第八章 设计工程

一、设计概述

1、什么是设计？什么是设计模型？

- 将问题转化成解决方案的创造性的活动

- 对解决方案的的描述

2、两种设计（不同的观察视角）

- Conceptual Design概念设计
- Technical Design技术设计

概念设计 在用户的视角

- 告诉顾客系统将要做什么
- 回答：（概念设计要回答的问题）
  - Where will the data come from数据来自何地?
  - What will happen to the data in the system系统中的数据会发生什么情况?
  - What will the system look like to users系统对用户来说看起来象什么?
  - What choices will be offered to users能向用户提供什么选择?
  - What is the timing of events事件的时间安排如何?
  - What will the reports and screens look like报告和屏幕看起来像什么?

- Characteristics of good conceptual design好的概念设计的特征
  - in customer language with no technical jargon用顾客语言而不是技术术语
  - describes system functions描述系统的功能
  - independent of implementation独立于实现
  - linked to requirements与需求相关

概念设计的任务与步骤

概念设计确定：

- 软件系统的结构
- 各模块功能及模块间联系(接口)

概念设计的过程 ：

(1)设想可能的方案

(2)选取合理的方案

(3)推荐最佳方案

(4)功能分解

(5)设计软件结构

(6)数据库设计

(7)制定测试计划

(8)编写文档

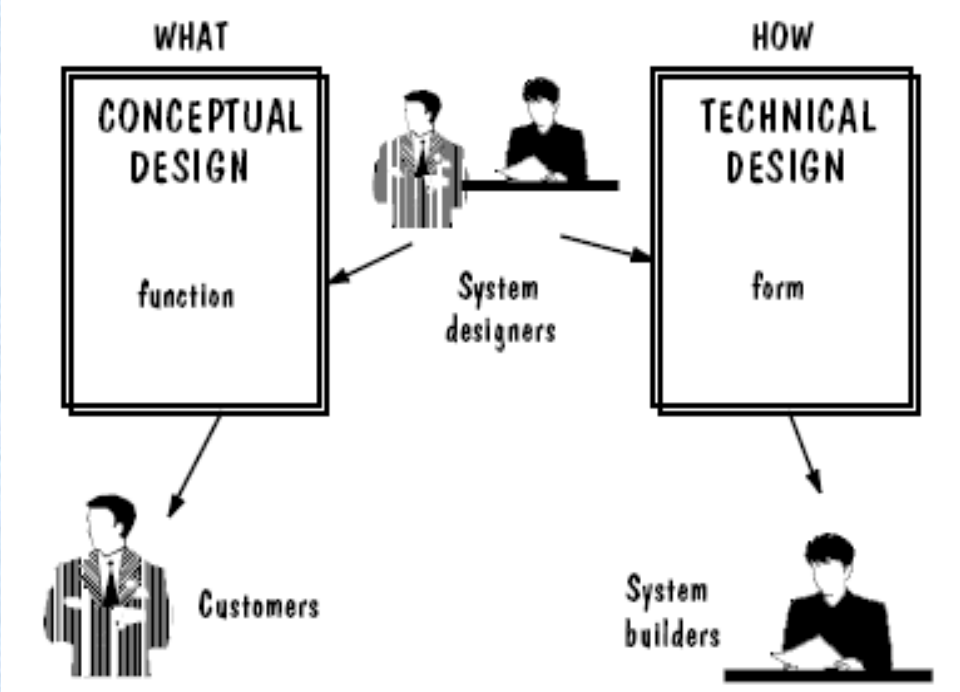
(9)审查与复审

技术设计

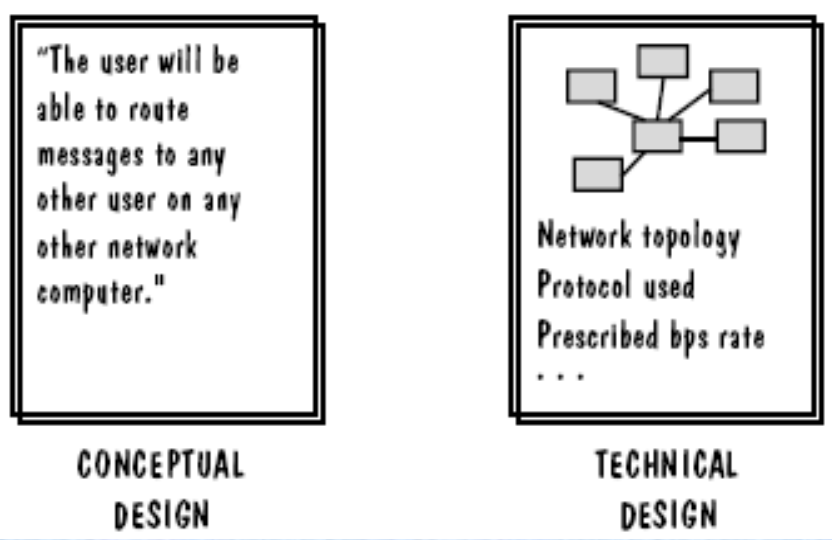
- Tells the programmers what the system will do告诉程序员系统将做什么
- Includes包括：（包括对以下条目的描述）
  - major hardware components and their function主要硬件组件和功能
  - hierarchy and function of software components软件组件的层次和功能
  - data structures数据结构
  - data flow数据流

技术设计主要任务

- 编写技术设计说明书：
  - 确定每个模块的算法，用工具表达算法的过程，写出模块的详细过程性描述。
  - 确定每一模块的数据结构。
  - 确定模块接口细节。
- 技术（详细）设计是编码的先导。



概念和技术设计



设计文档间的差异

3、五种创建设计的方法

- Modular decomposition模块分解
  - 将功能分配给组件
- Data-oriented decomposition面向数据的分解
  - 基于外部数据结构
- Event-oriented decomposition面向事件的分解
  - 基于系统必须处理的事件和事件改变系统的状态信息
- Outside-in design由外而内的设计
  - 基于系统的用户输入（属黑盒方法）
- Object-oriented design OO设计
  - 确定对象的类和它们之间的相互关系

模块化

- 模块分解结果形成的组成部分称*模块*或*组件*
- 模块化
  - 当系统的每一个功能正好由一个模块执行
  - 当每个模块的输入输出被*明确定义*

4、三种设计层次

- Architecture体系结构: 系统性能与系统组件关联起来
- Code design代码设计:为每个组件详细说明算法和数据结构
- Executable design执行设计:设计的最低级，包括内存分配、数据格式和位组合

5、软件体系结构的风格

三个方面：

- 组件（模块）
- 连接器（组件之间的联系）
- 组件（模块）组合的限制条件

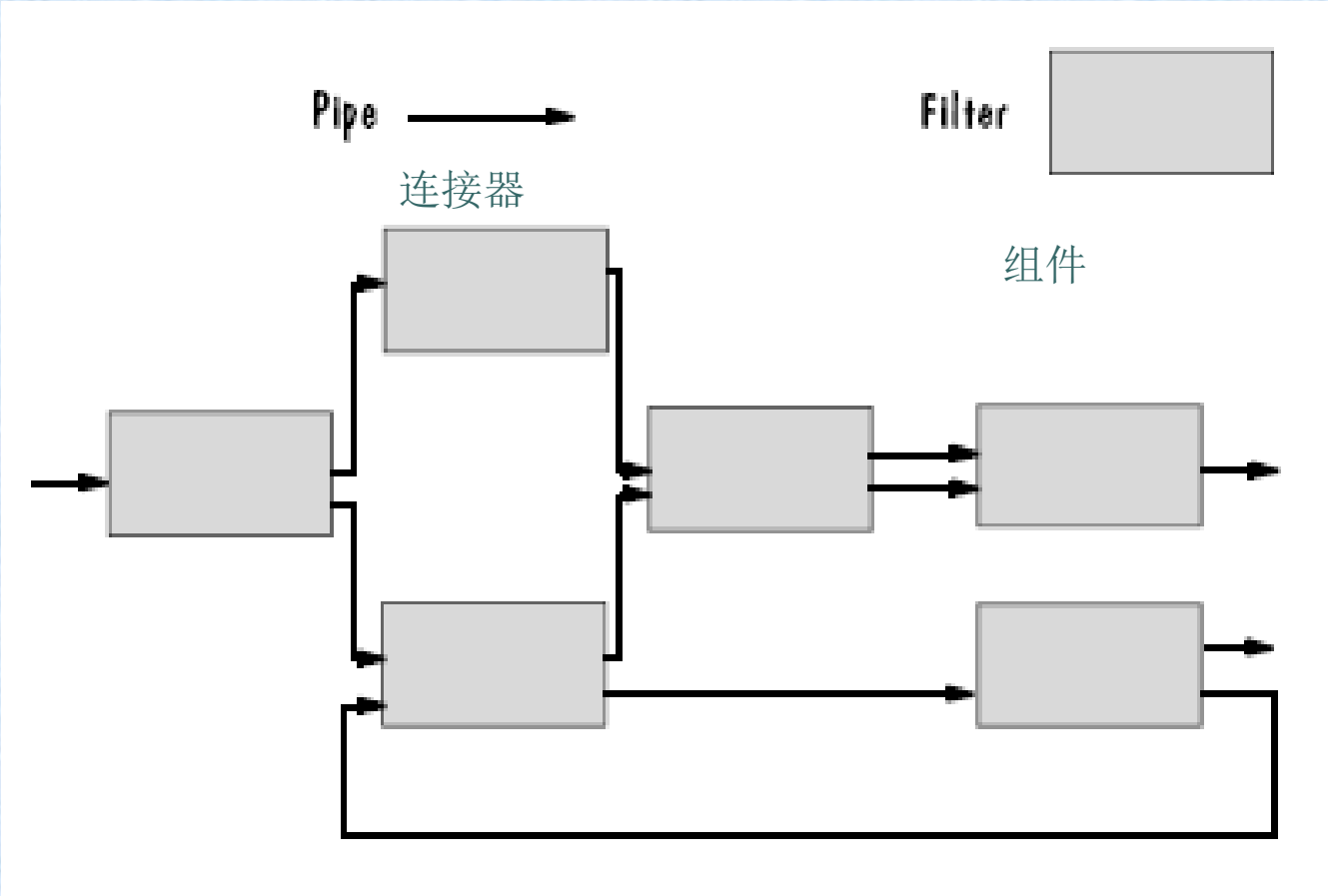
设计风格

- 管道和过滤器
- OO设计
- 隐含调用
- 分层
- 解释器
- 过程控制

pipe&filter
oo
call
layer

。。。。。。。。。。。。

管道和过滤器



- 性质
  - 关系表示明确
  - 复用、修改、模拟容易
  - 允许并发执行过滤器
- 局限性
  - 更适合批处理（不适合交互式处理）
  - 数据流之间需要对应
  - 类似过滤器潜在的重复操作执行

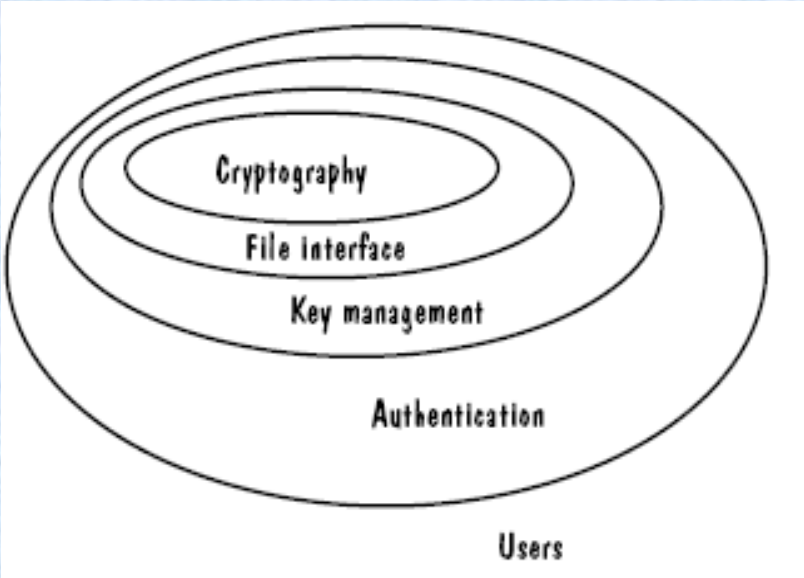
隐含调用（事件驱动）

某个组件宣告事件，其他组件处理事件。利用注册程序处理事件

- 优点
  - 易于复用其他系统组件
  - 对用户界面尤其有用
- 缺点
  - 不能确定某个事件的响应.
  - 测试系统很难.

Layering分层

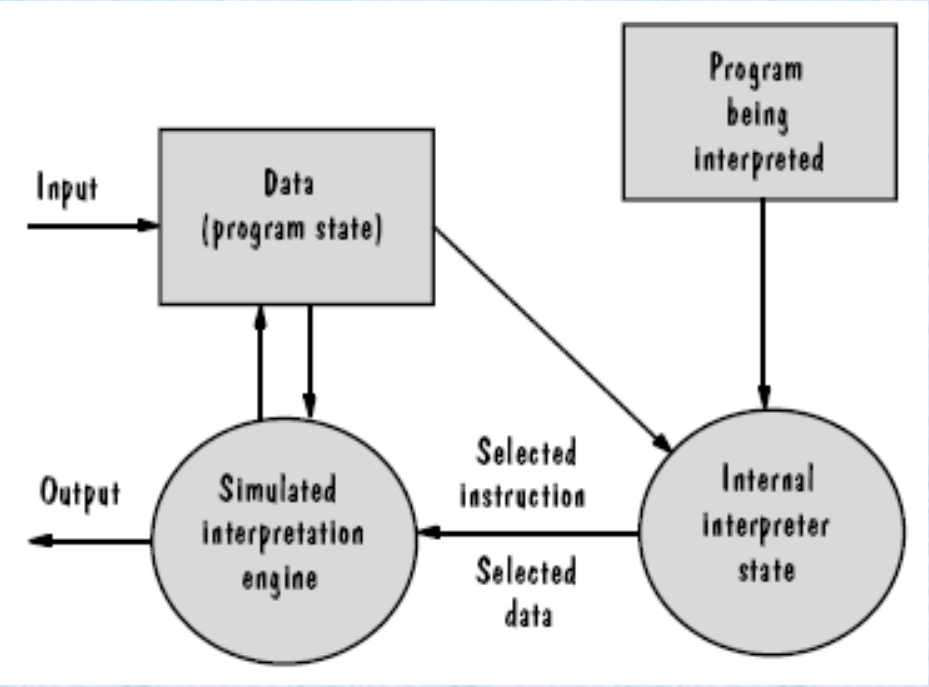
各层分等级，每层为它的外层提供服务



- 优点
  - 表示不同的抽象层次
  - 对层的修改通常只影响相邻的两层
- 缺点
  - 需求阶段定义多层抽象很困难
  - 性能问题?

最大问题：性能

解释器



- 解释器读入字符串并将它们转换成可以执行的实际代码.
- 一般用来构建虚拟机

6、重要的设计问题

- 模块性与抽象层次
- 协作设计
- 设计用户界面
- 并发
- 设计模式与复用

7、优秀设计的特征

属于模块化范围

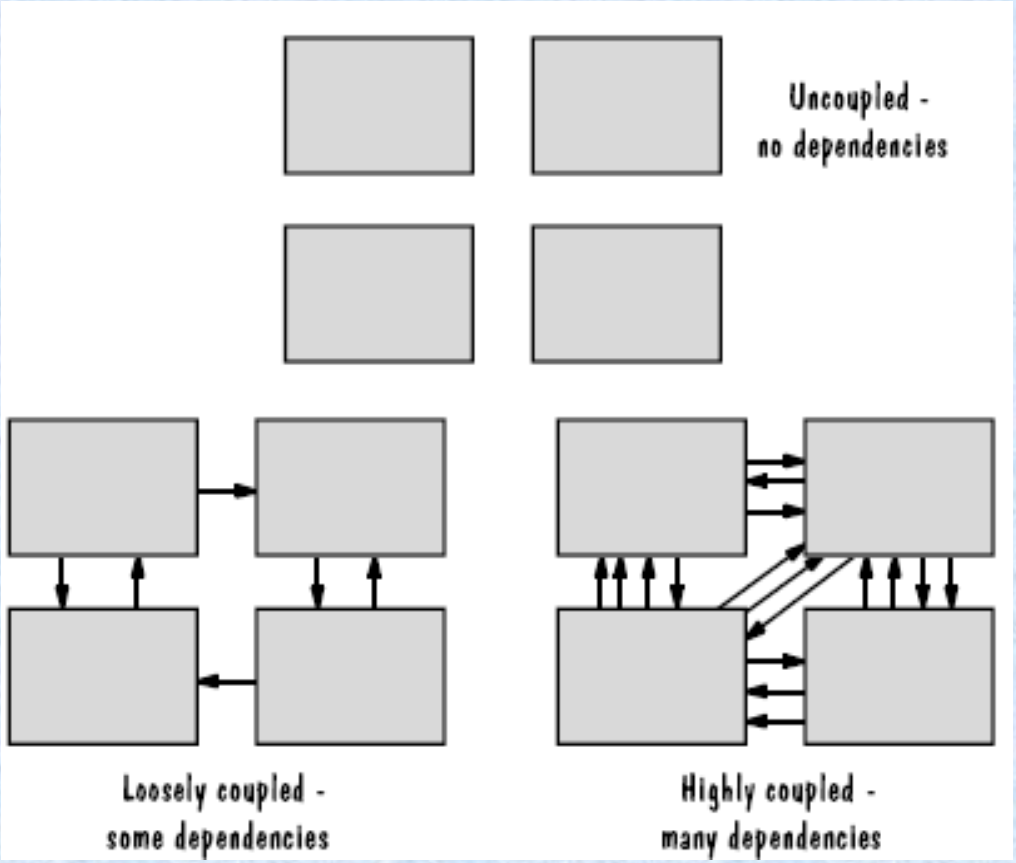
- 组件独立性
  - Coupling耦合性
  - Cohesion内聚性
- 例外设计与处理
- 防错和容错

7.1组件独立性

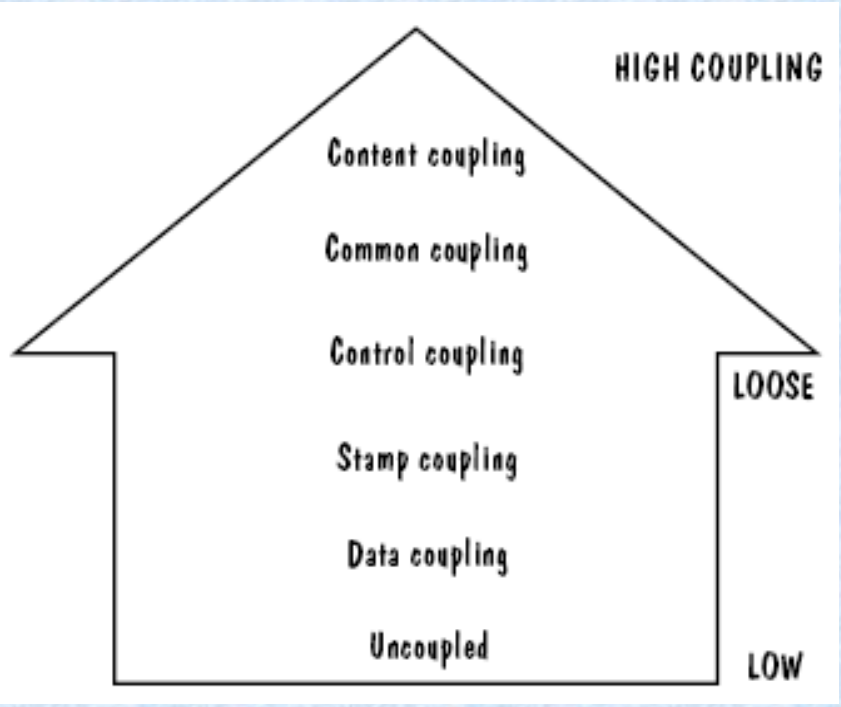
自主性：独立完成功能不依赖其他组件

- 组件独立是模块化、抽象、信息隐蔽和局部化的直接结果。
- 含义：一个模块具有独立功能而且和其它模块之间没有过多的相互作用
- 意义：独立的模块容易开发（规模小，接口简单）；独立的模块容易测试和维护；有效阻断错误传播（Ripple effect“涟漪效应”）
- 度量标准：内聚和耦合

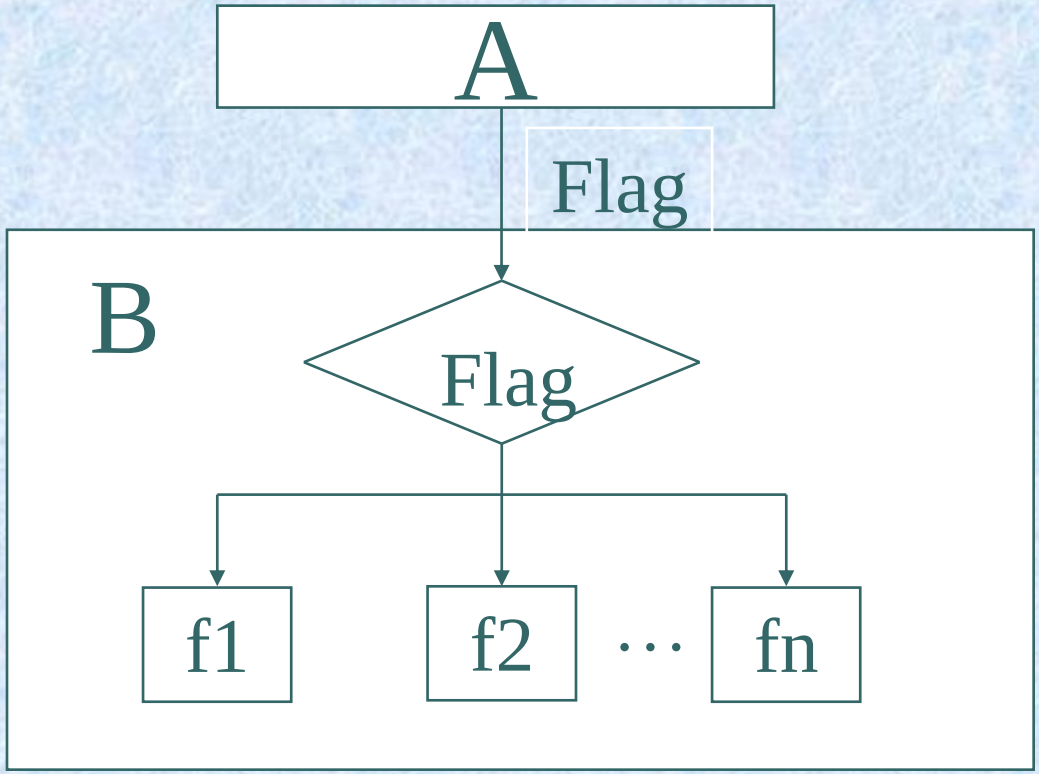
耦合性



- 定义
  - 指块间联系，即程序结构中不同模块之间互连程度
- 耦合强弱取决于
  - 模块间接口的复杂程度
  - 调用模块的方式



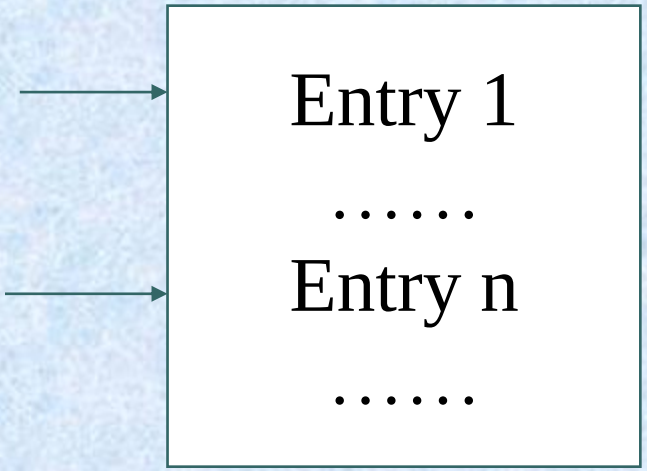
- 内容耦合：一个模块可以直接操作另一个模块的数据（如go to 语句的使用）
- 公共耦合：全局结构类型的数据
- 控制耦合：模块间传递的是诸如标记量的控制信息
- 标记耦合：参数传递的是诸如结构类型的数据
- 数据耦合：参数传递的是一般类型的数据
- 非直接耦合：通过上级模块进行联系，无直接关联



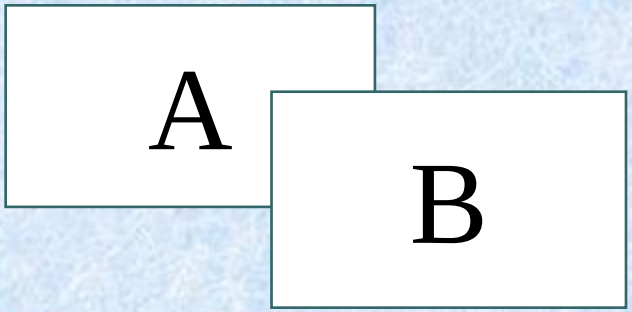
控制耦合

进入另一模块内部

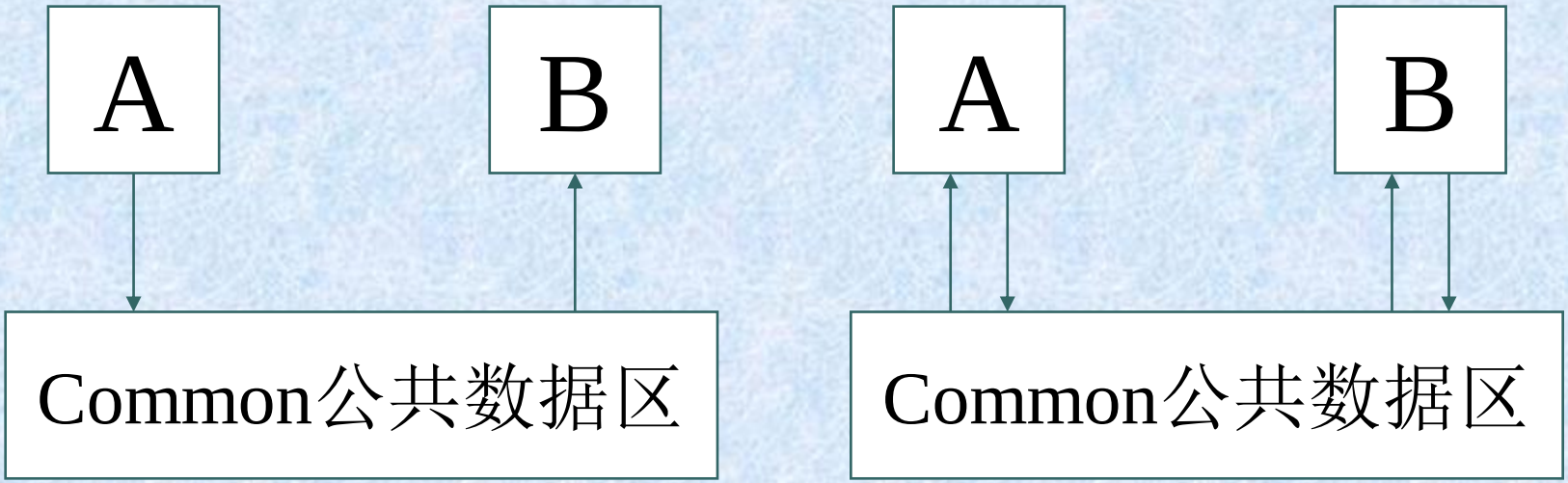
多入口模块



模块代码重叠

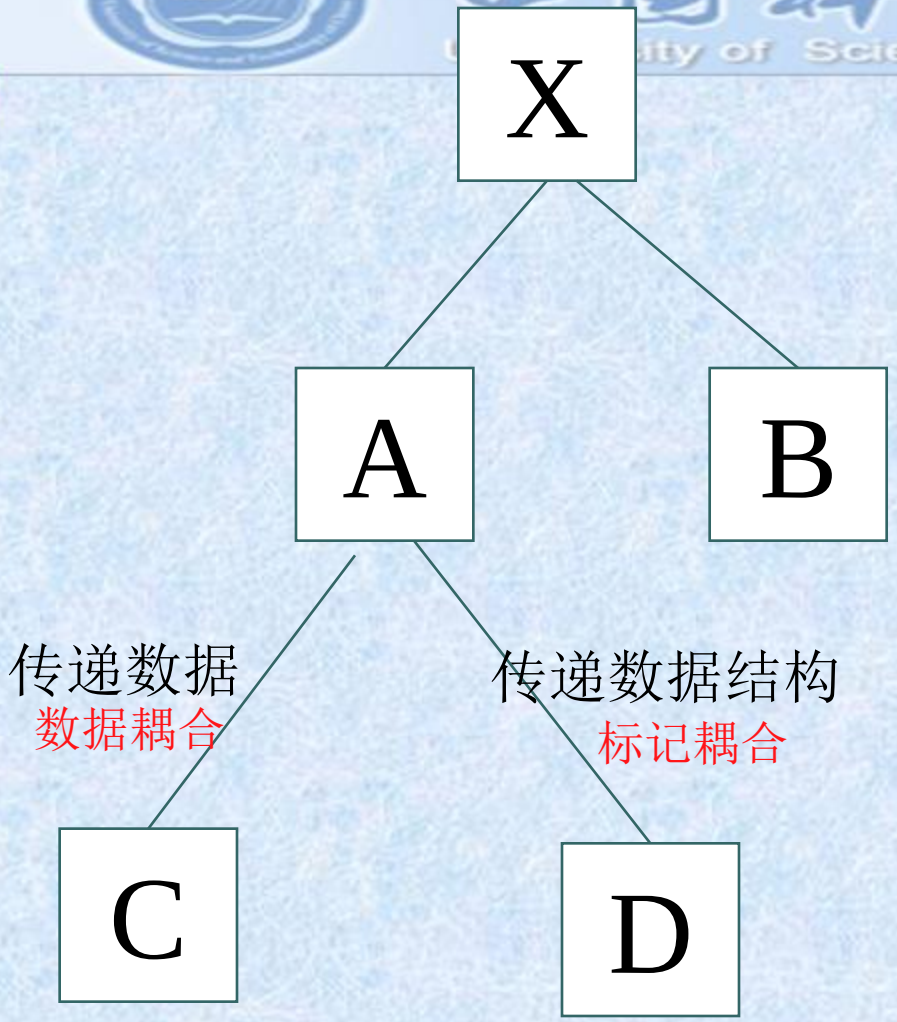


内容耦合



松散的公共耦合

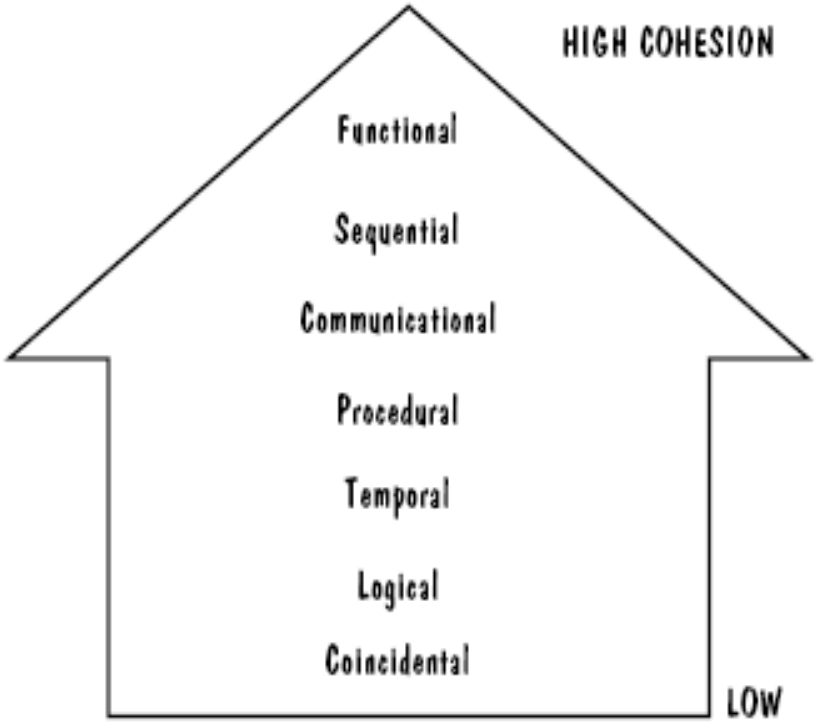
紧密的公共耦合



AB为非直接耦合
AC为数据耦合
AD为标记耦合

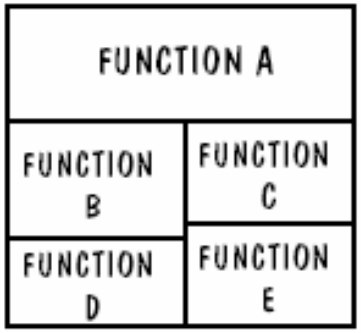
Cohesion内聚

- 块内联系或模块强度，指模块内各个成分（元素）彼此结合的紧密程度，即模块内部的聚合能力。
- “理想的模块仅仅做一件事”。



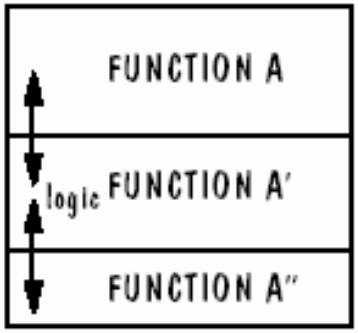
顺序的前后有输入输出，相互需要的，过程不需要

- 功能性内聚：一个功能一个模块，块内各成分属于一个整体
- 顺序内聚：模块内各个组成部分都是与一个功能密切相关，并是顺序执行的。一般是一个成份的输出就是下一个成份的输出
- 通讯内聚：模块内的各个成份都使用同一输入数据，或产生同一输出数据，即借公用数据而联系在一起
- 过程内聚：块内成份必须按照特定次序执行
- 时间内聚：因执行时间一样或顺序排列而把几个任务安排一个模块，如把“变量赋初值”、“打开文件”等完成各种初始化任务安排在一个模块
- 逻辑内聚：块内任务间在逻辑上相似或相同，例如求某班的平均分和最高分，因其输入和输出相同而安排在一个模块内完成。
- 偶然（巧合）内聚：一个模块所完成的几个任务之间关系松散，互不相关。主要是为了避免重复书写而把重复的代码集成到一个模块内。



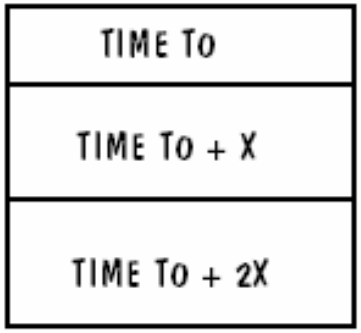
COINCIDENTAL 巧合

Parts unrelated



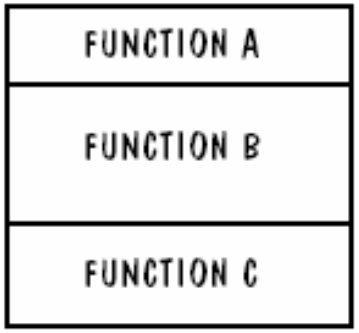
LOGICAL 逻辑

Similar functions



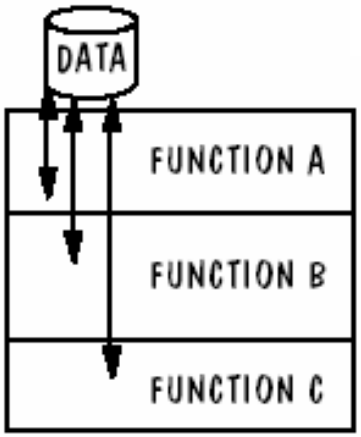
TEMPORAL 时间

Related by time



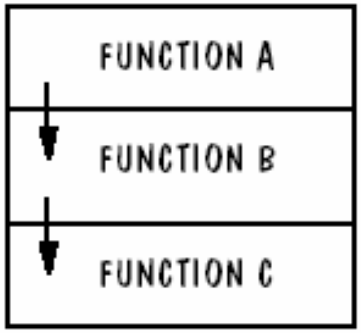
PROCEDURAL 过程

Related by order of functions



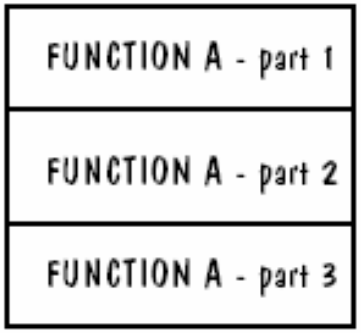
COMMUNICATIONAL通信

Access same data



SEQUENTIAL 顺序

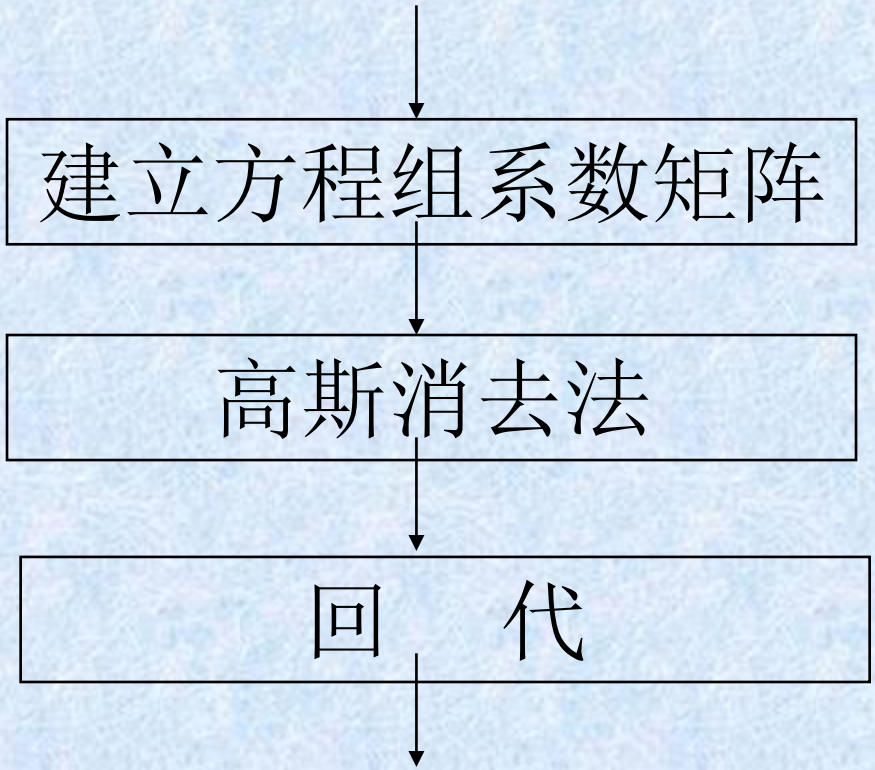
Output of one part is input to next



FUNCTIONAL 功能

Sequential with complete, related functions

示例



- 三个方框组成一个模块，则是过程化模块；
- 前两个方框组成一个模块，则是顺序性模块；
- 如一个方框就是一个模块，则是功能性模块。

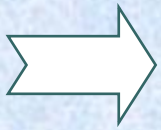
偶然性内聚

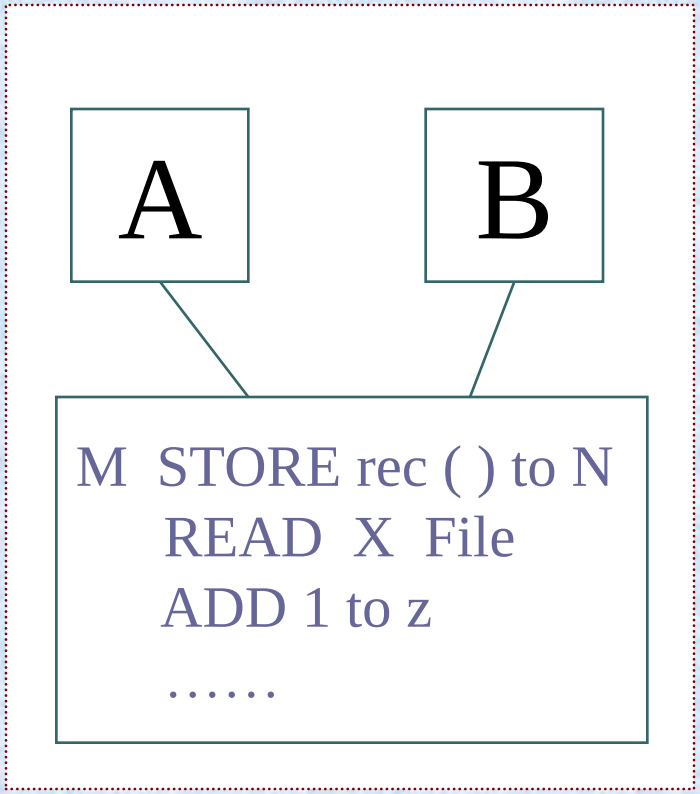
A

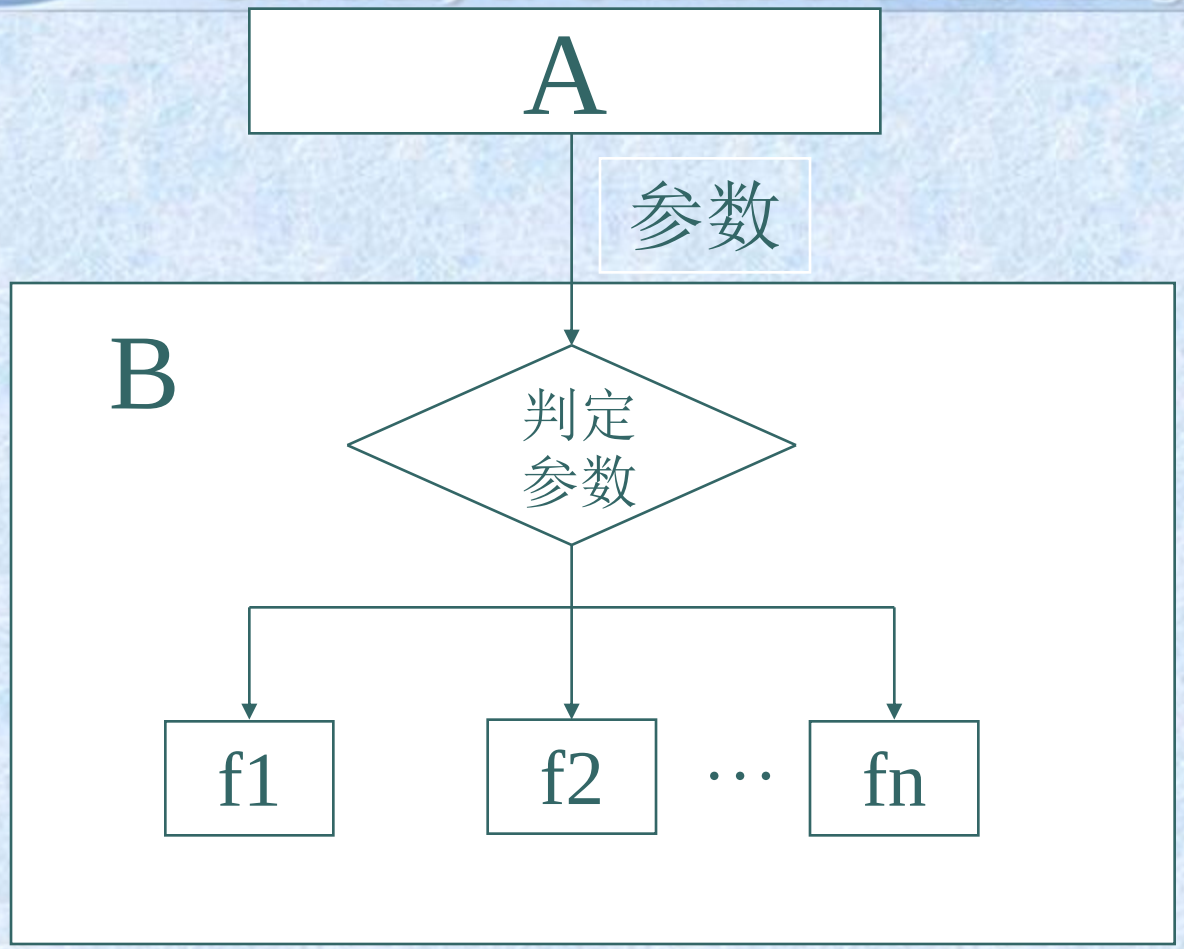
……
STORE rec ( ) to N
READ X File
ADD 1 to z
……

B

……
STORE rec ( ) to N
READ X File
ADD 1 to z
……

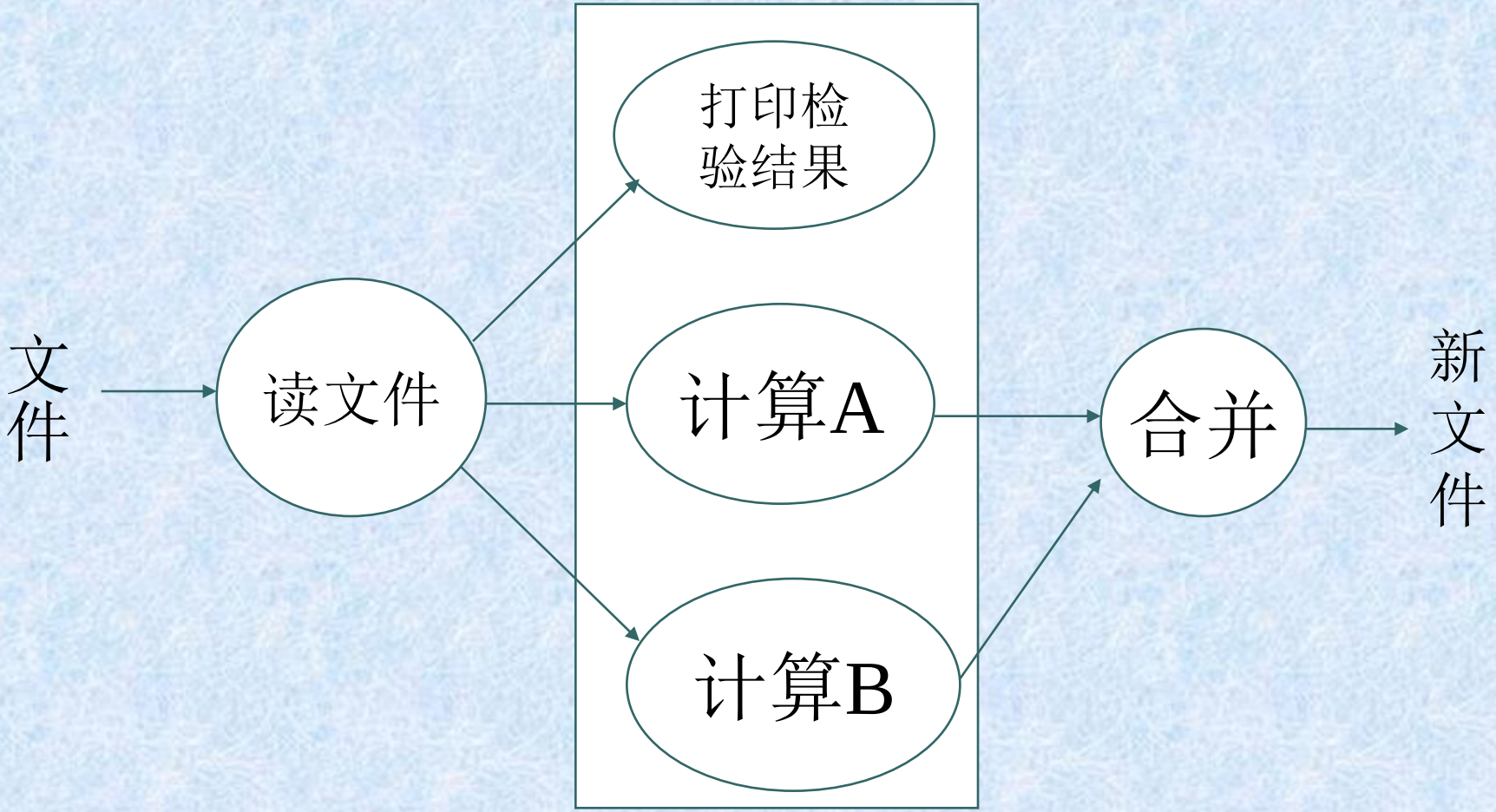






模块B属于逻辑内聚

加工记录模块



通信内聚

启发式规则的应用

- 提高模块独立性
- 设计规模适中的模块
- 深度、宽度、扇入、扇出适中
- 模块的作用域应该在控制域之内
- 降低接口复杂性
- 设计单入口和单出口的模块
- 设计功能可以预测的模块

扇出：一个模块控制别的模块数量，不超过9个，不少于3个
扇入：一个模块被几个控制，越大越好

说明：启发式规则是一种经验规律，对改进设计和提高软件质量具有重要的参考价值，但不要过分拘泥于这些规则。

提高模块独立性

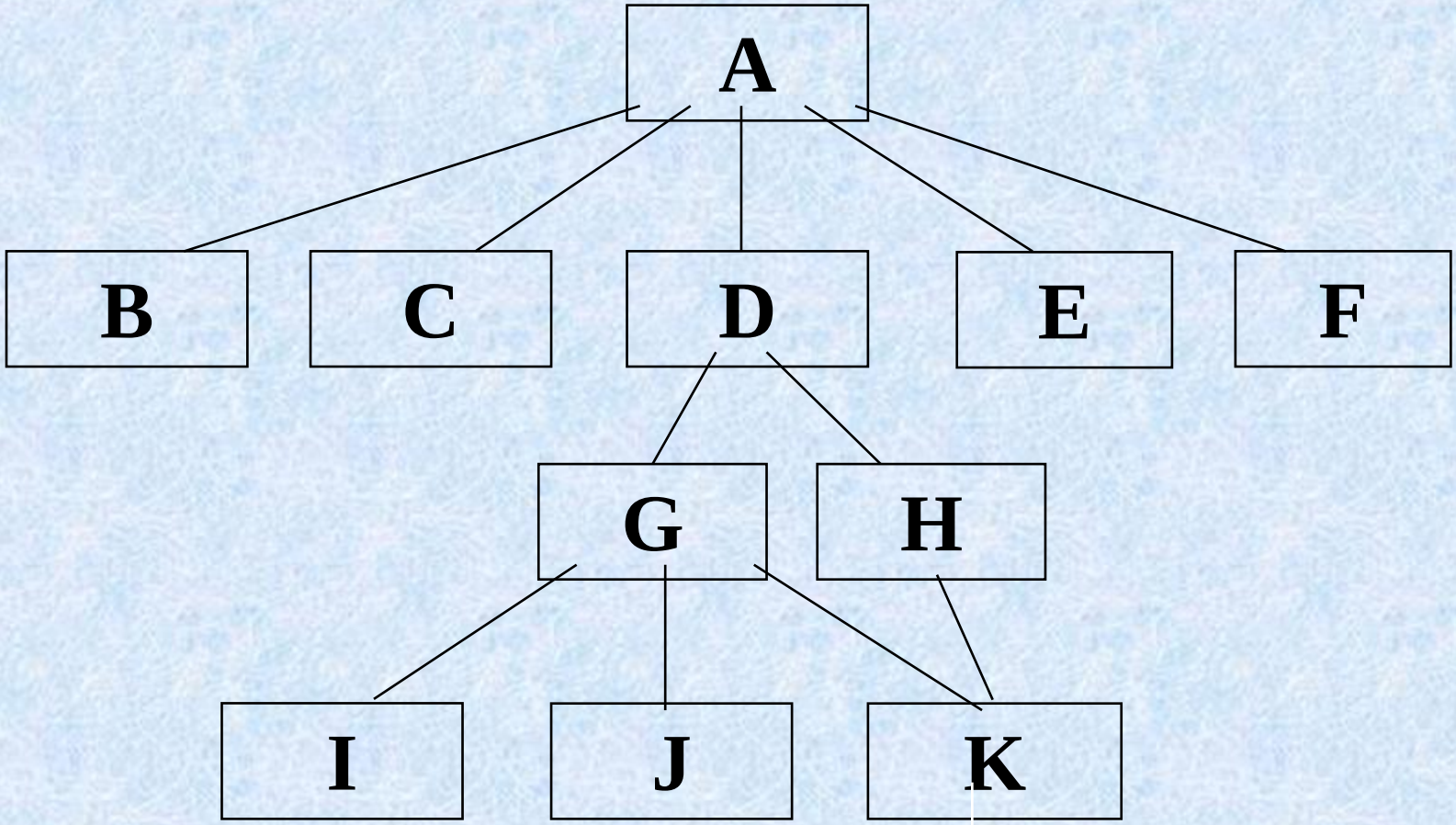
- 模块独立性是划分模块的最高准则。
- 高内聚，尽量一个模块一个功能；
- 低耦合，避免“病态连接”；
- 降低接口的复杂程度；
- 综合考虑模块可分解性、模块可组装性、模块可理解性、模块连续性和模块保护（因修改错误而引起的副作用被控制在模块的内部）等。

设计规模适中的模块 模块不能太大

- W. M. Weinberg的研究表明：如果一个模块长度超过30条语句，其可理解性将迅速下降；
- F. T. Baker：最好控制在50行左右，能够打印在一张纸上。
- 由于模块独立性是最高原则，对于一个设计合理的功能性模块，即使长达千句或小到几行，也是允许的。
- 分解模块不应该降低模块独立性。

深度、宽度、扇入、扇出适中

- 深度：软件结构中控制的层数。一般而言它与系统的复杂度和系统大小直接对应。
- 宽度：软件结构中同一个层次上的模块总数的最大数。
- 扇出：一个模块直接控制（调用）的模块数目。扇出过大说明模块过分复杂；过小也不好，不利于系统平衡分解，3到9为宜。
- 扇入：一个模块的扇入是指直接控制该模块的模块数目。扇入越大说明共享该模块的上级模块越多。
- 整个系统结构呈现“椭圆外型”。



如图的系统中：深度为4；宽度为5；模块A的扇出为5，扇入为0。模块K的扇出为0；扇入为2。

模块的作用域应该在控制域之内

- 控制域：控制范围，是包括模块本身以及所有下属模块（直接调用模块和间接调用模块）的集合。模块D的控制域为D,G,H,I,J,K。
- 作用域：作用范围，它是一个与条件判定相联系的概念。是受该模块内一个判定影响的所有模块的集合。如果模块D中有一个条件判定仅仅影响到模块G和H，其作用域为G, H, I, J, K，作用域在控制域内。如D的判定影响到E，通常需要在D中为判定结果设置一个标记，并把这个标记通过上级模块A传递给E，导致控制耦合。
- 两种改进方法：判定上移和在作用域但不在控制域的模块下移。

降低模块间接口复杂性

- 尽量少使用go to语句，避免病态连接和内容耦合。
- 注意全局变量的使用，控制外部耦合和公共耦合的使用。
- 将数据结构的传递改成数据传递，例如：求一元二次方程根的模块quad_root(table, x)中，利用系数数组table和根数组x进行参数传递。如果将其改为直接的系数和根传递，即quad_root(a, b, c, x)，则特征耦合→数据耦合。

7.2例外识别和处理

- 典型的例外
  - 无法提供某种服务
  - 提供了错误的服务或数据
  - 破坏性的数据
- 防御性设计不容易！

7.3 防错和容错

对于软件失效后果特别严重的场合，如飞机的飞行控制系统、空中交通管制系统等必须采用容错设计。

（1）N版本编程法

（2）恢复块技术

（3）防卫式程序设计

软件容错方法有很多，要根据实际情况进行选择

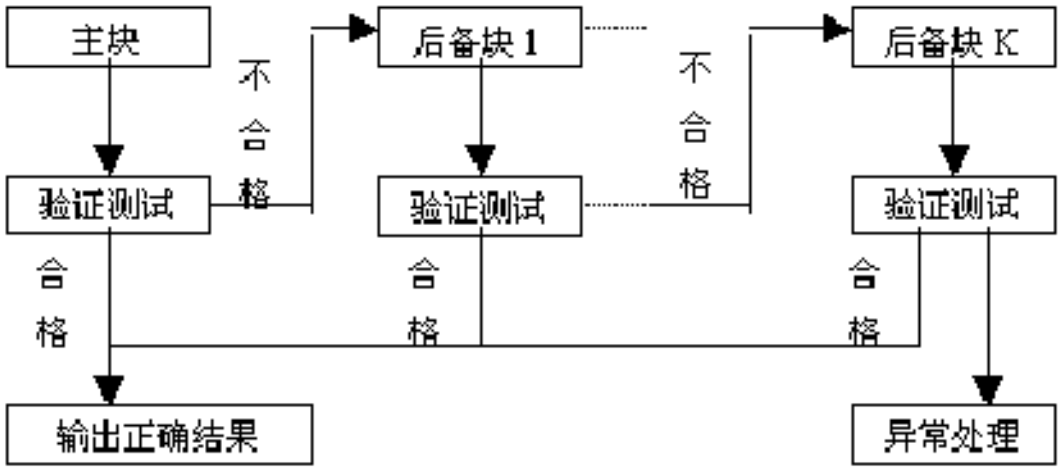


图　恢复块方法

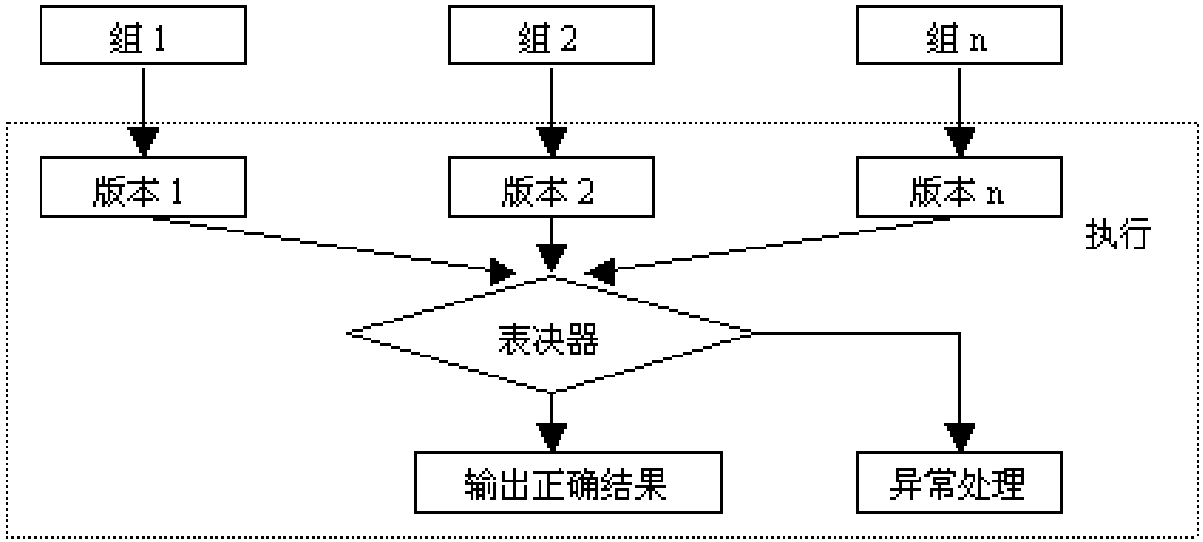


图　N版本程序设计

8、设计的评估和确认

- 数学确认
- 测量设计质量
- 比较设计

  Eg:一个规格说明，多个设计
- 设计评审

设计评审

- 初步设计评审
  - 同顾客与用户一起审查概念设计
- 关键设计审查
  - 向开发人员展示技术设计
- 程序设计审查
  - 程序员在实现之前获得对他们设计的反馈

关于设计评审的问题

1. 该设计是问题的解决方案吗？
2. 该设计是模块化的、结构良好的、易于理解的？
3. 是否能改进设计的结构和易懂性？
4. 该设计能够移植到其他平台？
5. 该设计可复用吗？
6. 该设计易于修改或扩展么？
7. 该设计支持测试么？
8. 适当的时候，该设计的性能是否最好？
9. 适当的时候，该设计是否能复用其他项目中的组件？
10. 算法是否合适，是否可以改进？
11. 如果这个系统是一个分阶段的开发过程，各阶段的连接是否足够充分，以保证阶从一个阶段到下一个段的变迁非常容易？
12. 设计文档是否齐全，是否包括设计选择和原理？
13. 设计是否交叉引用需求中的组件和数据？
14. 设计是否使用了适当的技术处理并防止发生故障？

概念+技术

9、设计归档

1. 设计原理
2. 菜单和其他显示屏格式
3. 用户界面：功能键，触摸屏描述，键盘布局，以及鼠标和游戏杆的使用
4. 报告格式
5. 输入：数据来自何方，如何将它们格式化，数据存储在什么介质中
6. 输出：数据送到何处，如何将它们格式化，数据存储在什么介质中
7. 一般功能特征
8. 性能约束
9. 存档过程
10. 错误处理方式

二、面向对象的设计的主要工作

主要工作：

- 用例实现精化
- 体系结构设计
- 构件设计
- 用户界面设计
- 数据持久设计
- 迭代精化

1、用例实现方案精化

可利用UML的交互图（顺序图、协作图）用于用例实现方案的表示。

用例实现方案的设计方法有三个步骤：

(1) 提取边界类、实体类和控制类；

(2) 构造交互图；

(3) 根据交互图精化类图。

将实体类精化是在实体类上加上设计风格和设计类，实体类和设计类交互完成需求功能。

1.1、 提取边界类、实体类和控制类

边界类

边界类描述系统与外部环境的交互，主要任务：

(1) 界面控制：包括输入数据的格式及内容转换，输出结果的呈现，软件运行过程中界面的变化与切换等。

(2) 外部接口：实现目标软件系统与外部系统或外部设备之间的信息交流和互操作。主要关注跨越目标软件系统边界的通信协议。

(3) 环境隔离：将目标软件系统与操作系统、数据库管理系统、应用服务器中间件等环境软件进行交互的功能与特性封装于边界类之中，使目标软件系统的其余部分尽可能地独立于环境软件。

控制类

控制类作为完成用例任务的责任承担者，协调、控制其他类共同完成用例规定的功能或行为。对于比较复杂的用例，控制类通常并不处理具体的任务细节，但是它应知道如何分解任务，如何将子任务分派给适当的辅助类，如何在辅助类之间进行消息传递和协调。

实体类

实体类表示目标软件系统中具有持久意义的信息项及其操作。

如何从分析模型中获取这些类？

- 执行者与用例之间的一种通信连接对应一个边界类。但是，如果两个以上的用例与同一执行者交互，并且这些交互具有共同的行为、完成相同或类似的任务，就可以考虑用同一边界类实现用例与执行者之间的交互。这就意味着边界类的作用范围可以超越单个用例。

- 实体类源于分析模型。有时也需要认真研读用例描述，从中发掘具有持久意义的信息项。
- 一个用例通常对应一个控制类。
- 如果不同用例的任务有较多类似之处，也可以考虑在多个用例的实现方案中共享同一控制类，此种情况应审慎对待，不同用例所需要的控制、协调行为往往会有差异。

1.2、 构造交互图

将分析模型中的用例描述转化成UML交互图，以交互图作为用例的精确实现方案。

- 用例描述中已包含事件流说明。
- 事件流中的事件应直接对应于交互图中的消息，而事件间的先后关系体现为交互图中的时序，对消息的响应则构成消息接收者的职责。

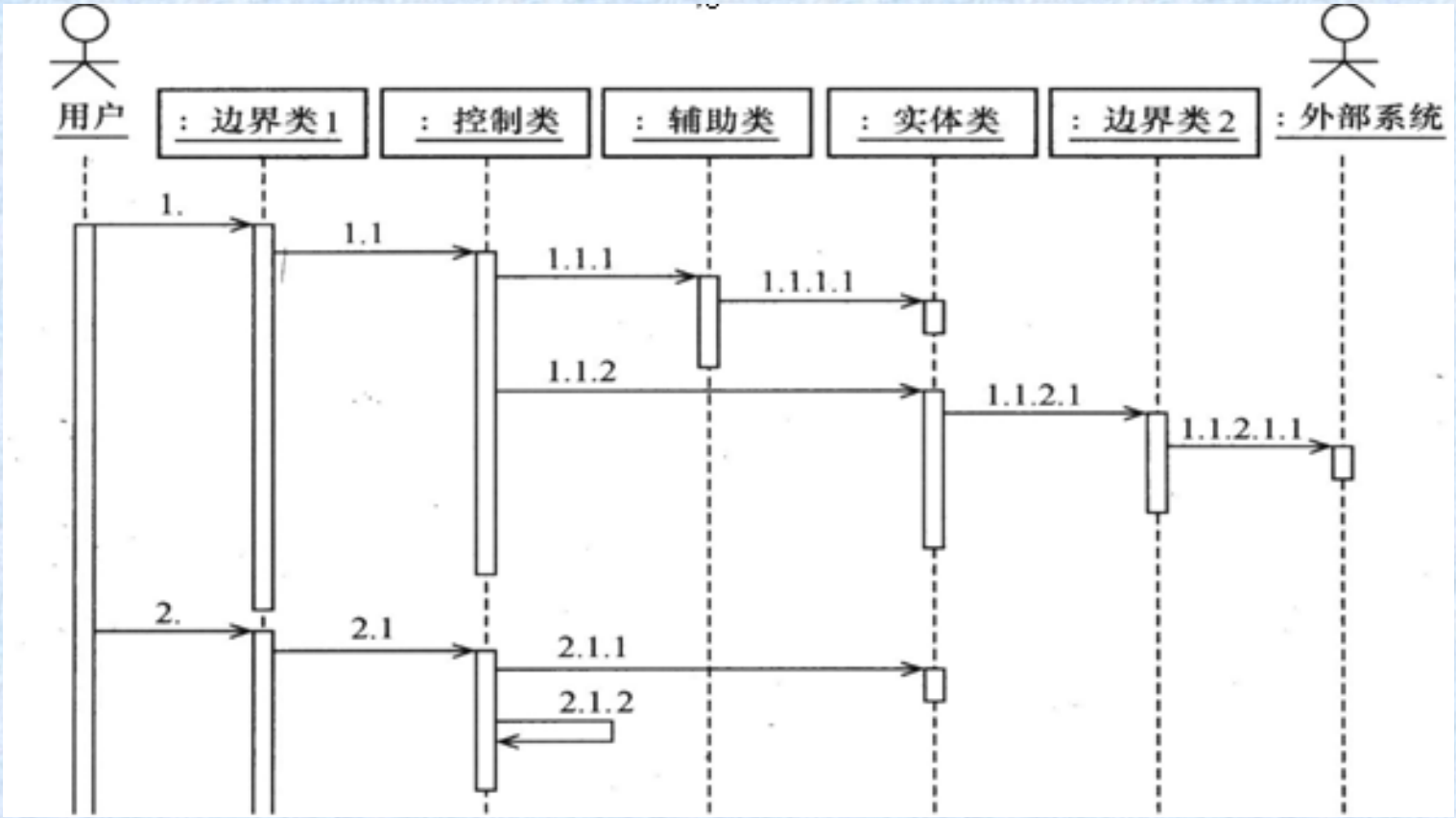
这种职责可用来帮助识别确立类的方法。

包含控制类和边界类的UML顺序图

- 用例的主动执行者
- 用户界面的边界类
- 控制类
- 实体类和辅助类
- 外部接口和环境隔离层的边界类
- 目标软件系统的边界之外的被动执行者

按此布局，在顺序图中不应该出现穿越控制类生命线的消息。

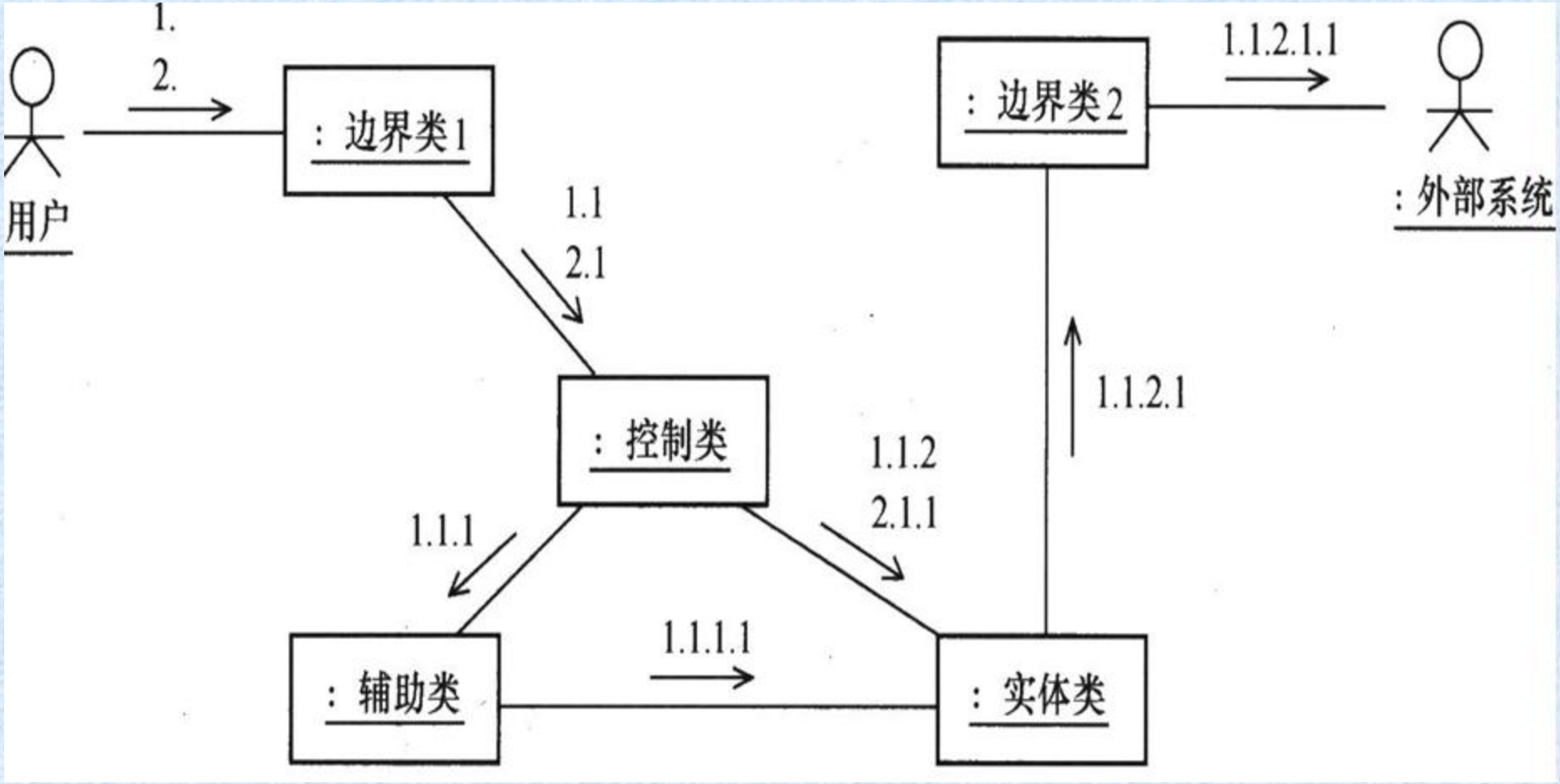
通过顺序图知道数据类型，消息交互


用户
：边界类1
：控制类
：辅助类
：实体类
：边界类2
：外部系统
1.
1.1
1.1.1
1.1.1.1
1.1.2
1.1.2.1
1.1.2.1.1
2.
2.1
2.1.1
2.1.2

当需要强调类之间的联系时可协作图。

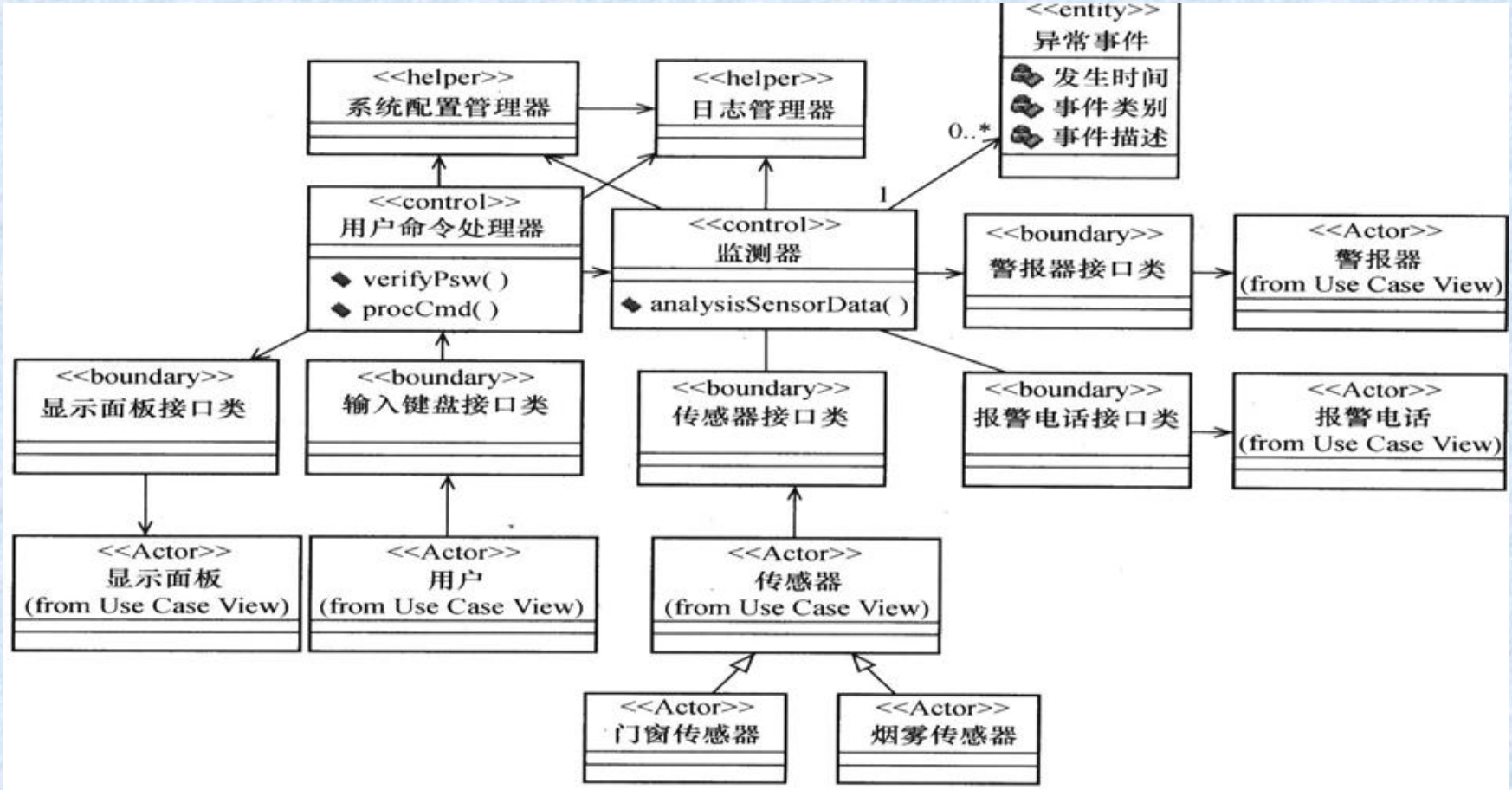
包含控制类和边界类的协作图的布局规则

- 控制类位于中心
- 主动执行者和作为用户界面的边界类位于左上方
- 作为外部接口和环境隔离层的边界类位于右上方
- 辅助类和实体类分别位于控制类的左下、右下方。


1.
2.
用户
：边界类1
1.1
2.1
：控制类
1.1.1
1.1.2
2.1.1
：辅助类
1.1.1.1
：实体类
1.1.2.1
：边界类2
1.1.2.1.1
：外部系统

1.3、 精化类图

可以利用交互图精化分析模型中的类图，将交互图中出现的新类添加到原有类图中，并且对相关的类进行精化，并最终确定类的方法和属性。注意，这是一个反复迭代的过程


<<entity>>
异常事件
发生时间
事件类别
事件描述
0..*
1
<<helper>>
系统配置管理器
<<helper>>
日志管理器
<<control>>
用户命令处理器
verifyPsw( )
procCmd( )
<<control>>
监测器
analysisSensorData( )
<<boundary>>
警报器接口类
<<Actor>>
警报器
(from Use Case View)
<<boundary>>
显示面板接口类
<<boundary>>
输入键盘接口类
<<boundary>>
传感器接口类
<<boundary>>
报警电话接口类
<<Actor>>
报警电话
(from Use Case View)
<<Actor>>
显示面板
(from Use Case View)
<<Actor>>
用户
(from Use Case View)
<<Actor>>
传感器
(from Use Case View)
<<Actor>>
门窗传感器
<<Actor>>
烟雾传感器

2、软件体系结构（软件架构）

定义（IEEE）：一个系统的基础组织，包含各个构件、构件互相之间与环境的关系，还有指导其设计和演化的原则。

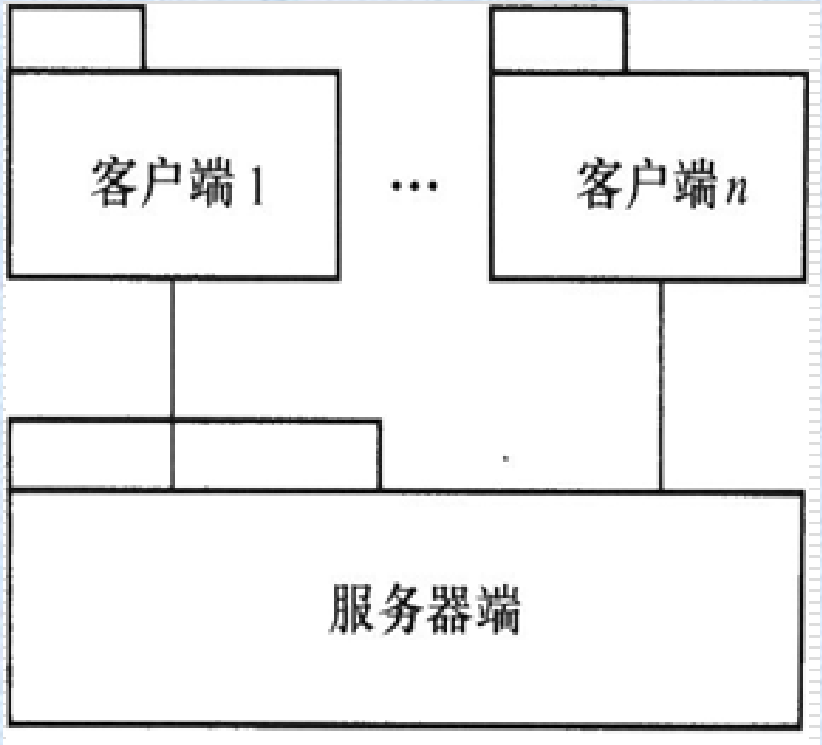
任何具有固定组成形式的数据、代码、数据集合、代码序列、数据和代码的结合体都可以称作结构。无论多么高层的结构，都是建立在基础结构之上的。软件结构的问题从最初的最基本、最底层的描述过渡到越来越高、越来越抽象的层次上。

2.1主流软件体系结构

- 客户/服务器模式、
- 模型—视图—控制器模式
- 分层模式

客户/服务器模式

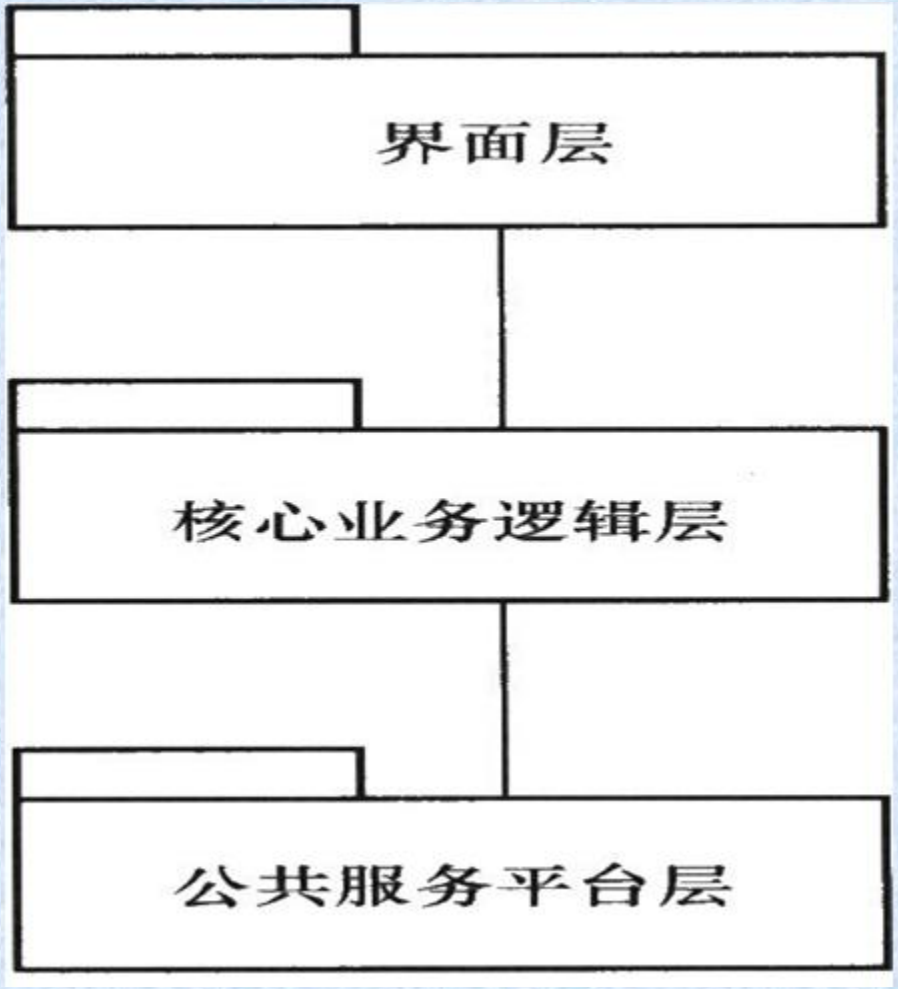
客户端负责用户输入和处理结果的呈现，服务端负责后台业务处理。



分层模式

将整个软件系统分为若干层次，最顶层直接面向用户提供软件系统的操作界面，其余各层为紧邻其上的层次提供服务。

分层模式可以有效降低软件系统的耦合度，应用普遍。


界面层
核心业务逻辑层
公共服务平台层

MVC模式

目的

- 将人机交互从核心功能中分离出来
- 模型对用户来说是不可见的，用户只需要观察视图
- 用户与模型的交互通过控制器提供的安全方法来实现

- MVC（Model-View-Controller）将一个交互式应用程序分成3个组件
  - 模型：包含核心功能和数据 （核心业务逻辑）
  - 视图：向用户显示信息
  - 控制器：处理用户输入
- 变更-传播机制保证了模型和用户界面之间的一致性

模型（Model）

- 封装了内核功能和数据
- 模型对于用户来说是不可见的(M与V独立)
- 模型独立于特定输出表示或者输入方式(M与C独立)
- 用户只能通过控制器操作模型(C是M与V之间的桥梁)

变更-传播机制

- 一个模型可对应多个视图
  - 如果用户通过一个视图的控制器改变了模型中的数据，那么依赖于该数据的其他视图也应该反映出这样的变化
  - 一旦模型的数据发生了变化，模型需要通知所有相关的视图做出相应的变化
- 维护数据的一致性

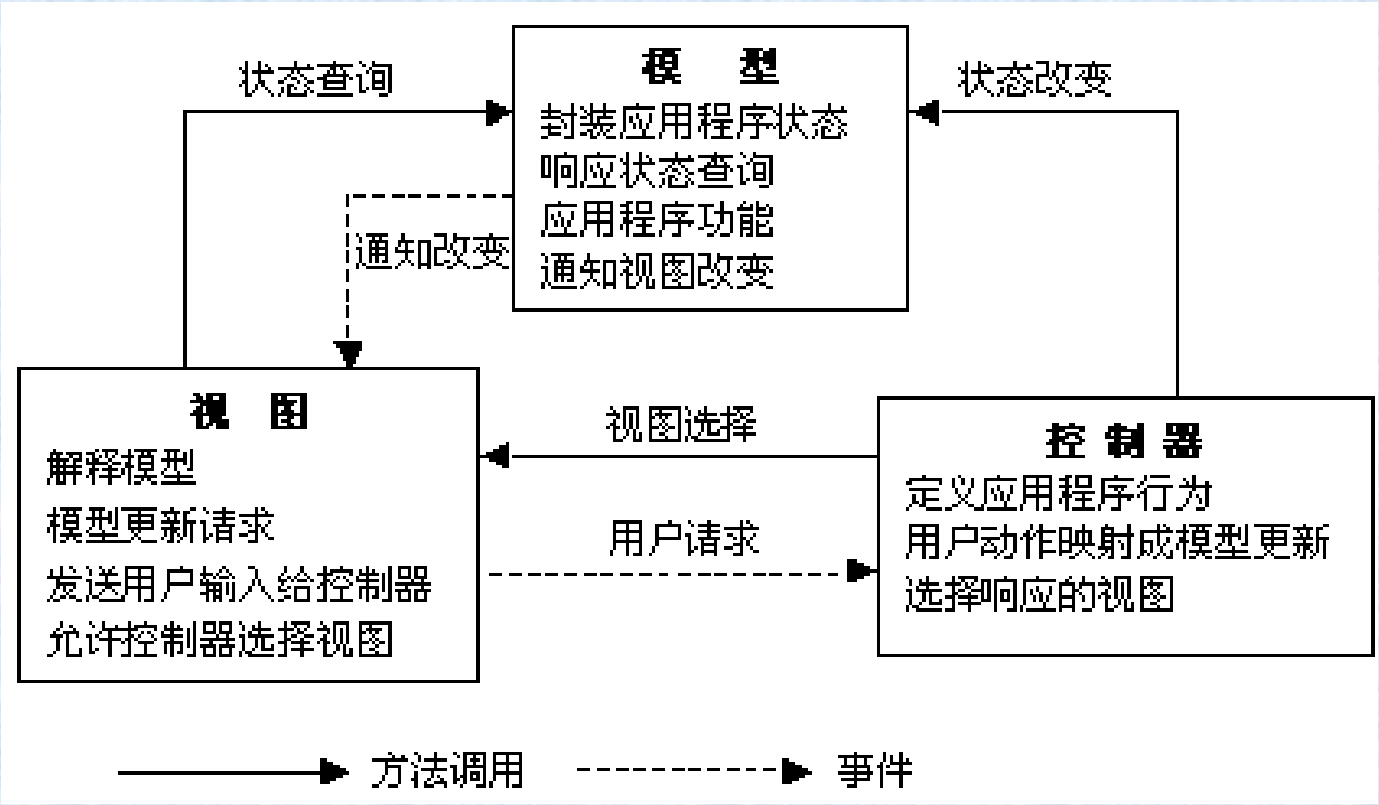
- 工作原理：
  - 模型维护了一个表
  - 所有视图还有一些控制器在这个表中登记了对变更通知的需求
  - 模型状态的改变将触发变更-传播机制，每个在表中登记的视图和控制器都会收到变更通知

视图（View）

- 向用户显示信息
  - 不同的视图使用不同的方法呈现信息
  - 每个视图组件都有一个更新函数，这个函数被模型变更通知激活
  - 这个函数被激活（此时模型已经改变）后，将使得视图重新和模型一致
  - 在初始化阶段，视图向模型登记请求变更通知（表）
- 从模型获得数据
  - 通过状态查询函数实现
  - 例如：定时刷新

控制器（Controller）

- 每个视图有一个相关的控制器组件(一一对应)
- 控制器组件接受事件，并翻译成输入
  - 事件如何发送到控制器由用户界面平台决定
  - 事件被翻译成为对模型或者视图的请求
  - 如果控制器的行为依赖于模型的状态，那么控制器也需要向模型登记请求变更通知
  - 例如：用户点击按钮，按钮的事件响应函数将采取相应的措施处理用户要求
- 用户仅仅通过控制器与系统交互



MVC模式特别适合于Web应用系统

注意：

- 视图与控制器是一一对应的关系
- 模型与视图是一对多的关系
- 变更-传播机制保持模型与视图、控制器之间状态的一致性

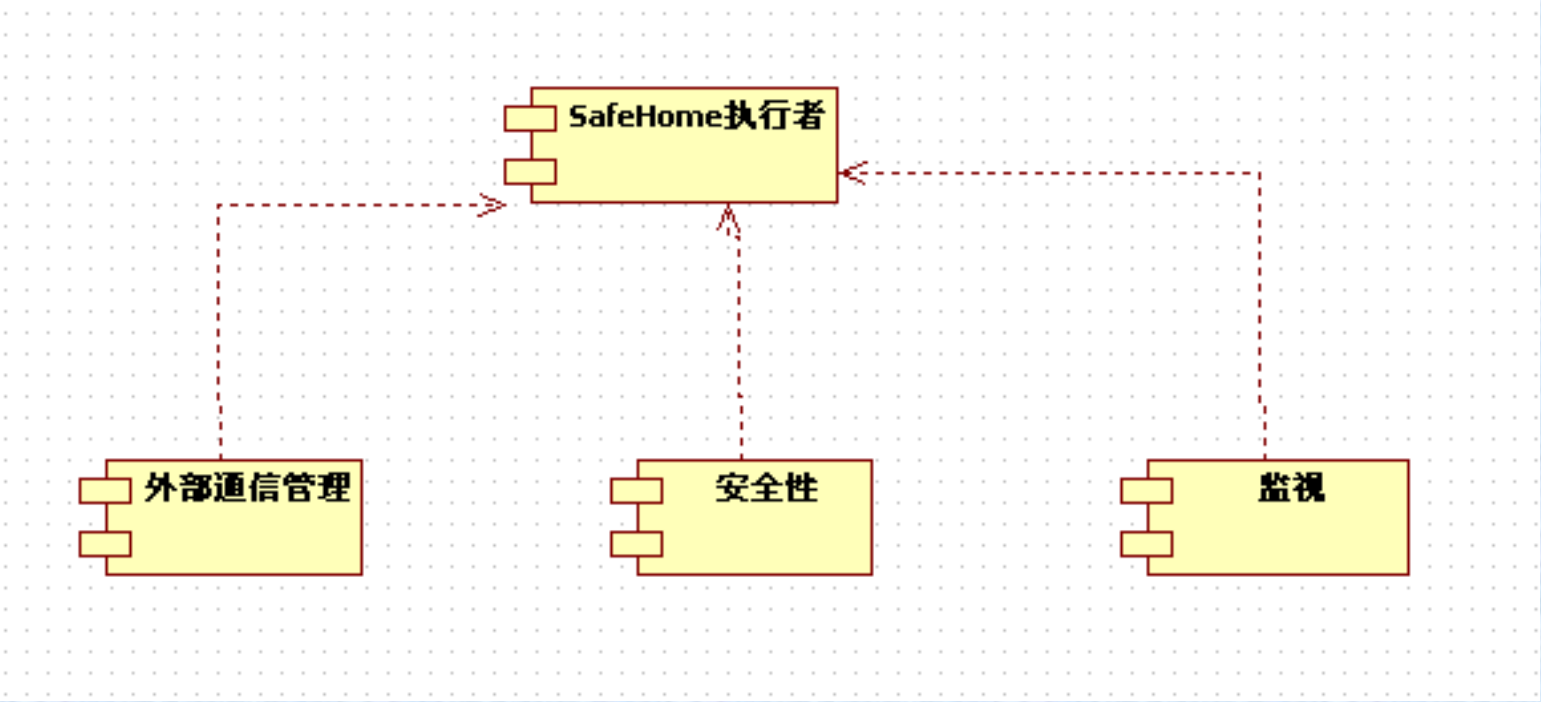
2.2 体系结构图

● 定义原始模型，系统的体系结构可有这些原始模型组成。原始模型可从前面的分析设计中获得

●对体系结构设计是一个逐步精化得过程。

●将体系结构精化为构件，当体系结构精华为构件时，系统的结构开始显现。

可使用包图（子系统）或组件图来表示体系结构图


SafeHome执行者
外部通信管理
安全性
监视

3、构件设计

构件设计将软件体系结构的结构元素变换为对软件构件的过程性描述。从基于类的模型、动态模型和静态模型等获得的信息作为构件设计的基础

构件设计：

- 在相对较低的抽象层次上详细地说明所有算法。
- 精华每个构件的接口
- 定义构件级数据结构
- 评审每个构件并修正所有已发现的错误

设计原则可参见前面优秀设计特征章节和后面OO设计原则

4、数据持久存储服务考点

设计数据持久存储服务的目的是，将软件系统中依赖于系统运行环境的数据存取部分与其他部分分离。数据存取通过一般的数据管理系统(如文件系统、关系数据库或面向对象数据库)实现。

数据持久存储服务的设计包括，定义数据格式和定义数据存取操作两部分。

(1) 定义数据格式。

根据目标软件系统对数据存储的需求，考虑持久存储介质的特性，设计数据的存储格式。

△针对文件系统，需要定义用于保存数据的文件类别、每类文件的记录格式；

△针对关系数据库，需要定义表格的字段名称、类型、关键字、约束条件等；

△针对面向对象数据库(OODB)，数据格式的定义相对简单，因为OODB直接支持对象的存储和读取。

(2) 定义数据存取操作。

数据持久存储服务至少包含数据的存储和读取两种操作。

△存储操作负责将实体对象中的属性数据写至持久存储介质

△读取操作则负责从持久存储介质中的序列化数据恢复对象的属性值。

△数据存取操作一般以服务类的形式为系统的其他部分提供数据持久存储服务。

△ 在引入持久存储服务之后，目标软件系统的其他部分对持久存储介质的访问必须通过该服务进行，不能直接访问。

- 数据库设计与持久存储服务设计相结合考虑
- 可使用成熟的数据持久化工具

Hibernate　iBATIS

关于持久化工具性能问题的考虑！

5、 设计用户界面

设计阶段要给出有关人机交互的所有系统成份，包括：用户如何操作 系统、系统如何响应命令、系统显示信息的报表格式等。

设计用户界面的策略与步骤：

- 熟悉用户并对用户分类。设计人员应深入用户环境，考虑用户需要完成的任务、完成这些任务需要什么工具支持以及这些工具对用户是否适用。

不同类型的用户要求不同，一般可按技术熟练程度、 工作性质和访问权限对用户分类，以便尽量照顾到所有用户的合理要求，并优先满足某些特权用户。

设计用户界面的策略与步骤：

- 用户类别分析用户工作流程与习惯。在用户分类的基础上，从每类中选取一个用户代表，建立包括下列内容的调查表，并通过对调查按结果的分析判断用户对操作界面的需求和爱好：

姓名

期望软件用途

特征(年龄、文化程度、限制等)

主要要求与爱好

技术熟练程度

任务客观场景描述

- 设计并优化命令系统。
  - 在设计一个新命令系统时，应尽量遵循用户界面的一般原则和规范，必要时参考一些优秀的商品软件。
  - 根据用户分析结果确定初步的命令系统，然后再优化。
  - 命令系统既可为若干菜单、菜单栏，亦可为一组按钮。

优化命令系统

- 应考虑命令的顺序，一般常用命令居先，命令的顺序与用户工作习惯保持一致；（工具栏）
- 根据外部服务之间的聚合关系组织相应的命令，总体功能对应父命令，部分功能对应子命令；
- 充分考虑人类记忆的局限性(即所谓“7±2 ”原则或“3×3”原则)，命令系统最好组织为一棵两层的多叉树；
- 应尽可能减少用户完成一个操作所需的动作(如点按、拖曳和击键等)，并为熟练用户提供操作捷径

- 设计用户界面的各种细节。此步骤包括：设计一致的用户界面风格；耗时操作的状态反馈；“undo”机制；帮助用户记忆操作序列等。

- 增加用户界面专用的类与对象。用户界面专用类的设计与所选用的图形用户界面(GUI)工具或者支持环境有关。一般而言，需要为窗口、菜单、对话框等界面元素定义相应的类，这些类往往继承自GUI工具或者支持环境提供的类库中的父类。（很多开发工具可自动实现）
- 利用快速原型演示，改进界面设计。为人机交互部分构造原型，是界面设计的基本技术之一。

6、精化设计模型

经过前面的分析和设计步骤，整个软件系统的设计模型基本实现了。必须对这些模型再进行分析、优化，为后续的实现阶段奠定坚实基础。

设计模型精化的任务

(1) 以顶层架构图为基础，精化目标软件系统的体系结构。

(2) 精化类之间的关系。

(3) 精化类的属性和操作。

(4) 针对具有明显状态转换特征的类，设计状态图。

(5) 针对比较复杂的类方法，设计活动图。

三、面向对象设计的原则工作

类的设计原则：

SRP　单一职责原则

OCP　开放封闭原则

LSP　里氏替换原则

DIP　依赖倒置原则

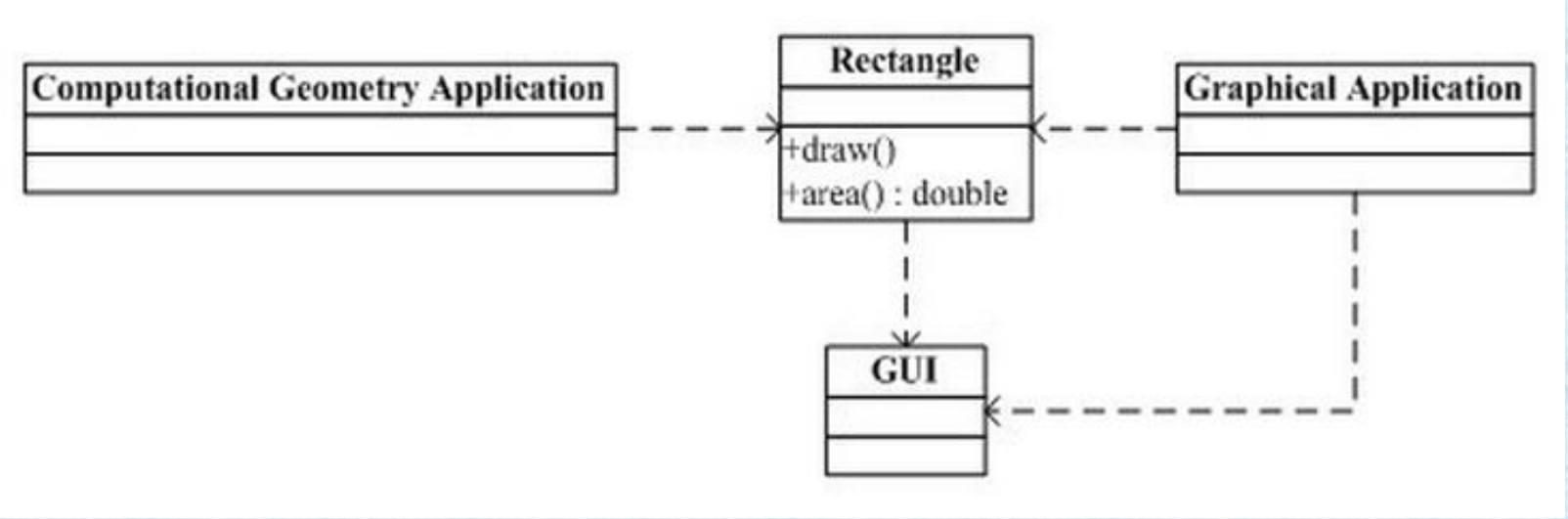
ISP　接口隔离原则

考给一个类图，判断违背了什么原则

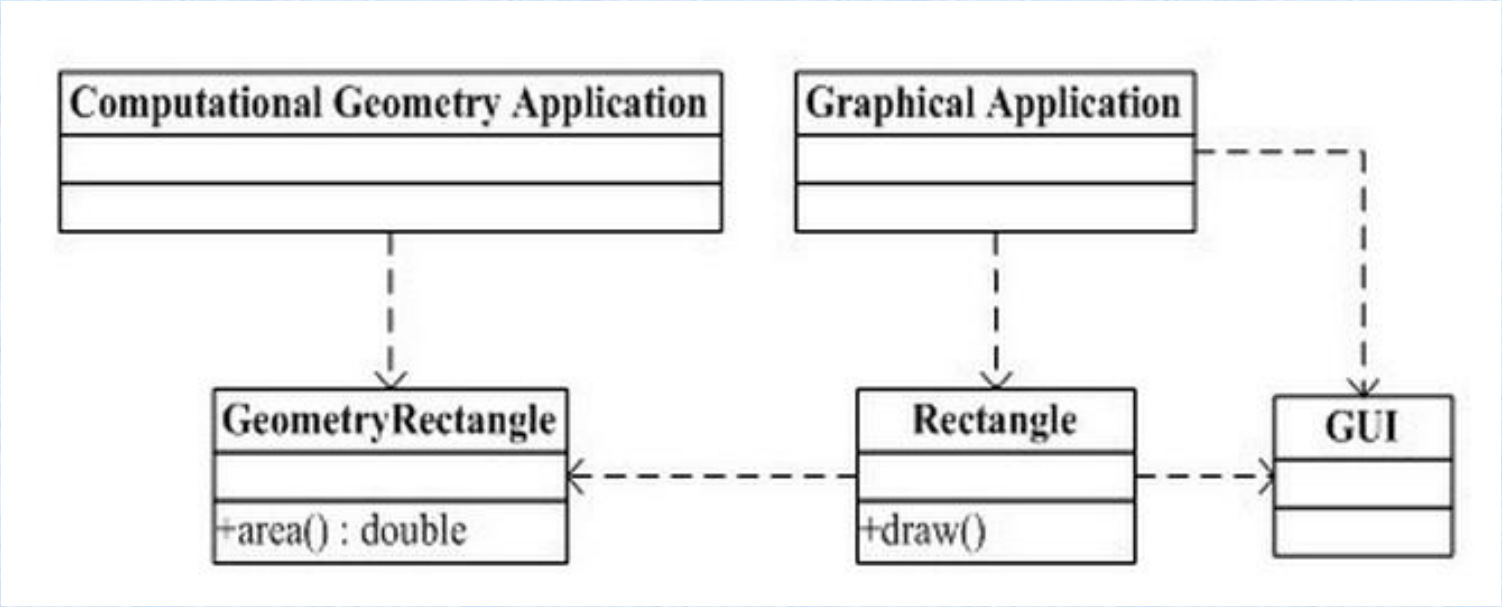
单一职责原则（SRP）

一个类应该有且只有一个改变的理由。即每一个类应该只专注于做一件事

- SRP原则衡量的标准—内聚.你写了高内聚的软件其实就是说你很好的应用了SRP原则。
- 怎么判断一个职责是不是一个对象的呢？你试着让这个对象自己来完成这个职责，比如：“书自己阅读内容”，阅读的职责显然不是书自己的。



Rectangle类具有两个方法，一个方法把矩形绘制在屏幕上，另一个方法计算矩形面积。该设计违反了SRP原则



较好的设计是把这两个职责分离到两个完全不同的类中。该设计满足SRP原则

开放-关闭原则（OCP）

软件实体应该可以扩展，但不可以修改。

开放：从哲学上其实是要我们在设计和开发软件时提高抽象层次，不要总在具体对象层面上进行处理。

关闭：设计好的类一定要对修改关闭

eg:我要去医院找医生看病

里氏代换原则（LSP）

子类型应该能代替掉其父类型，且代替后程序运行情况不会错乱。

eg:每个办公室需要一台PC.

可否用笔记本代替PC?

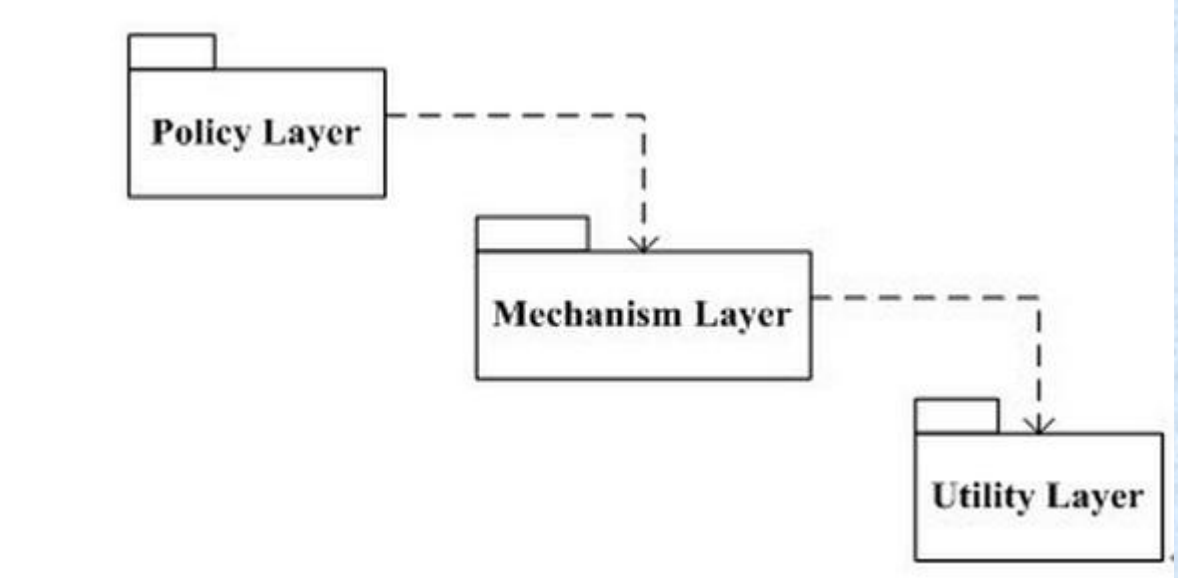
注意：LSP应用于程序世界和现实世界时有很大差别的，现实世界繁杂、不确定性因素多，而程序世界简单、确定。总之，LSP就是让你记住一 条，凡是系统中有继承关系的地方，子类型一定能代替父类型，而且替换后程序运行要正常。继承也是“一般和特殊”的哲学原理在程序世界中的体现。

依赖倒置原则（DIP）

- 高层模块不应该依赖于低层模块，两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。

为什么要倒置？如何实现倒置？

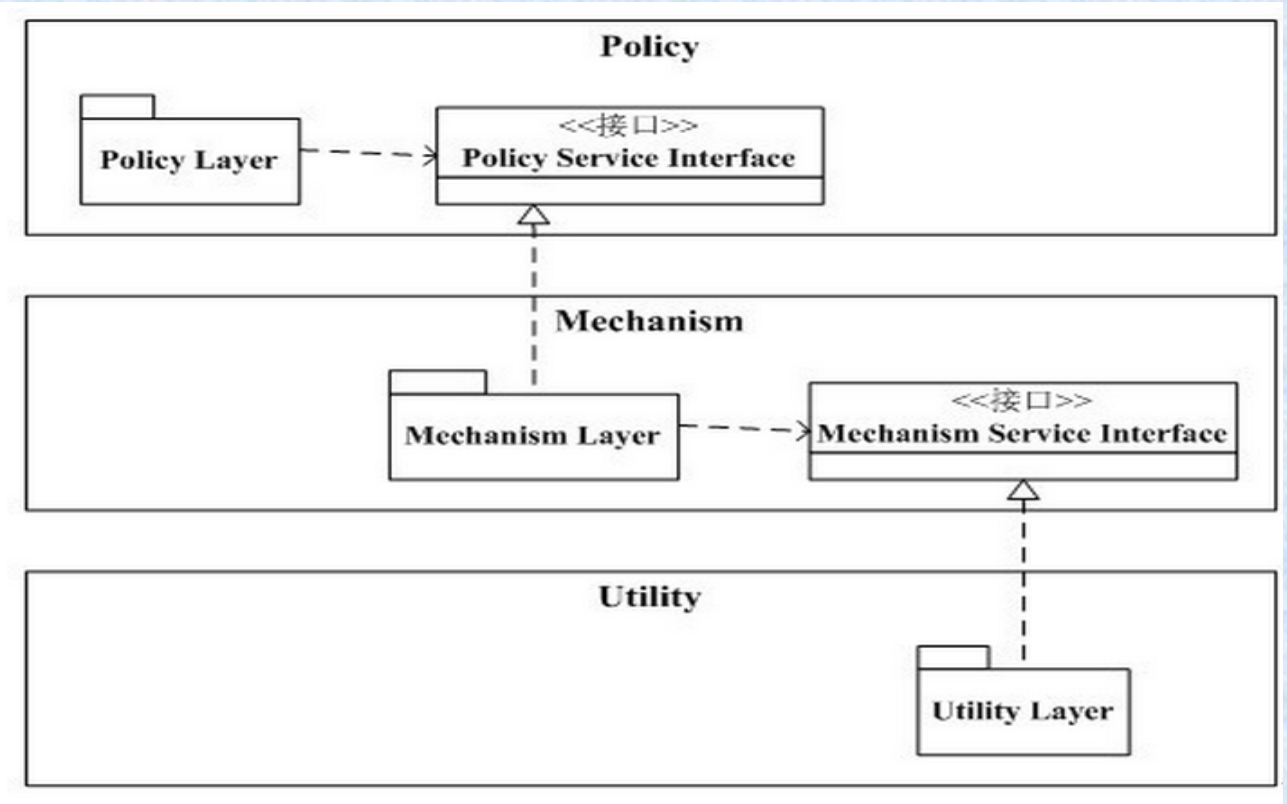
高层模块不应该依赖于低层模块，两者都应该依赖于抽象。



该设计有什么问题？
违反DIP原则。

从耦合的角度考虑，
耦合式高是低？

许多传统的软件开发方法，例如结构化分析和设计，总是倾向于创建一些高层模块依赖于低层模块，策略（policy）依赖于细节的软件结构



该设计 符合DIP原则。

从耦合的角度考虑，耦合式高是低？

低层模块实现了在高层模块中声明并被高层模块调用的接口

（Hollywood原则：“Don’t call us, we’ll call you.”）

抽象不应该依赖于细节，细节应该依赖于抽象

该启发式规则建议不应该依赖于具体类。也就是说，程序中所有的依赖关系都应该终止于抽象类或者接口

- 任何变量都不应该持有一个指向具体类的指针或者引用
- 任何类都不应该从具体类派生
- 任何方法都不应该覆写它的任何基类中已经实现了的方法

如何判断你的系统采用的是OO?

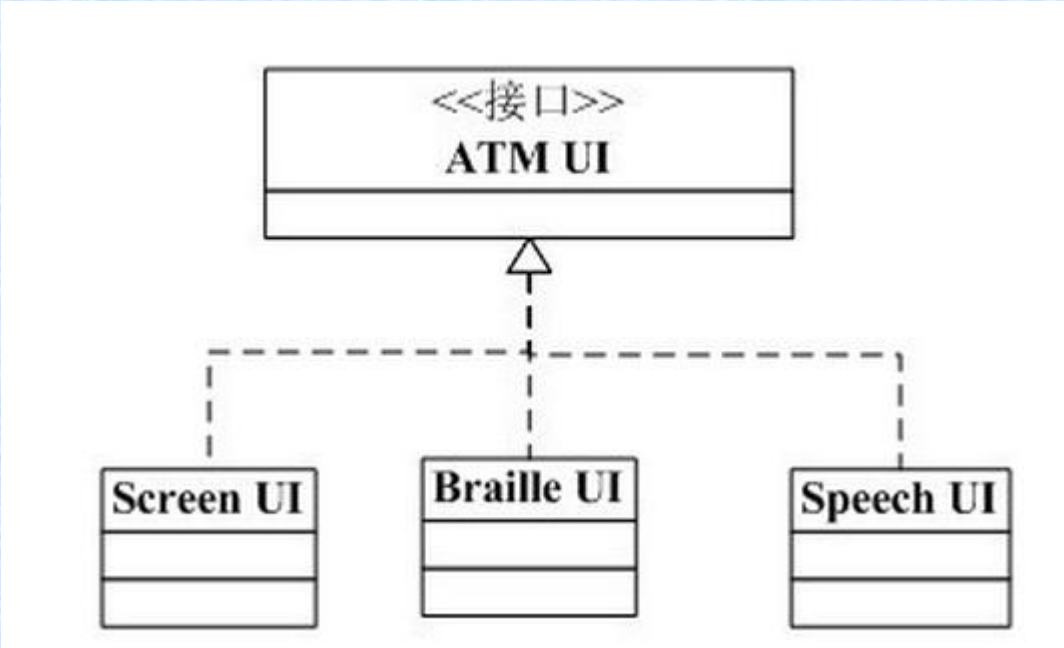
依赖关系的倒置正好是面向对象设计的标志所在。使用何种语言来编写程序是无关紧要的。如果程序的依赖关系是倒置的，它就是面向对象的设计。否则，它就是过程化的设计

接口隔离原则（ISP）

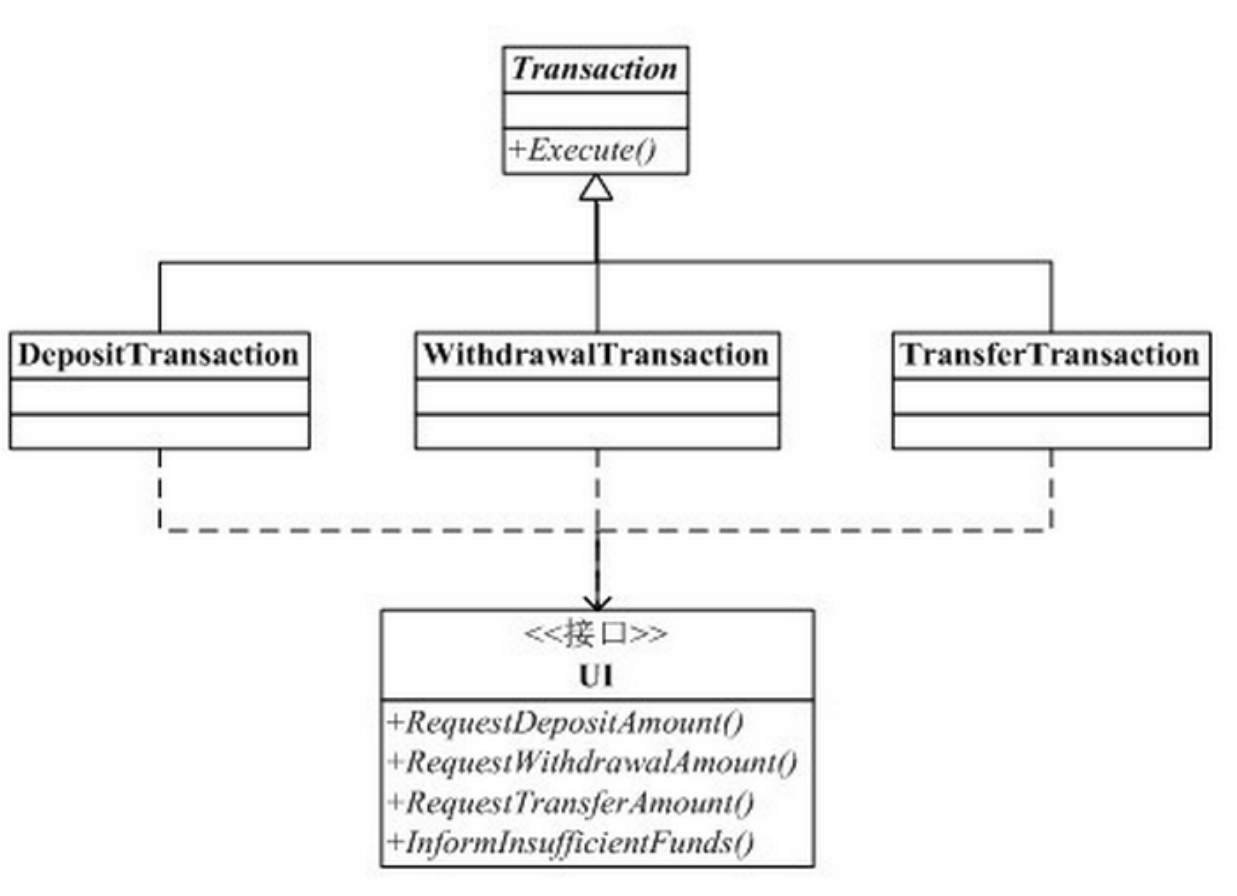
不应该强迫客户依赖于它们不要的方法。接口属于客户，不属于它所在的类层次结构。

如果一个客户程序依赖于一个含有它不使用的方法的类，但是其他客户程序却要使用该方法，那么当其他客户要求这个类改变时，就会影响到这个客 户程序。我们希望尽可能地避免这种耦合，因此我们希望分离接口。

假如ATM需要一个非常灵活的用户界面。它的输出信息需要被转换成许多不同的语言。输出信 息可能被显示在屏幕上，或者布莱叶盲文书写板上，或者通过语音合成器说出来。显然，通过创建一个抽象基类，其中具有用来处理所有不同的、需要被该界面呈现 的消息的抽象方法，就可以实现这种需求



可以把每个ATM可以执行的不同操作封装为类Transaction的派生类。然后通过其派生类调用UI方法来实现不同的操作

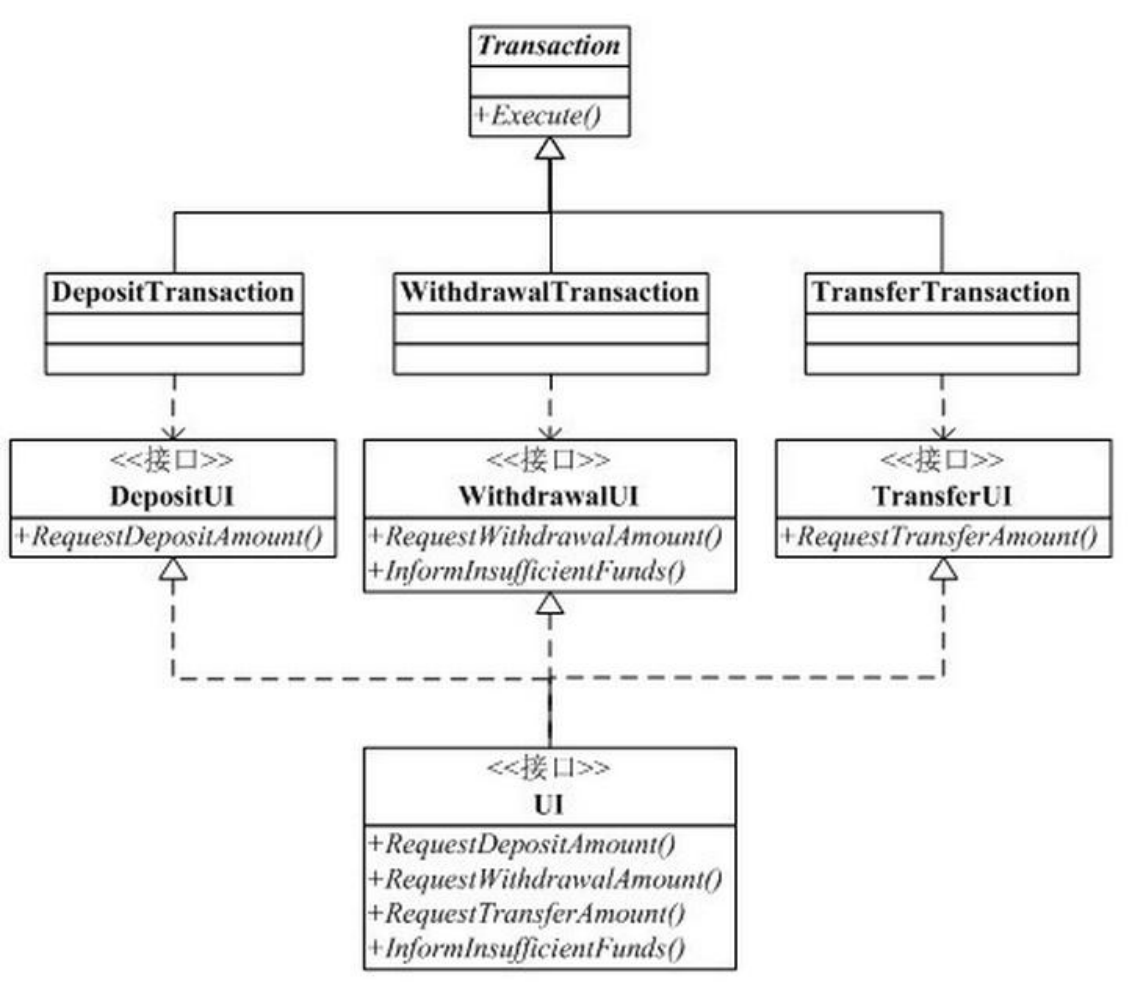


该设计有什么问题？

违反ISP原则。

从耦合的角度考虑，耦合式高是低？

改进后的设计，满足ISP原则



包的设计原则：

包的内聚性

REP　重用发布等价原则
CCP　共同封闭原则
CRP　共同重用原则

包的耦合性

ADP　无环依赖原则
SDP　稳定依赖原则
SAP　稳定抽象原则

随着应用程序的规模和复杂度的增加，需要在更高的层次对它们进行组织。类对于小型应用程序来说事非常方便的组织单元，但是对于大型应用程序来 说，如果仅仅使用类作为唯一的组织单元，就会显得粒度过细。因此，就需要比类“大”的“东西”来辅助大型应用程序的组织。这个“东西”就是包

重用发布等价原则（REP）

重用的粒度就是发布的粒度：一个包中的软件要么都是可重用的，要么都是不可重用的。

从发布的角度，发布时即要求所有类重用或不重用。不允许，一个包中9个类重用，一个类不重用。

REP指出，一个包的重用粒度可以和发布粒度一样大。我们说重用的任何东西都必须同时被发布和跟踪。简单的编写一个类，然后声称它是可重用的做法是不现实的。只有在建立一个跟踪系 统，为潜在的使用者提供所需的变更通知、安全性以及支持后，重用才有可能。

REP带给了我们关于如何把设计划分到包中的第一个提示。由于重用性必须是基于包的，所以可重用的包必须包含可重用的类。因此，至少，某些包应该有一组可重用的类组成。

共同重用原则（CRP）

从应用的角度来看，重用所有类，不重用的可以不放在一起。

一个包中所有类应该是共同重用的。如果重用了包中的一个类，那么就重用包中的所有类

CRP告诉我们更多的是，什么类不应该放在一起。CRP规定相互之间没有紧密联系的类不应该放在同一个包中

共同封闭原则（CCP）

把经常变的类放在一个包里面，以后要改变就改变这一个包。考虑后续的应用，安全性和调整能力都可以提升。

包中的所有类对于同一类性质的变化应该是共同封闭的。一个变化若对一个包产生影响，则将对包中的所有类产生影响，而对于其他的包不造成任何影响。把不稳定的类放到同一个包中，不会变化的类放到一个包中

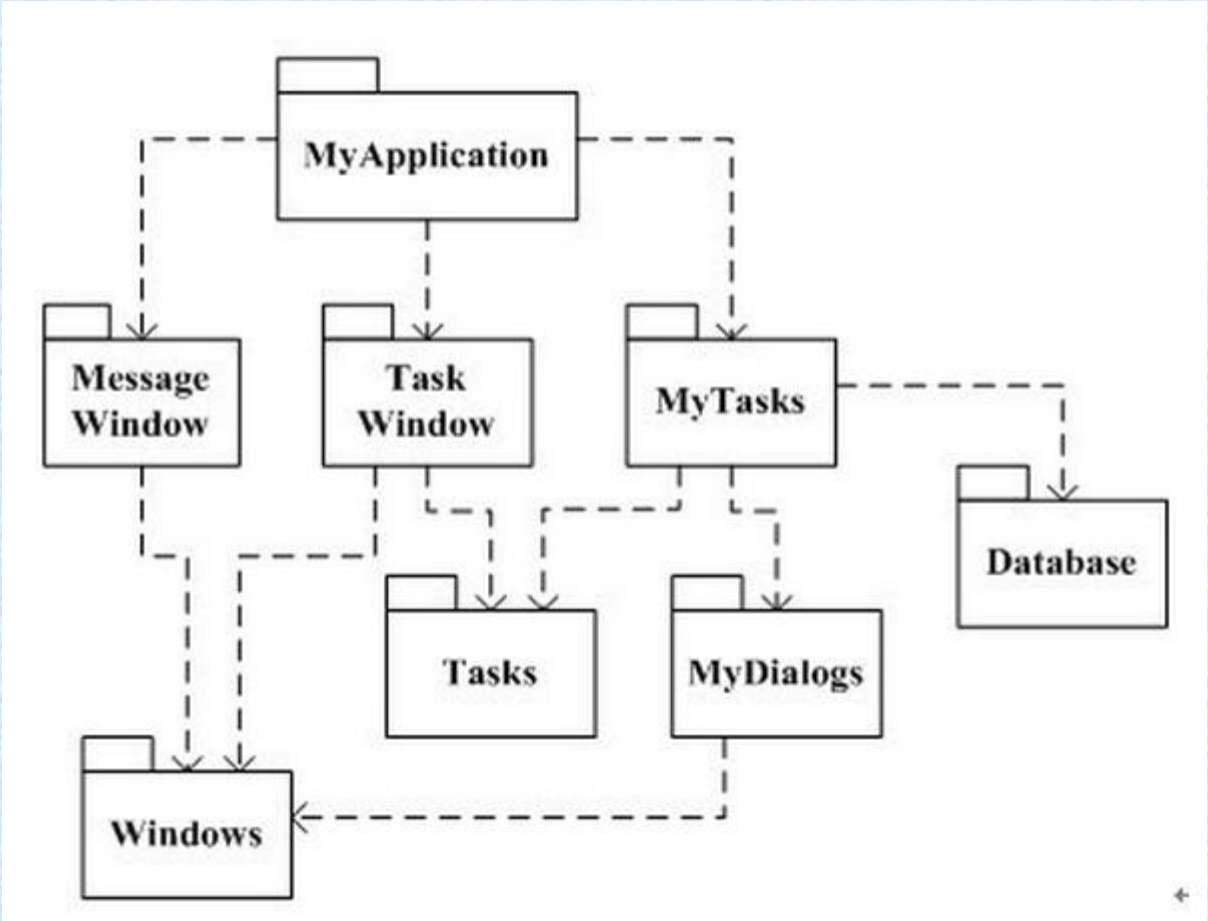
●这个原则规定了一个包不应该包含多个引起变化的原因。

●该原则也指出我们所设计的系统应该对于我们经历过的最常见的变化做到封闭。

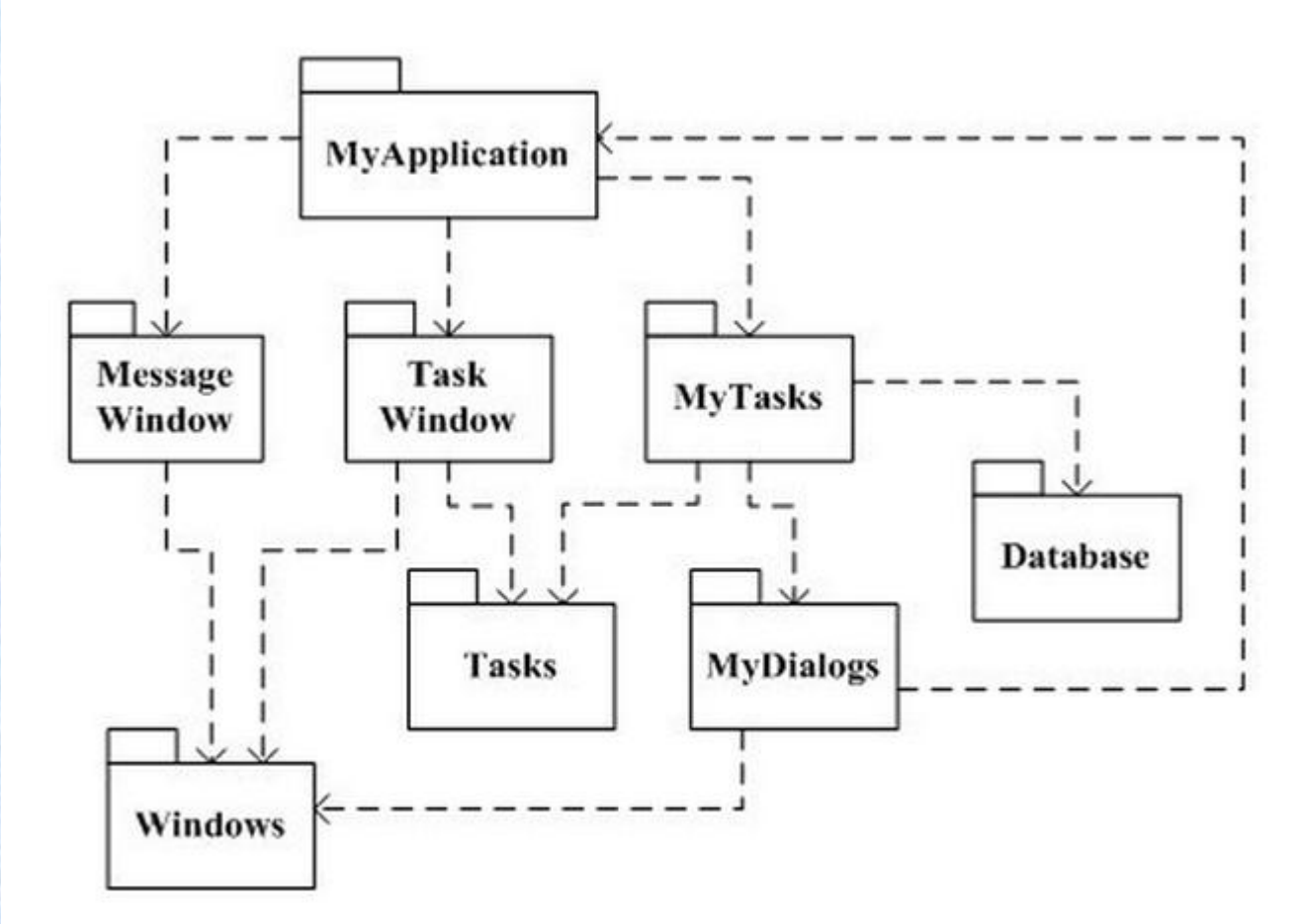
无环依赖原则（ADP）

耦合：运用DIP原则调整，减少依赖改变方向。

在包的依赖图中，不允许存在环。

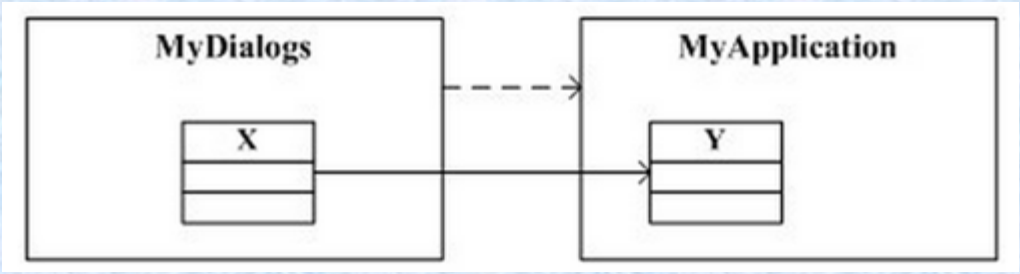


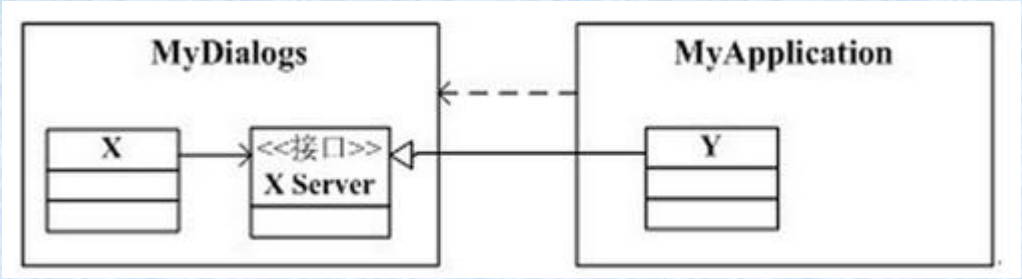
该原则指出包结构应该是有向无环图


MyApplication
Message Window
Task Window
MyTasks
Database
Tasks
MyDialogs
Windows

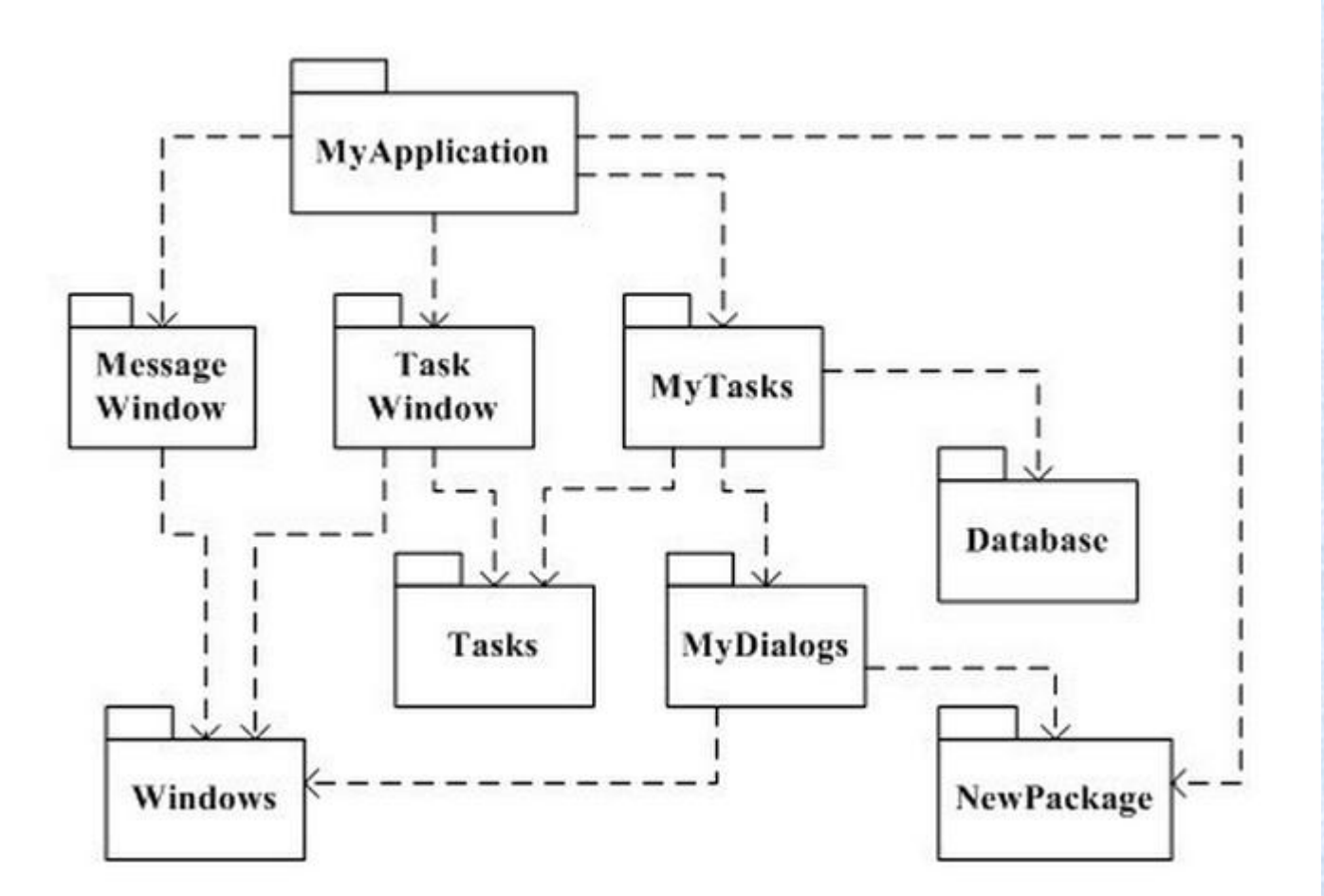
违反该原则的设计

如何修改上面的设计？





利用DIP原则


MyApplication
Message Window
Task Window
MyTasks
Database
Tasks
MyDialogs
Windows
NewPackage

满足ADP原则

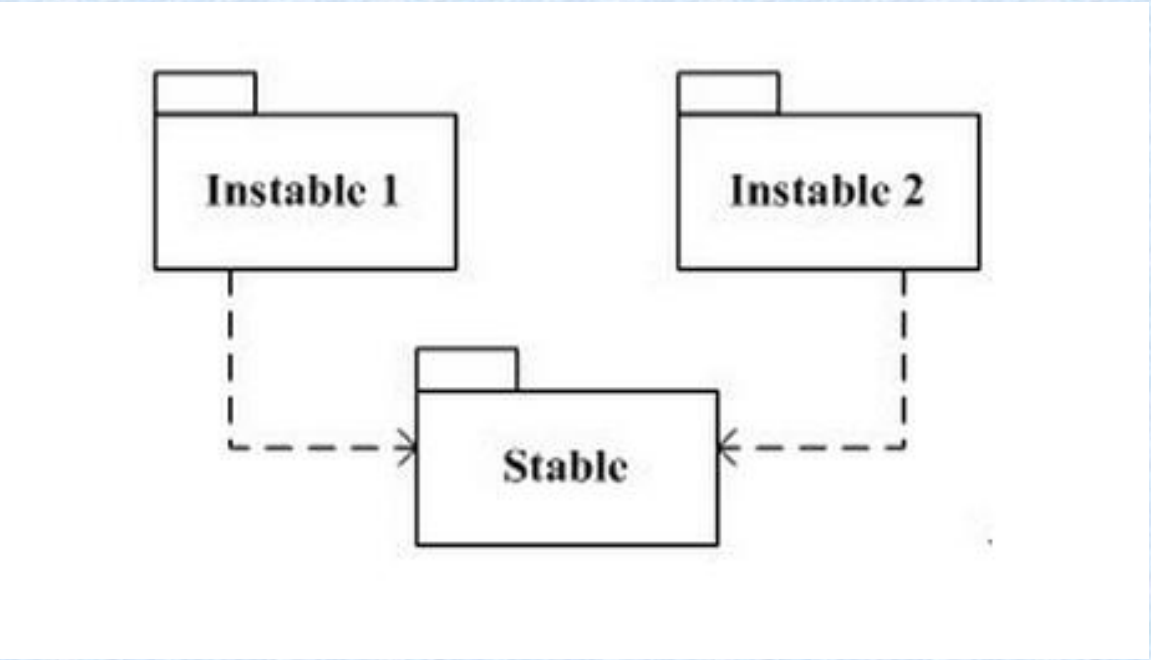
稳定依赖原则（SDP）

朝着稳定的方向进行依赖。

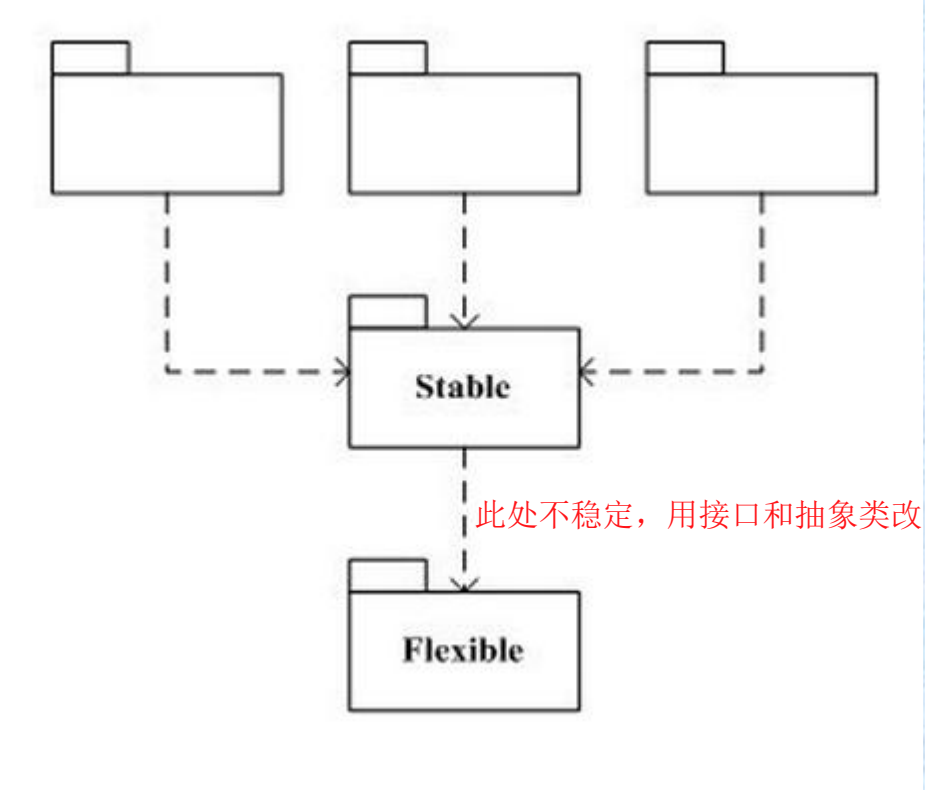
设计不能是完全固定的。要使设计可维护，某种程度的易变性是必要的。我们通过遵循共同封闭原则（CCP）来达到这个目标。使用这个原则，可以创建对某些变化类型敏感的包。这些包被设计成可变的。我们期望它们的变化。

对于任何包而言，如果期望它是可变的，就不应该让一个难以改变的包依赖于它！否则，可变的包同样也会难以更改。

理想的包配置

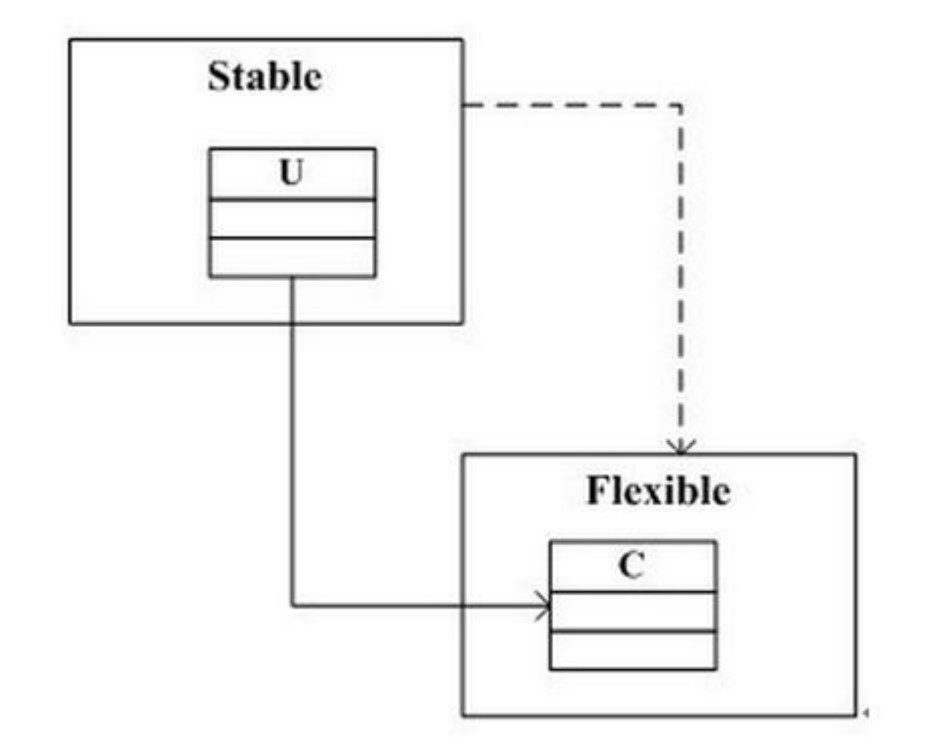


可改变的包位于顶部并依赖于底部稳定的包。把不稳定的包放在图的顶部是一个有用的约定，因为任何一个向上的箭头都意味着违反了SDP

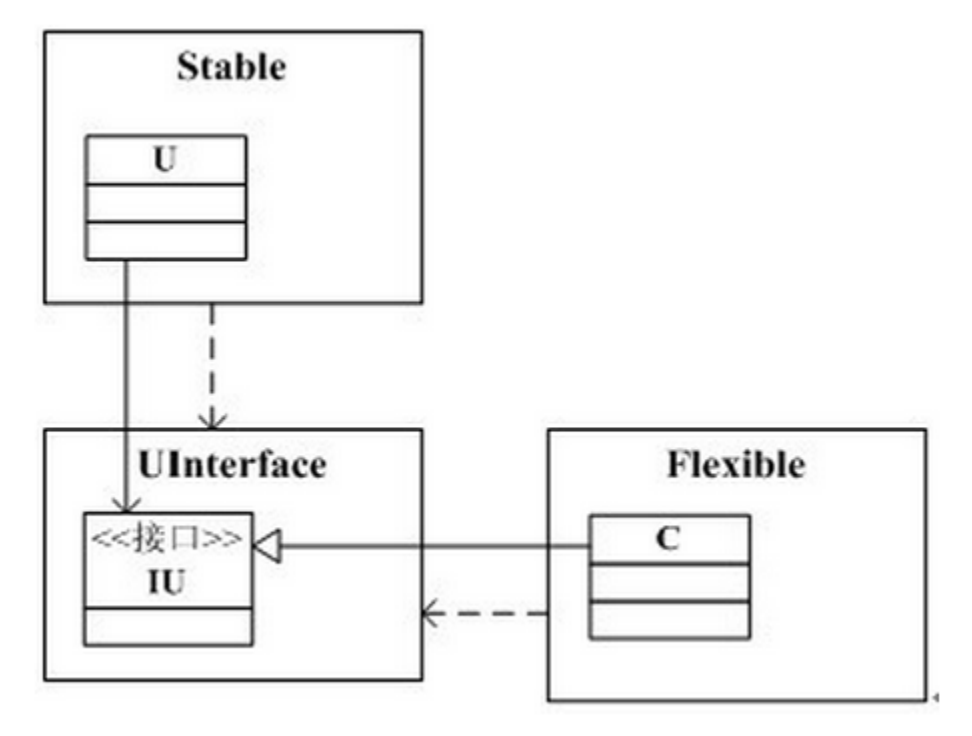


违反了SDP

造成违反SDP原则的原因如下



如何修改？



使用DIP修正稳定性违规

稳定抽象原则（SAP）

一个包越稳定，则越抽象。要基于大量的抽象类和接口。

包的抽象程度应该和其稳定程度一致。

该原则把包的稳定性和抽象性联系起来。它规定，一个稳定的包应该也是抽象的，这样它的稳定性就不会使其无法扩展。另一方面，它规定，一个不稳定的包应该是具体的，因为它的不稳定性使得其内部具体代码易于更改。

因此，如果一个包是稳定的，那么它应该也要包含一些抽象类，这样就可以对它进行扩展。可扩展的稳定包是灵活的，并且不会过分限制设计。