事件，添加一个事件类来解耦事件源和处理事件的对象；将处理系统事件消息的职责分配给类（业务或者全局组织——facade controller，全局“系统”——facade controller，领域中执行工作的活动体，人工类型（纯虚构）——use case controller）；增加了复用的可能性，将外部事件源和内部事件处理的类型和行为独立开，论证用例的状态，确保系统操作以合法的顺序出现，或能论证用例中活动和操作的现状态；使用Controller对象保证外部事件源和内部事件处理的类型和行为独立开，拥有更多的职责controller对象可能高耦合低内聚**

指的是事件控制器，对于一个非图形界面事件，最好能够有专门的控制器来针对它做出处理。

对于一个entereItem（）的事件来说，需要有一个专门的控制器，接受事件，然后处理事件。

# 多态性（Polymorphism）

**问题——谁处理基于类型的多个选择；解决方案——基于类型有多种行为选择时，使用多态方法调用来选择行为而不是使用if语句来测试类型；比显示选择逻辑更简单和可靠，更易于增加附加行为，但增加了设计中类的数量，也使得代码更难跟踪**

多用多态来实现行为判断，最好不要用if语句。

# 纯虚构（Pure Fabrication）

* **问题——如何不违背高内聚低耦合；解决方案——将高内聚职责结合分配给在问题域中没有对用的人工类，只有状态或只有行为，从而支持高内聚低耦合和复用；由于职责被分配给关注一个非常特化的相关任务集的类而得到高耦合，适中粒度的纯虚构类增加复用的可能性；例子，操作数据库的类，适用场景——分离表示层和模型层、分离平台（facilities）和模型层、分离复杂行为或复杂数据结构**

指的是，要实现高内聚，必须将功能单一化，虽然有时候class满足专家模式，但是，对于一些大型的，需要多次重用的方法，需要单独提取出来作为一个类，这样能够增加重用性。

# 间接性（Indirection）

* **问题——如何避免直接耦合、解耦对象以支持低耦合和高复用可能性；解决方案——将职责分配给一个中间对象来协调其他组件或服务，使他们不会直接耦合；与可变性解耦，提升复用**

有时候需要用间接的耦合来代替直接的耦合，对于上一个例子，通过建立一个中间类别来解开sale和database之间的耦合，把直接耦合转化为间接耦合。

# 防止变化（Protected Variations）

* **问题——如何设计对象、子系统和系统以便这些元素的差异性或不稳定性不会对其他元素产生非预期的影响；解决方案——识别可预期的差异点和不稳定点，分配职责以创建一个稳定的接口；信息隐藏，数据驱动（配置文件），服务lookup（运行时注册），interpreter驱动（泛化模块），映射或元数据级别设计（组件代替），统一存取（协议依赖），LSP（多态），Demeter法则（限制交互路径）**

# 面向接口编程

## ! 迭代器模式（Iterator）

通过一个迭代器来访问容器。这样可以做到多态，因为对于不同的容器，他们的迭代器是相同，所以，你可以指定一个迭代器的接口，即可以访问所有容器的数据而不需要具体知道是哪个容器。；

## ! 代理模式（Proxy）

如果访问真实的主题有危险，或者有某些禁忌的话，可以使用代理模式，它通过将真实的主题和代理的主题共同实现一个父类来达到目的，父类中只有一个方法，供给客户端使用。

在代理中可以使用preRequest和postRequest。

## 原型模式（Prototype）

通过给出一个原型对象来指明所要对象的类型，然后用复制这个原型对象的办法创建出更多相同类型对象。

# 依赖倒置——DIP

高层模块不应该依赖于底层模块，二者都应该依赖于抽象

抽象不应该依赖于细节，细节应该依赖于抽象

## 调停着模式（Mediator）

同样是一个中介的意思，也就是说，对于多个对象之间的复杂联系，通过一个中介来将他们所有的行为联系起来。

## ！桥接模式（Bridge）

桥接模式最大的特点在于将2个不同方面的组合通过继承和接口的方式将他们组合起来，也就是我们数学里面的排列组合；

例如：A B 两个方式写入数据，而C D两个方式写入不同的平在

而传统的方式在与通过单方面的技能来实现，而桥接模式则能够实现AC，BC，AD，BD之间的任意组合。

## ! 外观模式（Facade）

就是在一个子系统或者系统中（可能包含了很多种方法），通过建立一个类，来将这些你需要的方法的使用顺序，调用顺序组合起来，也就是一个中间类别的意思，同样是功能的抽象化。

Facade模式主要适用于以下几种情况:

* 不需要使用一个复杂系统的所有功能，而且可以创建一个新的类，包含访问系统的所有规则。如果只需要使用系统的部分功能，那么你为新类所创建的API将比原系统的API简单的多。
* 希望封装或者隐藏系统原系统。
* 希望使用原系统的功能，而且还希望增加一些新的功能。
* 编写新类的成本小于所有人学会使用或者未来维护原系统上所需的成本

# 开闭原则OCP

对拓展开放

对修改关闭

也就是说我们在设计一个类的时候，是不需要修改类本身的代码就可以进行拓展。

## ! 修饰者模式（Decorator）

最主要的特点是代替继承，实现类的拓展

还有就是动态的添加职责。

## 适配器模式（adapter）

对于类适配器：就是用一个类继承一个父类，同时实现一个接口

对于对象适配器：就是用一个类，他有上一句中父类的属性，同时实现一个接口。

## ! 策略模式（Strategy）

通过类拥有一个接口的属性，通过具体不同的实现该接口的类来做不同的事情。

# 共性与可变性

找出共性

判断共性中还有那些可变的地方

通过一些方法来解决可变性

方法如下：

If语句是最差的方法

策略模式比继承好

## ！状态模式

## ! 观察者模式（Observer）

通过一个存放观察者的列表，维护观察者。然后在需要调用观察者的时候在notify链表对应的观察者。

## 命令模式（Command）

通过将命令封装成类的，通过指定统一的接口来实现命令的统一。

## 工厂模式（Factory）

1、简单工厂模式，又叫做静态工厂方法（Static Factory Method）模式。   
2、工厂方法模式，又称为多态性工厂(Polymorphic Factory)模式   
3、抽象工厂模式，又称工具（Kit或ToolKit）模式

## ！单例模式

# 风格

## Share Data共享数据风格：

repository使用pull风格，需要agent不断地查询中间的数据是否发生变化，如果发生变化agent会做出响应；blackboard使用push风格，每一个agent会向中间数据注册，如果中间数据发生变化，则会通知每一个agent。它的部件包括中心数据结构以及一系列操作于中心数据结构之上的部件，连接件是程序调用或者直接的内存访问

1. 优点：能够储存大量数据，因为数据是共享的；集中区可以做一些复杂的设计；安全控制、并发控制、复用、备份都比较容易处理

2. 缺点：所有的agent都依赖于中间的数据区，这里是瓶颈；如果要修改中间数据，则会大幅度提高修改成本

## Pipe-Filter管道过滤器风格：

类似于一条流水线，流水线上的模块并发地做自己的工作，数据传递通过pipe传递。各个filter之间并发（只传递数据流）且独立（无控制流，且不共享数据），每一个filter都不知道其他的filter，其正确性完全独立，不依赖于上一个/下一个filter。它的部件是filter，连接件是pipe

1. 优点：一个系统的整体I/O行为就是对所有filter行为的组合；只要两个filter之间传递的数据一致，则它们就可以连起来使用，因此可复用性较高；因为filter是独立的，所以增加、删除filter比较容易；所有filter的操作通常可以并发，提高系统的性能

2. 缺点：不适用于交互式系统；传输数据需要额外的空间；两个filter的数据格式不一样的情况下，对数据打包、接包、转换格式会增加程序复杂度

## Implicit Invocation隐式调用风格：

基于事件/消息的方式相互调用。部件向广播媒介中声明一个事件，其他的部件可以注册监听这个事件，并将自己的程序与这个事件联系起来，当事件发生后，广播媒介调用所有监听了该事件的程序。它有以下约束：每一个事件的抛出者并不知道谁是接收者，也没有默认，因为它们之间是严格解耦的；事件抛出后，不能假设这个事件的处理顺序，甚至它是否会被处理都是未知的。它的部件是所有的代理（可以是对象、进程、方法），它的连接件是广播媒介（即事件管理）

1. 优点：只要部件注册到这个系统中的某一个事件，它就可以被用在这个系统中，因此可复用性较高；因为部件之间非常独立，部件的变更和替换不会需要其他部件修改接口

2. 缺点：因为事件是否会被处理是未知的，故而程序完整性和正确性很难被保证；程序很难测试和诊断

## Main Program and Subroutine主程序和子程序风格：

结构化，程序被分为一定的层次。控制权转移有严格要求，单线控制。这种方法的算法中，所有的子程序都由主程序调用。它的部件是程序、方法、模块，连接件是程序调用

1. 优点：程序清晰、容易理解，且有很强的正确性控制

2. 缺点：难以变更或复用，可能造成公共耦合

## Layered分层风格：

系统分成很多层，每一层都是一个独立的抽象部分。层之间通过程序调用来交互，只允许上层调用下层，不允许下层调用上层，而且不允许跨层调用。它的部件是一组特定的程序和对象，连接件是遵守可见性限制的程序调用

1. 优点：分层式结构适用于并行开发，只需要将各层的协议定义出来，各层内部的开发交由不同的开发小组各自负责即可；任何变更、增加都最多只会影响到两层，影响只会向上转移一层，绝对不会向下转移；只要接口相一致，就有很高的复用性

2. 缺点：不是所有的系统都能使用分层结构；逐层的调用会使得调用本身产生延迟，造成性能下降

## Object-Oriented面向对象风格：

对象将秘密封装起来，不与外部环境耦合，以获得较高的可修改性；对象是自治的，负责自己内部的数据表达和完整性；对象是平等的，不受其他对象控制，只能通过调用方法来访问一个对象。它的部件是对象或模块，连接件是方法调用

1. 优点：能够将数据封装起来，不会因为数据共享而产生相互之间的影响；因为对象的自治性质，可以将系统逻辑分成一个个小的逻辑

2. 缺点：如果要使用某一个对象，则必须先知道某一个对象的身份，则如果该对象的身份发生了变化，会对所有调用这个对象的其他对象产生影响；对象协作可能引发副作用

## b) 对给定场景，判断需要使用的风格

1) Main Program and Subroutine主程序和子程序风格：需要顺序执行任务的系统；对正确性要求很高的系统

2) Object-Oriented面向对象风格：适用于有一个中心问题并且需要保护相关信息的应用中；数据展示和相关操作被封装在抽象数据类型中

3) Layered分层风格：能够把程序分成不同的层次，层次之间的协议是稳定的，程序可能只在某一层内部发生修改。比如OSI七层模型，比如UNIX系统

4) Pipe-Filter管道过滤器风格：如果有一系列任务需要操作一个有顺序性的数据，且这些任务需要独立执行，那么便适用于此风格。比如有顺序的批处理程序、UNIX/Linux命令管道、编译器、信号处理、大数据量处理

5) Implicit Invocation隐式调用风格：通常适用于松散耦合系统，部件之间关联比较少，每一个部件都有它独立的操作，但是会有事件发生需要用到某些操作。

比如调试系统、数据库管理系统、GUI

6) Share Data共享数据风格：大型信息系统，能够将数据做集中处理，并且需要维护一个复杂的中心信息。常见如数据库、专家系统、编程环境。在网络应用中，网络日志使用repository风格，而网络聊天室使用blackboard风格