# **名词解释**

## 模块化

指解决一个复杂问题时自顶向下逐层把系统划分成若干模块的过程。每个模块完成一个特定的子功能，所有的模块按某种方法组装起来，成为一个整体，完成整个系统所要求的功能。模块具有以下几种基本属性：接口、功能、逻辑、状态，功能、状态与接口反映模块的外部特性，逻辑反映它的内部特性。

## 信息隐藏

指在设计和确定模块时，使得一个模块内包含的特定信息（过程或数据）——decision，这个decision只有模块内部了解，对于不需要这些信息的其他模块来说，是透明的，这样修改decision时，不会产生连锁反应。

## 软件体系结构（三个不同方面）

* Perry and Wolf：Software Architecture = {Elements, Form, Rationale}
* Shaw and Garlan: The architecture of a software system defines that system in terms of computational components and interactions among those components. (Component + connector + configuration)
* Bass, Clements, and Kazman: The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

## 4+1 View

描述软件体系结构的一种方式。

## GRASP模式（或其中之一）

参见下文GRASP模式一节

## 设计模式

是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。它通常是基于面向对象的，包括类和模块之间的依赖关系、结构、交互、公约。它主要包含四个因素：模式名称、问题、解决方案和结果。

## MVC

模型(Model),视图(View)和控制Controller)。使用MVC的目的是将M和V的实现代码分离，从而使同一个程序可以使用不同的表现形式。C存在的目的则是确保M和V的同步，一旦M改变，V应该同步更新。

# **模块化**

## 结构化的内聚与耦合类型

* 1. 耦合
     1. 内容耦合：将数据和控制元素混合起来。（一个模块直接访问另一个模块的内部数据；一个模块不通过正常入口转到另一模块内部；两个模块有一部分程序代码重叠；一个模块有多个入口）
     2. 公共耦合：如果一个connection连接到共享变量或数据，此时使用这些的模块都不是独立的，一个模块进行修改其他模块都会发生修改，其可理解性查，并且很难复用；公共环境中的每个元素都会增加整个系统的复杂度；使用接口传递参数的方式进行交互比使用共享变量的可增长性和可维护性更好。
     3. 控制耦合：connections传递数据和控制元素。
     4. 标记耦合（Stamp Coupling）:connections传递参数。
     5. 数据耦合：connections传递必须的参数。（数据参数而不是控制参数、公共数据结构或外部变量）
  2. 内聚
     1. 偶然内聚：代码随机堆在一起，没用共同的目的，内部联系少，外部联系增加。
     2. 逻辑内聚：根据逻辑的相似性聚合。
     3. 时间内聚：根据逻辑的相似性和时间的相关性聚合。仍然产生复杂的联系。
     4. 通信内聚：根据逻辑的相似性和数据相关性聚合，通常是读写共同的数据。
     5. 顺序内聚：根据解决某个问题的步骤进行绑定。
     6. 功能内聚：所有的元素都与同一个共同相关，是最好的内聚。
     7. 信息内聚：？

## 面向对象的耦合类型

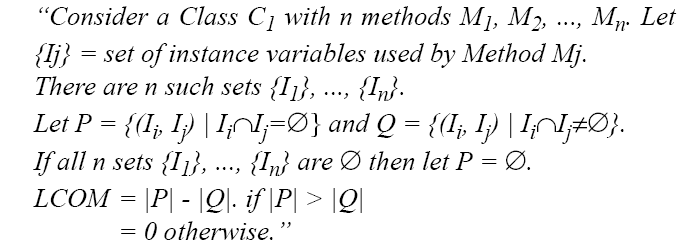
1. 交互耦合（Interaction Coupling）：
   * 1. 单交互耦合：方法调用和属性访问来进行选择，传递参数为对象类型不予考虑，不考虑继承问题。
2. 部件耦合（Component Coupling）： 一个类使用另一个类来声明自己的一个变量；并不包含集成关系；一般需要类级别。
   * 1. 四种类型
        1. 整个变量：聚集
        2. 参数形式传递：方法参数
        3. 创建：在方法局部区域中创建某个对象
        4. 隐藏：由其他对象给出
     2. 最差的耦合式隐藏的，然后为松散的（Scattered），然后为指定的。
     3. Demeter法则：四种可以调用的函数的情况
        1. 对象可以调用自身的函数
        2. 对象可以调用子对象的函数
        3. 对象可以调用其他对象给自己的对象的函数
        4. 对象可以调用自己创建的对象的函数
     4. 隐藏式耦合：有A，B，C三个类，A中X方法调用B中Y方法，B中Y方法自动调用C中Z方法。
     5. 松散式耦合：在A中的某个方法域中使用了B的对象。
     6. 指定式耦合：在A中声明了B的对象作为其属性。
3. 继承耦合：认为父类是子类的一个规格，一般情况下认为父子类之间没有耦合。
   * 1. 修改：子类已经打破了父类的许多规格，是最差的继承耦合。父子类已经不相似，connections+2，更加复杂。
     2. 完善：继承信息，除了子类修改的部分不需要关注其他部分信息。connections+1
     3. 扩展：完全没有修改父类的信息，只是增加某些内容。扩展是最好的，1 connection。

## 对给出的例子

### 判断其内聚和耦合的类型，并将其修正为正确的类型。

### 分析其Connection类型，计算不同Connection类型的数量。

* + 完善和扩展继承耦合时，父类和子类之间的interaction coupling是忽略的。
  + 计算其他类的数量：使用这个类的方法或变量或包含这个类的方法或变量，不含继承。
  + 类的职责：当一个消息被接受后要执行的方法的数量。
  + 消息传递耦合：将所有功能运行完毕后才能决定是否含有。
  + 数据抽象耦合：使用多少其他类型组成自己的抽象类型。
  + 传入传出耦合：传入指被依赖，传出指依赖。
  + 继承树的最大深度：继承树越深，理解上和编写上的复杂度越高。
  + 父类下子类的数量：子类越多，调试、抽象起来越困难。
  + LCOM：



如果结果>=1，那么，这个类应该被分解

# **信息隐藏**

**原则：**

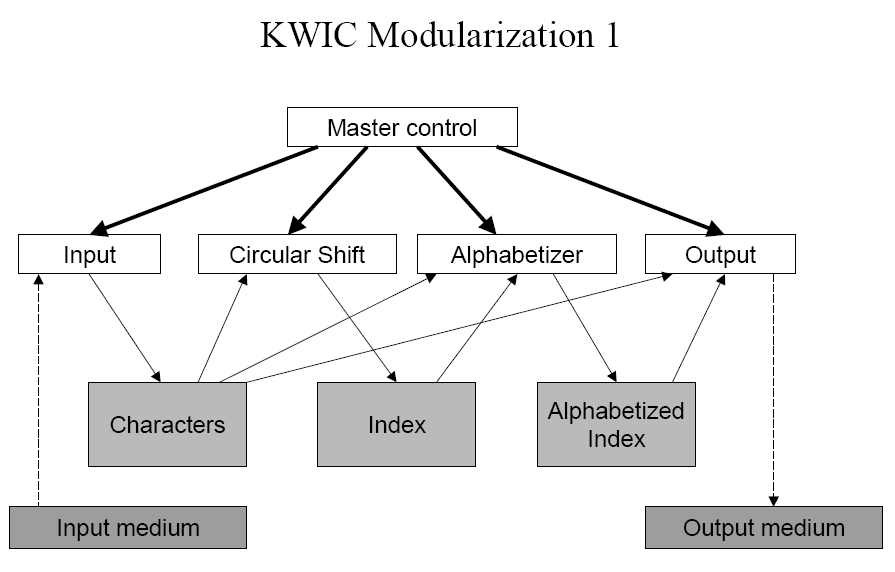
**每个模块都实现一个secret**

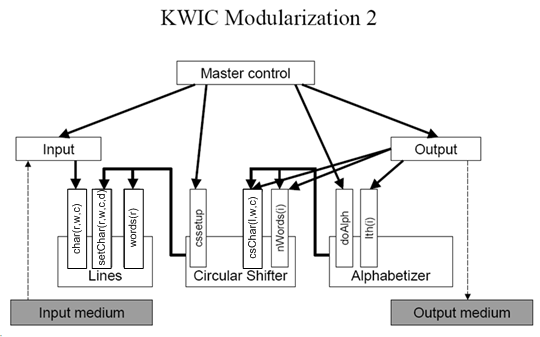
**每个模块的接口进行抽象明确的描述**

**从实现的细节中抽象出可能发生修改的部分**

**把可能发生修改但不能被隐藏的部分参数化**

## 对简单的应用，按照“处理流程”与“信息隐藏”两种方式分别给出设计描述。





Lines

how characters/lines are stored

Circular Shifter

algorithm for shifting, storage for shifts

Alphabetizer

algorithm for alpha, laziness of alpha

## 对简单的应用，解释其Module Guide

* 1. 对于每个模块来说，Module Guide应该包含：
     1. 与其他模块相关的secrets：Primary Secret（隐藏对外信息，从用户需求角度来说），secondary Secret（隐藏实现细节，从实现角度）
     2. 独立模块在整体系统中的角色
     3. 每个模块所提供的功能
  2. 实例请参见飞机场的例子。。。

# **软件体系结构风格**

## 描述或比较相关风格

# **全局分析的步骤**

抹平需求和体系结构之间的gap

1、关注Architecture Design Requirement

ADR是需求、质量属性、约束和其他影响体系结构的因素。它们不出现在SRS中。这些因素应该文档化。

2、用System Solution Strategies （SSS）模型关联ADR和软件体系结构的结构模型。

SSS应该包含结构的和非结构的策略。

这些应该文档化，包括他们的上下文，基本原理，以及如何关联到ADR。

3、指导体系架构师用SSS开发软件体系结构。

比较和选择解决策略

支持软件开发过程的其他部分

4、ADR和SSS应该对软件开发过程的其他部分产生影响。

对SRS的可行性、完整性和一致性做出反馈。

确保体系结构级别的需求被满足。

识别开发中的风险部分并和stakeholder交流。

将重要的体系结构决策告知stakeholder

跟踪体系结构到需求以及需求到体系结构。

两个基本部分

设计、描述系统的结构

使用4个视图以降低设计和理解体系结构的复杂性

不同的视图关注不同的点，不需要同时考虑所有的系统方面。

全局分析

全局分析开发出描述设计决策上下文和基本原理的文档。

识别外部影响因素和影响体系结构的临界需求。

产生设计体系结构的决策。

2a

分析因素（约束和质量）

识别并描述因素

描述因素，识别这些因素的灵活性和可变性

分析这些因素的影响

2b开发策略

1、识别出争论点和影响因素

关联因素和问题

分析限制和约束

开发解决方案，明确策略

指出相关策略

SA 设计活动

结合全局分析和多视图结构

从一个初始化视图框架开始

根据功能需求设计

做出决策

根据策略

修改视图元素

# **根据分析类图，建立基本的设计类图**

添加属性、操作和association

提取父类

不要单纯为了复用而使用继承，只有在存在泛化关系时再使用。

应用设计模式

提取接口

Change actors to interface classes.

add actor domain classes.

start up class

controller

coordinator

class for data type

container class

engineering design association

# **有几种控制流处理方式？进行比较**

组件和交互必须同时设计。

交互设计应该自上而下

最抽象的交互在SRS和用例模型中定义

从程序的交互和环境（outside）开始，设计如何实现交互的模块（inside）称为outside-in design。

Controller

A **controller** is a program component that makes decisions and directs other components.

Controller Styles

将decision making分布到各程序组件的方式。

*Centralized*

集中式：少量控制器做出所有的决策。

易于找出做出决策的位置。

易于查看和修改做出决策的过程。

控制器可能变得过于庞大和复杂，难以理解、维护和测试。

控制器可能会将其他组件视作data repositories，增加了耦合，破坏了信息隐藏。

Delegated

代理式：少量控制器做出主要决策，其他决策则分布在程序中。

控制器只和少量的组件耦合，降低了耦合。

信息被更好地隐藏。

程序更易于分层。

代理式控制是推荐的 control style。

Dispersed

分布式：所有决策均分布在程序中。

许多组件没有分配到数据和职责。

控制流难以理解。

组件无法只依赖自身完成职责，增加了耦合。

难以做到信息隐藏。

内聚性差。

不满足模块化原则。

**GRASP模式**

GRASP: General Responsibility Assignment Software Patterns

不是设计模式，而是对象设计的基本原则；着眼于职责分配，而不是体系结构设计。

Expert:

将职责分配给 **拥有完成该职责所必须的信息** 的对象。

优缺点：

保持了信息封装

促进低耦合高内聚

可能导致某个类过于复杂。

Creator

根据潜在的creator class和待实例化的class之间的关系，决定由哪一个类来实例化另一个类。

出现下列情况时，由class B来实例化class A

B *aggregates* A objects

B *contains* A objects

B *records* instances of A objects

B *closely* uses A objects

B *has the initializing data* for creating A objects

最好是当

B *aggregates* or *contains* A objects

优缺点：

通过由实例对象自行创建需要引用的对象，促进低耦合。

通过自行创建需要引用的对象，避免了对其他对象的依赖。

Low coupling

分配职责时保持低耦合

好处：

使得类易于理解、维护和复用

限制了变化所影响的范围

High cohesion

分配职责时保持高内聚

好处：

使得类易于理解、维护和复用（易于复用是因为分配到的职责粒度合适）

支持低耦合

Controller

如果程序需要从内部接收事件，则增加一个事件类可以使产生事件的对象和处理事件的对象解耦。

优缺点：

增加了复用的可能性。controller使得内部事件产生和内部事件处理相互独立。

使得controller object与更多的职责高耦合却无内聚。

Polymorphism

当需要根据对象的类型选择多种行为中的一种时，最好使用多态机制，而不是用if来测试对象的类型。

优缺点：

比显式地使用选择逻辑更加简单和可靠

以后易于增加额外的行为

增加了需要设计的类的数目

可能使得代码难以跟踪

Pure Fabrication

为了做到高内聚、低耦合，将一组高度耦合的职责分配给一个人为的、不代表任何问题域模型的对象。

举例：DAO，Data Access Object

典型的场景

分离表示和模型

分离平台和模型

分离复杂的行为

分离复杂的数据结构

好处：

保持高内聚低耦合，方便重用。

Indirection

将多个组件和服务之间的交互职责分配给中介对象，以解除这些组件和服务的直接耦合。

好处：

低耦合，便于修改

提高了可重用性

Protected Variations

识别出可预见的变化和不稳定，分配职责时围绕它们创建稳定的接口。

Information Hiding

Data driven (configuration files)

Service lookup (runtime registration)

Interpreter（解释）-Driven(generalize module)

Reflective or Meta-Level Designs (Component replace)

Uniform Access (adherence to protocols)

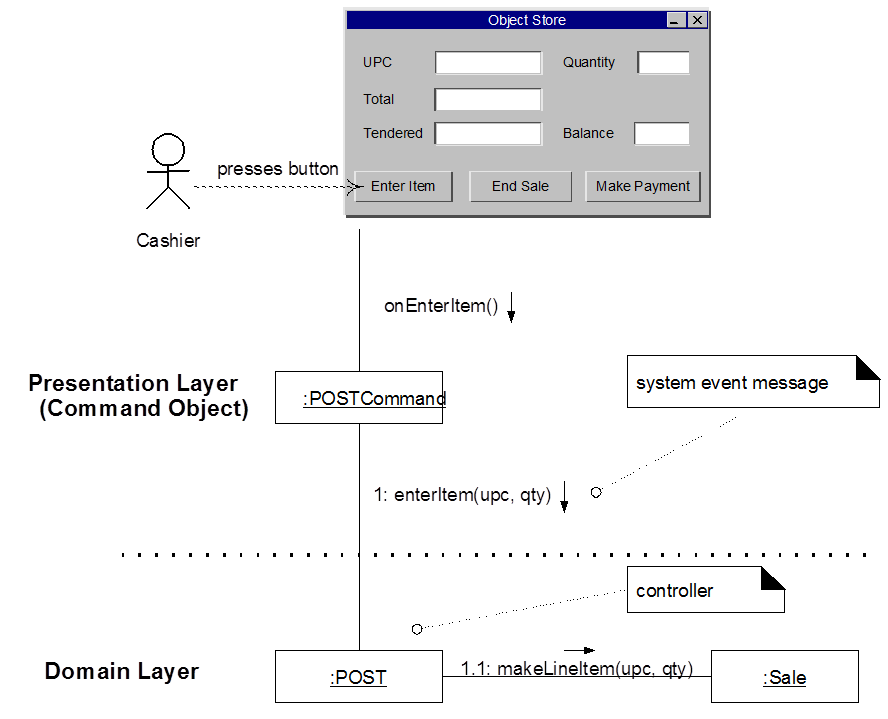
LSP (polymorphism)

Law of Demeter (restrict communication paths)

## 对给定场景，判断职责的分配

1）增加一个事件类，使产生外部事件的对象和处理事件的对象解耦。

2）增加一个间接对象由处理事件的对象引用，使得处理事件的对象和业务模型解耦。



# **设计模式**

## 普通Programming to Interfaces有哪些手段？

## 集合类型Programming to Interfaces有哪些手段？

## 一个模块的信息隐藏有哪两种基本类型？

## 实现共性与可变性有哪些手段？

## 对象的创建有哪些常见解决方法？

## 设计模式部分所有的思考题