# 1名词解释：

模块化（关键是思想）、信息隐藏、软件体系结构（三个不同方面，高层，关注点，设计结构）、4+1 View、GRASP模式（任意挑一个或其中之一）、设计模式（主要考designpattern的？？，ppt上不明确）、MVC（特点，画出solution）

**1）模块化：**

——是一种责任分配，而不是单纯的子程序；

——代表具体对自身的设计决策而不被其他模块知道

——支持实现中的灵活性

——不要描述过程的步骤

——高内聚、低耦合

模块是可以被独立对待的：即可以独立的理解、使用（或复用）、编译等；一个模块的错误不会影响其他模块；模块的改变也不会产生连锁反应。

抽象和分解：抽象是把不要的东西扔掉，去掉不重要的，隐藏细节；

**2）信息隐藏：**

——基本设计原则：

——每个模块的实现都是秘密；

——每个模块的接口被抽象而精确地描述；

——我们抽象出可能会变化的实现细节；

——我们将那些不能被隐藏的变化部分进行参数化；

——关键技术是尽可能多的限制对程序实体的访问。

——每个模块包含一个基本的秘密：外部行为VS内部实现。每个模块隐藏重要的设计决策的实现以达到只有那个模块本身知道实现细节。所有的设计决策都是相互独立的。

——模块拥有额外的秘密：变化

——百科：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），对于不需要这些信息的其他模块来说，是透明的。通过信息隐藏，可以定义和实施对模块的过程细节和局部数据结构的存取限制。

3）软件体系结构（高层结构、关注点、设计风格）：

——高层结构：

部件、连接件、配置

高层抽象的作用

高层结构

需求（质量属性）、项目环境、商业目标

交流媒介的作用

关注点

问题、候选设计、理由、解决方案

指导开发的作用

设计决策

软件体系结构

——部件：软件体系结构的一个基础元素，承载了系统主要的计算与状态，封装了系统的功能处理和数据。部件包括抽象规格与具体实现两部分。抽象规格定义了部件的特征集，包括部件的类型、功能性、约束、质量属性等特征。抽象规格还定义和命名了部件对外可见、可被外界引用的接口实体，称为端口。每个端口是一个一致的接口集合，它代表了部件对外承诺的一种职责。



——连接件：软件体系结构的另一个基础元素，承载了部件之间的交互。作为交互中介，连接件只将参与方关联到抽象的协议角色，而不是具体的部件类型。除了提供交互通道外，连接件还要维护交互规则，调节交互效果，并提供交互过程所需的各种复杂辅助机制。抽象规格定义了连接件的特征集，包括类型、接口规则、交互断言、交互协议（例如顺序、性能）等。连接件抽象规格所定义和命名的对外可见、可被外界引用的接口实体称为角色。每个角色代表一个交互参与方需要满足的一些条件，基本的条件是匹配该角色的端口所应符合的规则，复杂的条件可能会包括加密通信、负载均衡等。



——配置：一种将部件和连接件整合起来，构成系统的整体结构，达到系统设计目标的专门机制。配置通过部件端口与连接件角色相匹配的方式，将系统中部件和连接件的关系定义为一个关联集合，这个关联集合可以形成系统整体结构的一个拓扑描述。

——高层抽象的作用：软件体系结构最为核心的思想是将注意力集中在系统总体结构的组织上，它实现的手段是运用抽象方法屏蔽错综复杂的模块间连接，使人们的认知提升并保持在整体结构的部件“交互”层次，并进一步将交互从计算中分离出来，建立“部件+连接件+配置”的软件系统高层结构组织方式。

——关注点：软件体系结构的主要关注点可以分为三个类型：

——需求。需求是用户和维护人员关注的内容，它包括（要求系统帮助用户完成任务的）功能需求，也包括（衡量功能实现程度的）质量属性。一个好的软件体系结构必须要承载需求，以完成从需求向设计的转换与过渡。

——项目环境。项目环境是客户、项目管理者和开发者关注的内容。客户关注的是项目的商业环境，例如目标市场的大小等。项目管理者关注的是项目的开发环境，例如成本、人员组成等。开发者关注的是技术环境，例如中间件平台的选择等。项目环境限制软件体系结构设计的决策范围。

——商业目标。商业目标是客户的主要关注内容，它说明了软件系统被开发的出发点和战略目标。商业目标是引导软件体系结构设计的上下文，为软件体系结构的设计决策指明方向。

——交流媒介的作用：对涉众关注内容的选择、表现、权衡折中等工作使得软件体系结构变成了涉众需要的一种综合与平衡。因此，对于项目而言，软件体系结构不仅仅是一个高层抽象的技术蓝图，还是一个体现项目主要关注内容的需要蓝图。技术蓝图和需要蓝图的融合使得软件体系结构具备了工程蓝图的作用，指导后续开发工作的同时，还起着交流媒介的作用。

——设计决策：



——决策是指决定策略与办法，将软件体系结构设计中的决策定义为：“对元素、特征和处理的选择，它们涉及一个或多个关注点，直接或间接地影响到软件体系结构”。

关于体系结构设计决策所应该包含的具体内容，有着很多不同的看法一般来说，体系结构设计决策核心的知识可以分为四个部分：关注点、解决方案、策略和理由，

——指导开发的作用：软件体系结构设计决策将关注点转化到高层结构之上，使得后续开发工作可以在一个限定的范围内、稳定可靠的框架下进行。同时，软件体系结构设计决策所产生的设计规则、设计约束和附加需求也对后续的开发工作提出了要求和限制。所以，作为设计决策集合的软件体系结构既是后续开发的工作基础，又限定了后续开发必须要满足的上下文环境，对后续开发工作具有指导作用。

4）4+1 View：

——逻辑视图：面向对象分解，系统将问题域分解成一系列关键的抽象，以对象或类的形式表现。

——view：最终用户

——consider：功能需求

——不仅是功能性分析，还可以识别系统不同部分之间共同的机制和设计元素。

——进程视图：进程分解

——view：Integrator

——consider：非功能需求（并发、性能、scalability）

——style：几个风格都可以满足这个视图

——使用多层次的抽象，最高时进程的逻辑网；系统被分成几个相互独立的任务：主要任务是体系结构相关的任务、次要任务是帮助类的任务

——重点关注系统运行起来之后的特征

——开发视图：子系统分解

——viewer：程序员和软件经理

——consider：软件模块组织（层次结构、软件管理、复用、工具约束等）

——style：分层风格

——物理视图：将软件映射到硬件上

——viewer：系统集成师

——consider：非功能需求（可用性、可靠性（容错性）、性能（吞吐量）、scalcbility）

——场景：将所有放在一起

——viewer：其他视图所有人和评价者

——consider：四个视图间的一致性、可验证性

——体系结构设计阶段帮助架构师；帮助解释和验证文档

5）GRASP模式（或其中之一）：

General Responsibility Assignment Software patterns（通用职责软件模式），核心思想是职责分配，用职责设计对象。

——主要特征：对象职责分配的基本原则；主要应用在分析和建模上。

——核心思想理解：自己干自己的事（职责的分配）；自己干自己的能干的事（职责的分配）；自己只干自己的事（职责的内聚）；

——包含9个基本模式：

1 信息专家：解决类的职责分配问题的最基本模式。

——问题：当我们发现完对对象和职责后，职责的分配原则是什么？

——解决方案：职责的执行需要某些信息，把职责分配给该信息的拥有者。即某项职责的执行需要某些资源，只有拥有这些资源的对象才有资格执行职责。

——优点：信息拥有者类同时就是信息的操作类，可以减少不必要的类之间的关联；各个类的职责单一明确，容易理解；保持信息的封装性；促进低耦合和高内聚；造成某个类过于复杂。

2 创建者：解决类的实例和创建职责问题的模式。

——问题：类的实例的创建职责，应该分配给什么样的类？或者说类的实例应该由谁来创建？

——解决方案：B包含A，或B聚集A，或B记录A，或B频繁使用A或B有A初始化数据时，类A的实例的创建职责就分配给B。（其中，最提倡聚集和包含）一般用工厂模式或抽象工厂模式作为替代方案。

——优点：整个结构清晰易懂；有利于类或组件的重用；防止职责的分散；降低耦合性。避免依赖其他的类来创建自己的对象。

3 高内聚：为降低类的复杂程度，简化控制而提出的面向对象设计的原则性模式。（其他模式的根本）

——问题：怎么做才能降低类的复杂程度，简化控制？

——解决方案：紧密相关的功能（职责）应该分配给同一个类。

——优点：聚集相关功能，结构清晰，容易理解；只聚集相关功能，使得类的职责单一明确，从而降低类的复杂程度，使用简单。类易于维护；支持低耦合；支持复用。

4 低耦合：为降低类之间的关联程度，适应可变性而提出的面向对象设计的原则性模式。（其他模式的根本）

——问题：怎样做才能降低类之间关联程度，适应需求的变化呢？

——解决方案：为类分配职责时，应该尽量降低类之间的关联关系（耦合性）。亦即，应该以降低类之间的耦合关系作为职责分配的原则。

——优点：独立性，有利于重用和维护；适应需求变化，一旦发生变化时，可以把影响范围缩小到最小范围。

5 控制器：解决事件处理职责问题的模式。

——问题：在UI层之外，应该由哪个类来处理（控制）系统操作（事件）呢？

——解决方案：把系统事件的处理职责分配给控制器类。担当控制器类角色的候补类可能为：系统全体，设备，子系统等的表现类（Façade Controller）；系统实践发生的用例的控制类，通常被命名为Handler，Coordinator，Session等。

——优点：防止同类职责的分散。满足高内聚，低耦合原则；有利于共通处理；变化的高适应能力，能把变化的修改范围控制在最小范围内。增加复用的潜力，基本思想是控制器对象使外部事件源和内部事件处理的类型和行为相互独立（解耦）；随时查看用例的状态，保证系统操作的合法性，或解释当前活动或操作的状态。

6 多态性：扩展模式的一种，通过多态操作把基于类型的可变行为的定义职责分配给行为发生的类。（命令模式、策略模式、依赖倒置原则）

——问题：根据类型（类）的不同而发生变化的行为的定义，应该分配给谁？

——解决方案：提倡通过多台操作把基于类型的可变行为的定义职责分配给行为发生的类。继承有三种：①父类型子类型（类型多态）②处理差异性（行为多态）③为了复用（效果多态）

——优点：避免重复代码；避免重复的分歧条件；易扩展，只要实现了统一的通用接口，便可实现行为的扩展。比显示的选择逻辑更为简单和可靠；在设计中增加类的数量；会使代码不易追踪；把多个耦合变为一个耦合。

7 纯虚构：扩展模式之一，把非问题领域中的职责分配给人工定义的类，

——问题：非问题领域中的职责应该分配给谁？或者说，按照信息专家等模式分配职责时，存在某些不恰当的职责时，应该怎么做？

——解决方案：提倡把那些非问题领域的职责分配给那些人工生成的或者容易此类职责的概念类。可以只有行为没有状态或只有状态没有行为。

——优点：高内聚，不必分配问题领域以外的职责给Domain类，从而保证各Domain类内部功能上的高度聚集性；低耦合，问题领域以外的职责被分配给第三方非Domain类，一方面可以降低各Domain类之间的关联程度，另一方面可以比较漂亮地整合系统的各方面的职责；重用性，各Domain类由于功能上的聚集和关联度的降低，可以更容易地得到重用。

8 间接模式：解决类的关联问题的模式。

——问题：为了避免类之间的直接关联，应该给什么样的类分配“关联”责任？

——解决方案：当多个类之间存在复杂的消息交互时，间接模式提倡类之间不直接进行消息交互处理，而是导入第三方类，把责任（多个类之间的关联责任）分配给第三方类，降低类之间的耦合程度。（中介者模式）

——优点：高内聚，通过把“关联”的功能分散到第三方类，原来的类可以更加关注自身功能的实现；低耦合，原本关联类之间不直接关联，降低类之间的耦合性；高重用性，第三方类对“关联”功能的集中处理，与原来的类对自身功能的专注，有利于类的重用。

9 差异性保护：扩展模式之一，它设计稳定的接口应对将来可能发生的变化或其他不安定的因素。

——问题：对存在于系统，子系统，或对象等元素中的各种变化或不安定的因素，为了不产生对其他元素的不利影响，在它们中间该怎样分配职责？

——解决方案：提倡在可预测的变化或不安定因素的周围，用稳定的接口来承担职责。

——优点：提高系统对变化的应对能力。一旦系统的可预见的不安定因素发生变化（比如追加功能等），只需要生成一个已有的稳定接口的实现类就可以了，无需修改原来的类；高内聚。具体的功能在各子类中实现，各类的内部功能具有高度聚集性；低耦合。用户类只跟稳定接口通信，减少了跟其它陌生对象的关联的机会，降低了类之间的耦合性。

6）设计模式（定义）：

——过去设计当中的经验总结；通常由类与对象组成；对某种思想或原则的应用

——设计模式4要素：名称，问题，解决和效果

——（百科）一套被反复使用、多数人知晓的、经过分类编目、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。

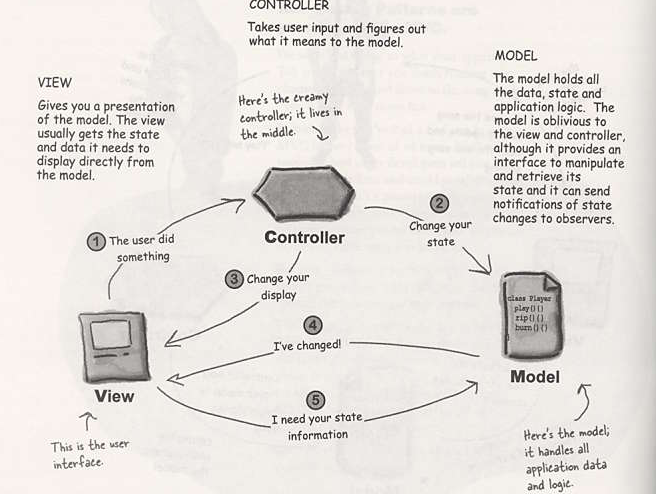
7）MVC（画出样子、特点）：

——将核心业务模型与表现形式和使用这个模型的控制逻辑分离开来

——允许多个视图分享同一个业务数据模型

——使得支持多用户的实现、测试和维护等变得容易

——至少画出这个图



——最好画出这个图



# **模块化**

## **结构化的内聚与耦合类型；**

**内聚（按程度由低到高排列）：**

●偶然内聚：代码之间没有任何相关性（statements are together by coincidence）没有共同的目的

例子：

A = B + C

GET CARD

PUT OUTPUT

IF B = 4 THEN

E = 0

●逻辑内聚：逻辑相似

* Some logical connection implied
  + example - EDIT ALL DATA
* If this is the highest form of binding in the module, then will get intertwined code

EDIT: PROC (TYPE, DATA)  
/\* do some common edits \*/  
IF TYPE = “A”  
THEN /\* edit by type A rules \*/  
IF TYPE = “B”  
THEN /\* edit by type B rules \*/  
/\* do some more common edits \*/

●时间内聚：跟逻辑内聚相似，但是在时间上有联系，比逻辑内聚

耦合简单

* + Read routing data, set up table
  + Look up distance between two airports <- Data relationship

Compare:

Get\_Distance (Airport1, Airport2) returns (distance)  
If first time used, then set up routing table

●通讯内聚：跟逻辑内聚相似，在数据上有联系

All elements related by a reference to common data

* + eg, Read and print input data
  + “Read” and “Print” are unrelated functions, but
  + “input data” is common.

●顺序内聚：解决同一个问题

* + Read next transaction and update database

●功能内聚：针对相同的目标All of the elements are related to the single function

例子：

* + Compute Square Root
  + Determine shortest route to destination
  + Condense Database

●信息内聚**：**

**如何判别是哪种内聚（用一句话来描述一个模块）：**

大于一个动词，多功能，可能为通讯或者顺序内聚

有时间词“first，next….”可能是顺序或者时间内聚

Does not have a single object,Probably logically bound

有词语“Initialize”, “Clean up”,暗示为时间耦合.

\*用一句话来描述该模块功能看是否能描述清楚，如果能，可能为功能内聚

**耦合（按程度由高到低，即由差到好）：**

●内容耦合：Hybrid of data and control elements（混合数据和控制元素）

* + Modification of one module’s code by another module
  + The target module is very dependent on the modifying module.

● 公共耦合：connection to Common environments

比如大家同时使用一个全局变量

● 控制耦合：传递了数据和控制元素

● Stamp 耦合：传递了除必须以外多余的数据

比如只需要数组的前三个元素，结果将整个数组都传递过来

● 数据耦合：传递必须的数据

## **面向对象的耦合类型；**

**对象级别：**

**●交互耦合**

两个类之间的简单的交互耦合：Method Invoke（消息调用） or Attribute Access （属性访问）between two class

不包括继承，参数和属性不包括类类型

**●组件耦合：**The component relationship between classes is defined by the use of a class as domain of some instance variable of another class，不包括继承，一般是class级别的。

一般分为四种：

Whole variable: Aggregation聚合，整体与部分的关系

Parameter: Method Parameter 参数传递

Creator: Creator in some method’s local在某个方法中创建一个

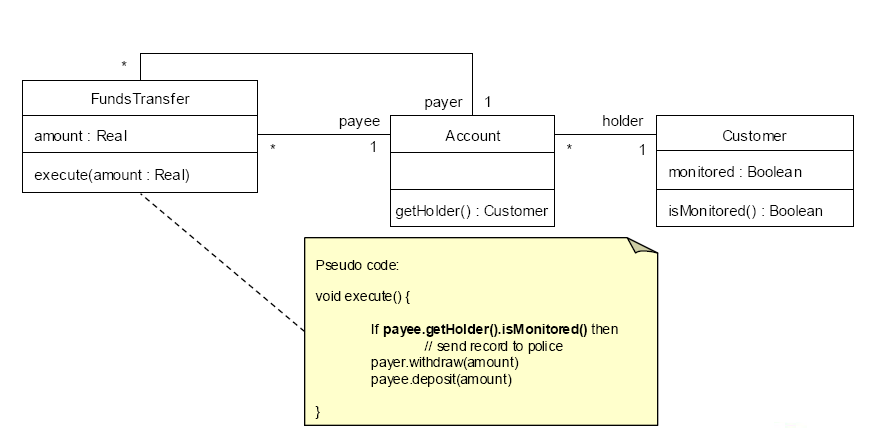
Hidden: Given by another object由其他对象给的

根据上面四种，又分为

**◎ Hidden：**最差的组件耦合，不明显

The coupling between two classes C and C’ is rated hidden if C’ shows up neither in the specification nor in the implementation of C although an object of C’ is used in the implementation of a method of C

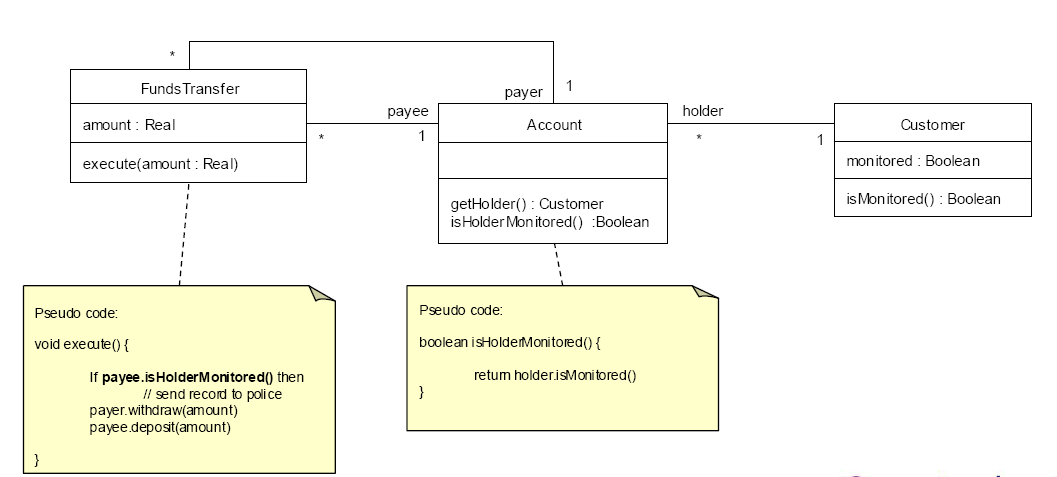
A similar problem is encountered if the return value of a method invocation is immediately used as input parameter in another method invocation



Demeter 法则：

1. 调用自己的函数
2. 调用自己子对象的函数
3. 别人给的，要有接受的本地对象之后才能使用
4. 可以使用自己创建的

修改后：



**◎Scattered（分散）：**We rate two classes C and C’ as scattered coupled if C’ is used as domain in the definition of **some local variable** or **instance variable** in the implementation of C yet C’ is not included in the specification of C

需要读完所有的代码

不可避免

**◎Specified（规格化）：**We rate two classes C and C’ as specified coupled if C’ is included in the specification of C whenever it is a component of C（在接口中使用） 比分散耦合好，不需要读懂全部代码

解决方案: Get Specified, Design by Contract——Suffered Interface VS Required Interface

* + Suffered Interface: Specified
  + Required Interface: Used

**●继承耦合**

可以分为三类：

**◎Modification（修改）**：Modifying without and rules and restricts

最差的继承耦合

If a client using a parent ref, the parent and child method are all needed

* + Implicit
  + There are two connections, more complex

Harm to polymorphism

**◎Refinement （完善）：**根据规则修改继承的部分再加上额外的实现

* If a client using a parent ref, the whole parent and refinement of child are needed
  + 1+connections
* Necessary!

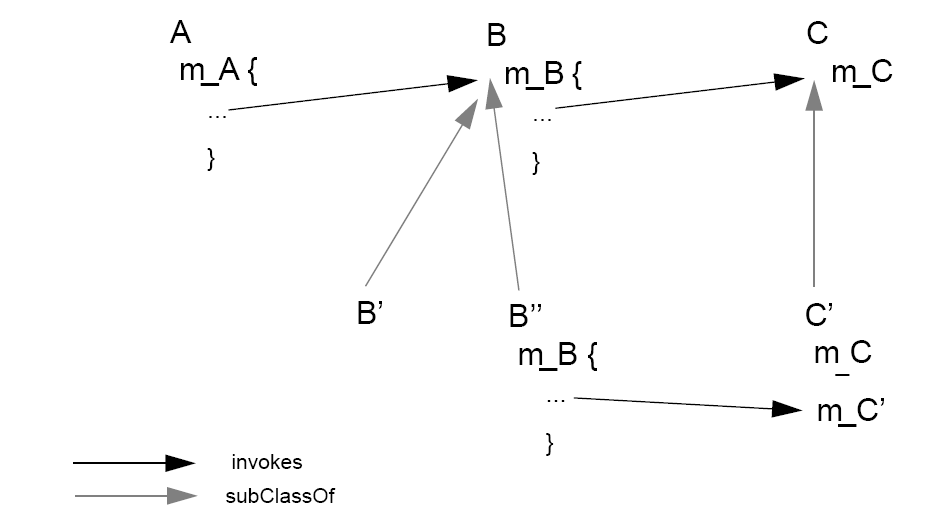
**◎Extension （扩展）：**the subclass only adds methods and instance variables but neither modifies nor refines any of the inherited ones

* If a client using a parent ref, only the parent is needed
  + 1 connection

**继承如何减少耦合？？**

Remember: in Refinement and Extension inheritance coupling, the interaction coupling between super-class and subclass is ignored

* + Object-level coupling remain, but class-level coupling reduced

****

**A-B B-C B’’-C’ A-B’’ B-C’ A-C A-C’**

**类级别：**

**●CBO（Coupling between object classes）:**

A count of the number of other classes:

* + which access a method or variable in this class, or
  + contain a method or variable accessed by this class
  + Not including Inheritance

Want to keep this low

**●RFC(Response for a class):**

方向很重要，是对别的类有很多引用还是自己被别的类引用

一个对象接收到信息时实际上最终引起的方式的执行数目，不包括继承

keep this low

**●MPC(Message passing coupling):**

一个类中定义在局部的send语句数目。

The number of send statements sent out from a class may indicate how dependent the implementation of the local methods is on the methods in other classes.”

keep this low

**●DAC(Data abstraction coupling):**

The number of attribute having an ADT type dependent on the definitions of other classes（用了多少类作为自己的数据类型）

keep this low

**●Ce and &D (efferent and afferent coupling)传出和传入耦合：**

* + Ca: The number of classes outside this category that depend upon classes within this category.
  + Ce: The number of classes inside this category that depend upon classes outside this category

keep this low

**●DIT（Depth of the Inheritance tree）：**

* + the maximum length from the node to the root of the tree
  + as DIT grows, it becomes difficult to predict behavior of a class because of the high degree of inheritance
  + Positively, large DIT values imply that many methods may be reused

keep this low

**●NOC(Number of children):**

* + count of the subclasses immediately subordinate to a class
  + as NOC grows, reuse increases
  + as NOC grows, abstraction can become diluted
  + increase in NOC means the amount of testing will increase

## **对给出的例子：考试方式**

* 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。

（这段代码对其他代码有什么样的connection，分析connection的类型：包括 共享的、公共环境，模块间的、方法调用的；从面向对象角度有对象之间的，隐藏的，聚合的）

# 3信息隐藏

## 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。

（什么是信息隐藏，信息隐藏的应用。告诉你要做一件什么样的事，用上述两种方法设计）

可以参考ppt2的KWIC例子（两种模块化的方式）（找了半天没找到什么相关内容）

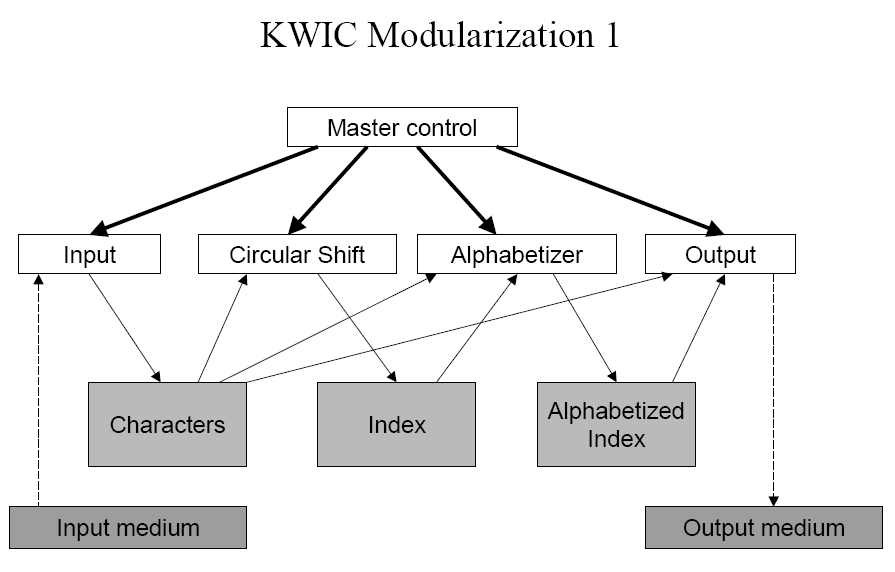
Information Hiding 理解:

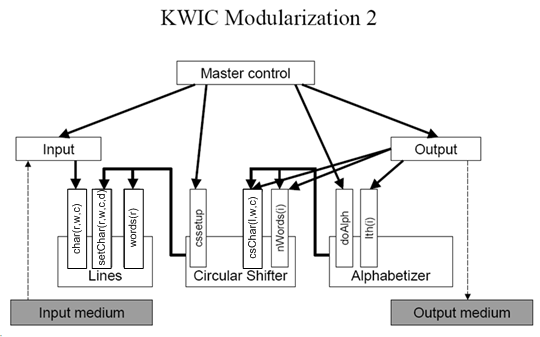
信息隐藏是为了让系统的模块可以独立并行的开发,既可以独立的被测试,优化,修改和理解.他减轻了维护的负担,且做到了调用某一模块时不影响其它模块,提到了软件的可重用性,同时降低了构建大型系统的风险.

每个模块隐藏模块design decision的实现(特别是设计的可变性)

所有的design decisions are 相互独立,不依赖.

两种方式进行模块化的Example:KWIC





**两种模块化的方法**

* 模块化1

处理流程中每个重要步骤 作为 一个模块

* 模块化2

根据 信息隐藏 原则:

每个模块有一个或多个secret

Lines

* + - * + how characters/lines are stored

Circular Shifter

* + - * + algorithm for shifting, storage for shifts

Alphabetizer

* + - * + algorithm for alpha, laziness of alpha

每个模块有自己的design decision 并且其他模块不可见

**两种模块化的对比:**

**1总体比较:**

* + Note: both systems might share the same data structures and the same algorithms
  + Differences are in the way they are divided into work assignments
  + Systems are substantially different even if identical in the runnable representation
    - Possible because the runnable representation is used only for running
    - Other representations are used for
      * + Changing
        + Documenting
        + Understanding

**2可变性比较:**

* Design decisions that may change
  + Input format: (1, 1)
  + All lines stored in memory: (all, 1)
  + Pack characters 4 to a word: (all, 1)
  + Make an index for circular shifts rather than store them: (3,1)
  + Alphabetize once, rather than either: (3,1)
    - Search for each item as needed
    - Partially alphabetize, partially search

**3独立开发比较:**

* 模块化1
  + Must design all data structures before parallel work can proceed
  + Complex descriptions needed
* 模块化2
  + Must design interfaces before parallel work can begin
  + Simple descriptions only

**4可理解性比较:**

* 模块化2更好
  + Parnas subjective judgment
  + Less coupling among modules
    - Programming to interfaces

总结:

## 对简单的应用，解释其Module Guide。（什么是ModuleGuide，每个部分的作用）

**Module Guide**

当系统有大量交互的模块时（一个有上百个模块的复杂系统）,由许多模块行为构成的系统的集体行为很难被理解。

解决方法是创建一个层次结构的文档——Module Guide。它提供一个好的文档支持，从而降低了系统维护的成本。

**对于每个模块，module guide应该说明：**

（secret） 1每个模块所包含的Secrets。Secrets包括primary和secondary

Primary secret是模块内部隐藏的信息

Secondary secret是实现模块的实现决策

（role） 2每个模块在整个系统中扮演的角色。

（facilities）3每个模块实现的功能。

**作用**

如果没有module guide，职责将无法追溯到确定的一个或两个模块；module guide帮助我们在设计是相对的减少失误。

Module guide帮助系统的开发人员和维护人员更容易的找到被change所影响的模块。

**其他文档：**

Module Interface Document : 精确完整的描述每个模块的接口

Module Design Document : 描述每个模块实现的内部设计

Requirements Document : Module Guide根据它查找相关的changes

# 全局分析步骤

* 分析影响因素：（约束 和 质量因素）3步

1识别和描述因素

2描述因素，描述factors的灵活性和可变性特征

3分析这些因素的影响

* 开发策略：3步

1 识别问题和影响因素

2 开发解决方案solution和具体策略strategies

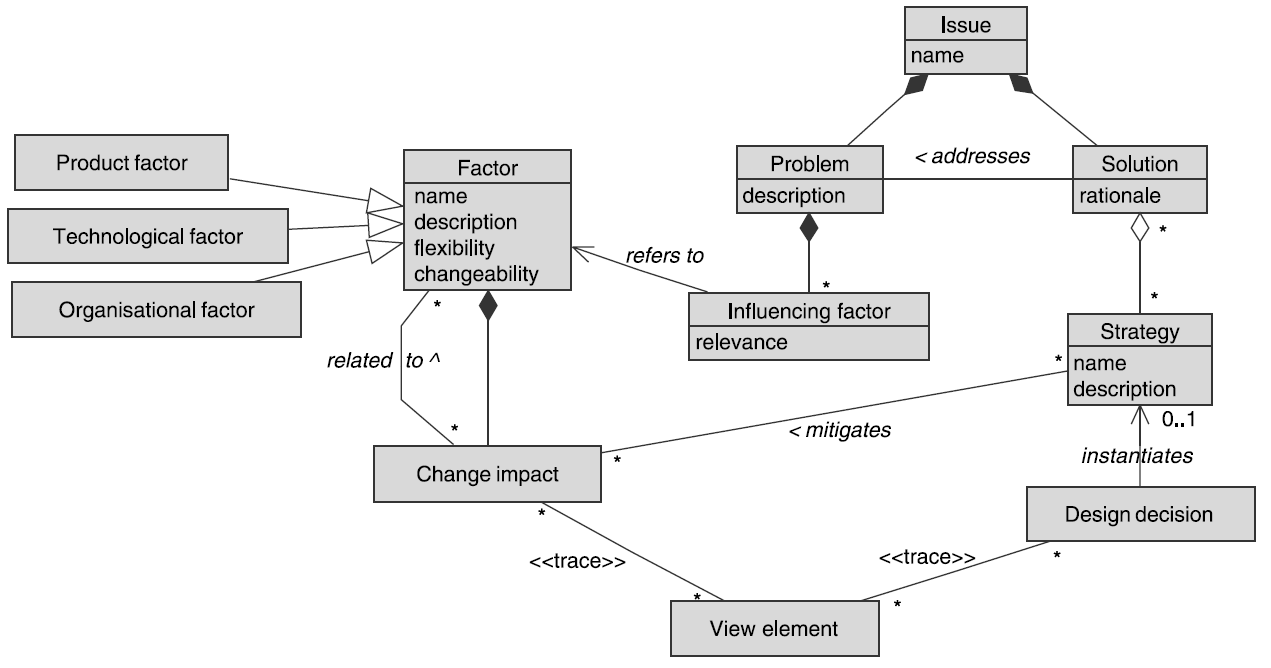
3识别相关策略

**因素factor（三种）**

* **Product factors（**Function features，User interfaces，Performance，Security，Error tolerant）
* **Organizational factors**（Building or Buy ，Staff，Process，Tools，Cost，Time）
* **Technological factors**（Hardware，Software，Network，Platform，Framework，Standard）

**全局分析的产物：**

1. Factor Table
2. Issue Card
3. Strategies

****

**全局分析核心思想（4点）**

1 重视体 系结构设计需求ADR。Global Analysis: Analyzing Factors）

2用 系统解决方案策略SSS将 体系结构设计需ADR 同最终的 体系结构模型（the structural models of the software architecture ）联系起来。（Global Analysis: Develop strategies）

3 整合strategies，由SSS 开发得到 SA：可以参考4+1view模型 （Making decision to each view in according to SSS ）

4 ADR ， SSS 将支持开发过程中的其它部分。

# **6 根据分析类图，建立基本的设计类图（给分析类图，给要求，做设计）**

——把参与者转换为接口类

——添加参与者类：记录与参与者有关的数据

——添加启动类：程序开始的位置

——转换或添加控制器和协调器：控制并协调程序活动的类

——添加实现数据类型的类：概念模型中包括的很多结构和行为的类型，将他们作为数据类型来实现

——转换或添加容器类：容纳对象的集合

——转换或添加工程设计关系：泛化关系、接口、接口实现关系、依赖关系等。



# 7 有几种控制流处理方式？进行比较

**（集中，委托，分散，特点区别和比较；把7和8合起来，比如说给一个不好的顺序图是集中式控制的，结合grasp，重新建立一个委托式的。）**

1. **有几种控制流处理方式？进行比较（集中，委托，分散，特点区别和比较；把7和8合起来，比如说给一个不好的顺序图是集中式控制的，结合grasp，重新建立一个委托式的。）**

**控制器在交互作用设计中具有重要的地位，因为它们在协作中是中心角色。它们通常是启动和结束交互作用，把任务委托给其他组件并返回结果的组件**

**集中式：少数控制器做出所有重大的决定。非控制器组件只是保存数据或执行简单功能。**

**优点：易于判定决定是哪里做出来的，易于保证正确性（逻辑思路明确）**

**易于修改决策过程**

**缺点：控制器可能变得太大，太复杂，难以理解，维护，测试等等。这样的控制器被称为膨胀式控制器。其内聚性不高，而且是大型模块，这违背了两条模块性设计原则**

**控制器可能把其他组件视为数据仓库，只是在其中存储或者从中检索数据。这样往往会增强耦合性，并破坏信息的隐藏，因此违背了另外两条模块性设计原则**

**适用情况：仅当程序只需要做出少量的决定时才应该使用集中式控制样式**

**规则：**

**在交互作用设计中，避免使大多数消息都源自同一个组件。**

**使组件保持小型化。**

**确保不把操作职责都分配给少量组件。操作职责的集中是集中控制样式的特征。**

**确保操作职责与数据职责一致。**

**集中式控制的顺序示意图：**

：

**委托式：决策权分布在整个程序中，控制器做出事关全局的决定，并协调其他组件的活动，但把较低级别的决策委托给其他组件来做出。即只关心发起后面的决策并不关心，类似于中介代理。**

**优点：控制器只与较少的组件耦合，程序的总体耦合性降低。在委托职责的时候，控制器不需要知道那些与受托组件协作的组件，因此降低了耦合性。**

**信息得到更好的隐藏。与从组件中获取数据，修改数据之后再返回给组件不同，委托控制式鼓励组件自己修改自己的数据**

**使程序易于分层。正如我们以后将看到的那样，分层体系结构样式是一种非常重要和强大的组织程序模块的方式。委托控制样式使分层的组织更易于实现。**

**适用情况：委托控制样式确实没有任何缺点，这是首选的控制样式，尤其适合于面向对象的系统。**

**规则：使组件把尽可能多的低级任务委托出去。组件负责一些高级任务，而完成高级任务通常需要完成若干低级任务。换句话说，可以把功能分解为更简单的功能。这些低级任务通常是通过协作完成。**

**分散式（例如SOA&EJB）：决策权广泛散布在整个程序中，只有很少或者没有组件做出自己的决策，识别出哪些组件式控制器是困难的。**

**在这样的设计中，有很多容纳少量数据，拥有少量小型操作的小型组件。每项任务都必须通过数十个交互作用进行跟踪。**

**缺点：理解控制流程很难。必须跟踪大量的消息，才能弄清某项任务时如何完成的。这样的设计既难以理解，又非常难以修改。**

**当把组件分割得太小时，他们往往不能独立做任何事情，结果就是耦合性增强。**

**隐藏信息是困难的，因为每个组件的工作过程严重依赖其他组件的实现方式**

**内聚性差（低内聚）**

**有些模块化原则不能被满足。**

**规则：避免每个组件都需要发送很多信息的交互（如果每个对象都需要发出很多消息，则表明控制样式过于分散）**

**Demeter规则：**

**对象的操作只应该给下列实体发送消息：**

**● 对象obj;**

**●Obj的属性；**

**● 该操作的实参；**

**● 作为该操作实参或obj属性属性的某个集合的元素；**

**●该操作创建的对象；**

**●全局类或全局对象。**

**例子应用：查看书上335**

**（6和8结合，分析类图，告诉你有哪些职责要分配到分析类图里面去，让你用grasp方法来做一个设计类，必须准确的告诉每一个方法是什么类。grasp不光是方法的分类，有的时候会出现新的类）**

# 8 **GRASP模式**