第5章 软件体系结构设计与评估

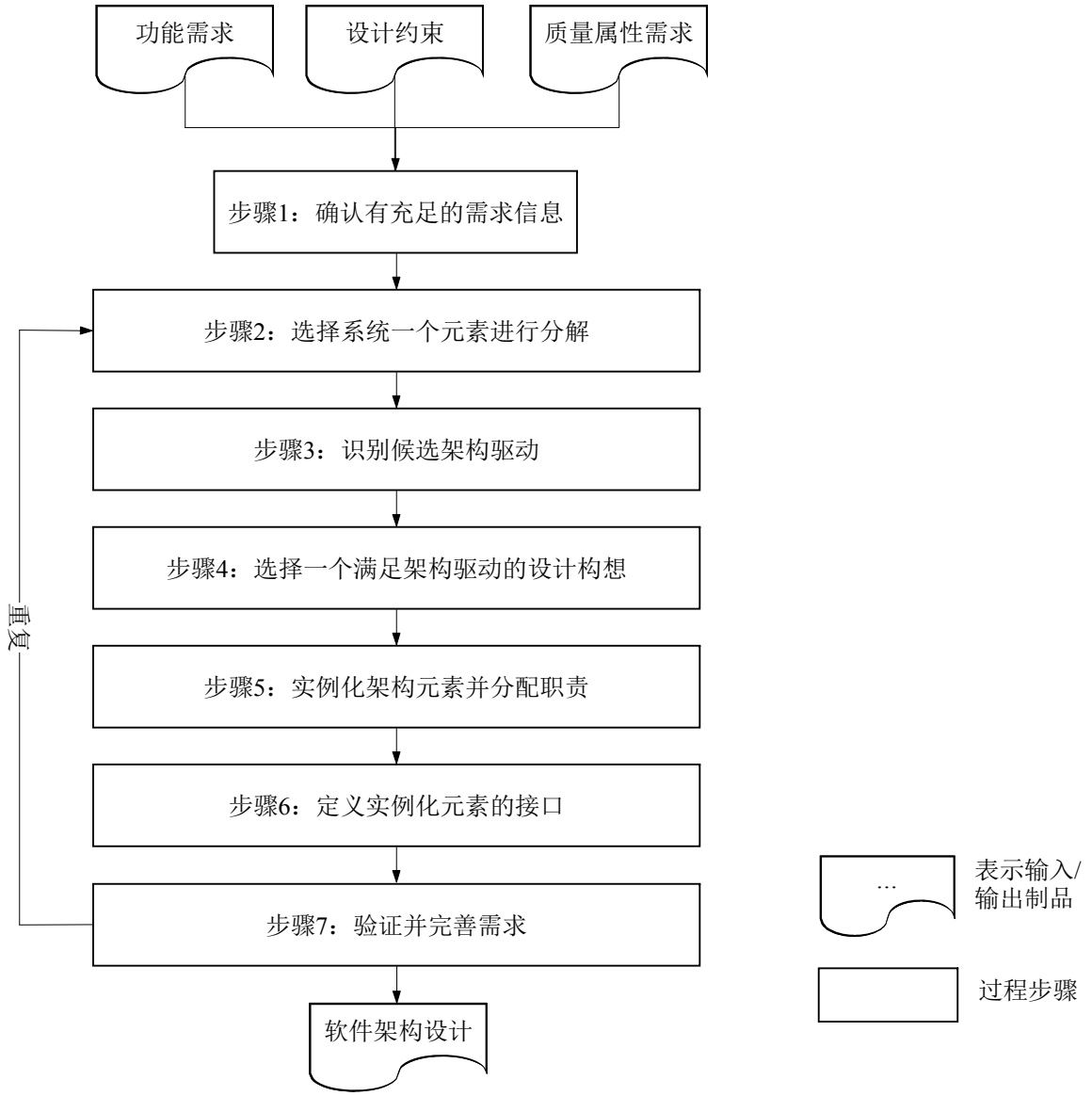
内容

- 5.1 架构为中心的软件开发过程
- 5.2 属性驱动的设计方法
- 5.3 基于模式的设计方法
- 5.4 模块设计与评估方法
- 5.5 软件体系结构评估

基于模式的设计方法概述

- 模式是从软件开发中提取并经充分验证的经验，可再用于促进好的设计实现。
- 无论哪类模式它们都会提供一个功能、行为的基本框架，有助于实现应用程序的功能。此外，它们往往还会描述软件系统的非功能需求，支持采用所定义属性来构造软件。
- 模式是属性驱动设计方法（ADD）中构造高质量软件体系结构的一个重要工具，不但如此基于模式还可独立开展软件体系结构的设计。

回顾：ADD的设计步骤总览



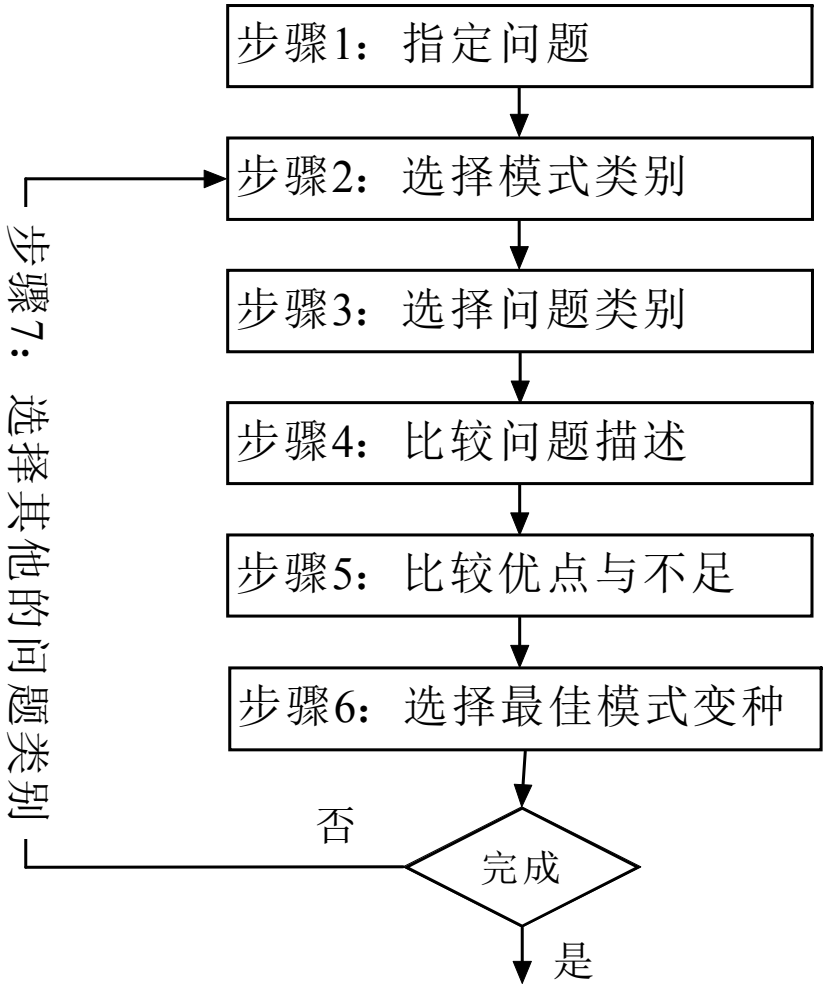
回顾：ADD设计步骤4——选择一个满足架构驱动的设计构想

- 选出架构中的主要元素及其之间的关系：
  - （1）识别候选架构驱动相关的设计问题
  - （2）对于每一个设计问题，创建一个解决该问题的模式列表
  - （3）从模式列表中选出最能满足候选架构驱动的模式
  - （4）考虑目前已经识别的模式，并确定他们之间的关系，将所选中的模式进行组合可以产生新的模式
  - （5）通过不同的架构视图来描述已经选择的模式
  - （6）评估并解决设计构想不一致性的问题

基于模式的设计步骤

- 1.选择一个需要细化设计的构件
- 2.为该构件定义需求：
  - 分析系统的其他部分或者外部系统如何与该构件交互
  - 考虑该构件的哪些部分将对架构的不同部分产生影响
- 3.针对步骤2中定义的需求和交互，**找出最适合的架构风格或模式**
- 4.**使用与问题相匹配的模式**来**指导类和构件的设计**
  - 使用步骤3中的成熟的软件解决方案作为额外的信息，来进一步完善在步骤2中定义的类和构件
- 5.在构件中进行迭代，为每个构件重复步骤2-4

3.针对步骤2中定义的需求和交互，找出最适合的架构风格或模式



步骤3.1 指定问题

- 选择模式的第一步是指定需要解决的问题，指定的问题需要尽可能具体。
- 如果一个问题包括很多不同部分，则可以将该问题分割为一些更小的子问题。
- 对于每一个定义的子问题，考虑该问题对应的约束，这些约束形成了问题的上下文。
- 在考虑了上下文之后，需要分析比较这些问题以得到一个较好的解决方案，这些**问题就是需要做出的权衡**

例如，为一个系统添加错误容忍度能够增加可靠性和可用性，但却造成了较高的代码复杂性、更长的开发时间和更多的执行代码（降低性能）

步骤3.2：选择模式类别

- 在对每个问题定义了子问题之后，需要决定选择哪个模式来解决正处理的问题。针对不同层次的问题选择不同的模式类别：
  - 当在定义系统的基础结构时，需要寻找一种架构模式；
  - 当在构建架构的一些构件（构件级设计）时，需要寻找一种设计模式
  - 例如在设计初始时，我们往往把整个系统按照一种层次架构模式划分成不同的层

步骤3.3：选择问题类别

- 在定义了可以提供解决方案的模式类别之后，需要定义要解决的问题类别

架构模式问题类别	设计模式问题类别	惯用法问题
结构	结构型	Java
分布式系统	创建型	C++
交互系统	行为型	Small memory systems
自适应系统		Ruby
实时系统		Smalltalk
容错系统		Performance tuning

步骤3.4：比较问题描述

- 可以使用步骤3.1中问题的详细描述来进一步缩小模式的范围。**使用详细的问题信息来检查候选模式对应的问题：**
  - （1）判断候选模式的问题是否与正在尝试解决的问题相匹配；
  - （2）判断候选模式是否需要应用其他的模式。如果尚未应用其他模式，则需要考虑是否能够应用这些模式，或者应用了这些模式后会对软件造成怎样的影响；
  - （3）结合问题的结构检查候选模式的结构。考虑是否已将最主要问题划分成了一些小问题，或者是否在划分子问题上走得太远。这些问题的答案将有助于重新定义问题，以使其能够更好的与模式进行匹配。
  - （4）结合上下文检查候选模式，不匹配的模式应该从候选列表中移除。

步骤3.5：比较优点与不足

- 需要进一步检查之前所考虑的模式，重点关注需要进行的**权衡**以及应用这些模式会产生的**后果**:
  - 候选模式是否能够解决问题并满足应用程序的需求，如果无法满足需求，则需要将该模式从考虑中移除
  - 需要考虑模式所产生的影响：应用某个模式后是否能够带来所期望的好处？模式所带来的不足是否是可管理的？应用某个模式能否提供一条清晰的设计路线？应用某种模式是否会给所解决的问题引入更多的问题？应用某种模式是否会对整体设计有负面的影响？
  - 在考虑了优点和不足之后，需要选择一种最佳模式。

步骤3.6：选择最佳模式变种

- 模式有时会包含一些变种，模式变种可以以其他的方式实现解决方案
- 如表示分离风格可考虑使用：
  - MVC
  - MVP
  - MVVM
  - …

步骤3.7：选择其他的问题类别

- 如果无法找出满足问题所有需求的模式，可以尝试重新寻找模式。但是，这次需要拓宽问题类别。

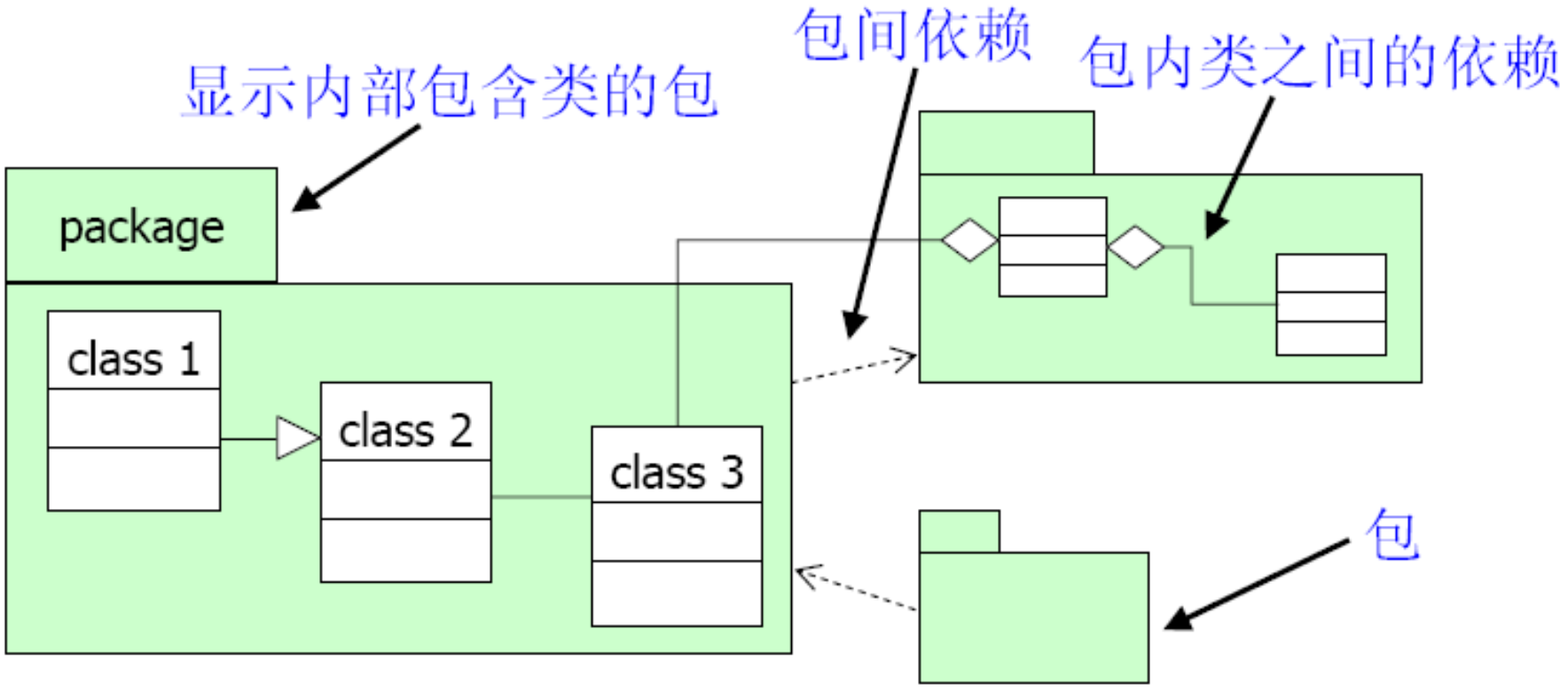
内容

- 5.1 架构为中心的软件开发过程
- 5.2 属性驱动的设计方法
- 5.3 基于模式的设计方法
- 5.4 模块设计与评估方法
- 5.5 软件体系结构评估

模块的种类

- 函数（function）
- 对象与类（object and class）
- 构件（component）
- 服务（service）
- 包（package）

OO中的“模块”种类



模块化设计的目标 (Goals of design)

- Decompose system into components（将系统分解为一组模块）
- Describe component functionality（描述每一个模块的功能）
- Determine relationships between components（决定模块之间的关联关系）
  - Identify component dependencies（识别依赖关系）
  - Determine inter-component communication mechanisms（决定模块之间的通讯机制）
  - Specify component interfaces（确定模块的接口）
  - Interfaces should be well defined to facililate component testing and team communication

什么是一个好的模块化设计？
(What is Good Design？)

- A good design is one that balances trade-offs to minimize the total cost of the system over its entire lifetime (在系统整个生命周期中，通过功能分解降低系统复杂度，以最小化开发和维护成本)
- A matter of avoiding those characteristics that lead to bad consequences (避免那些可能导致坏的后果的性质)

- Need of criteria for evaluating a design (需要标准来评价设计)
- Need of principles and rules for creating good designs (需要原则与规则来指导设计)

模块化的核心技术：Decomposition (分解，即“分治”)

- Why?
  - Handle complexity by splitting large problems into smaller problems, i.e. “divide and conquer” methodology (分而治之，将复杂大问题分解为小问题)
- Steps:
  - Select a piece of the problem (initially, the whole problem) (选定问题)
  - Determine the components in this piece using a design paradigm (分解得到模块)
  - Describe the components interactions (描述模块的交互)
  - Repeat steps 1 through 3 until some termination criteria is met (重复上述步骤，直到满足了终止标准为止)

Benefits of Modularity
(模块化的优点)

- Modularity facilitates software quality factors, e.g.:
  - Extensibility (可扩展性)
    - well-defined, abstract interfaces
  - Reusability (可复用性)
    - low-coupling, high-cohesion
  - Portability (可移植性)
    - hide machine dependencies
- Modularity is important for good design since it
  - Enhances for separation of concerns (保证分离性)
  - Enables developers to reduce overall system complexity via decentralized software architectures (降低系统复杂度)
  - Increases scalability by supporting independent and concurrent development by multiple personnel (增加系统伸缩性)

模块化对满足和改善
SA的非功能需求具有重要意义

5.4 模块设计与评估方法

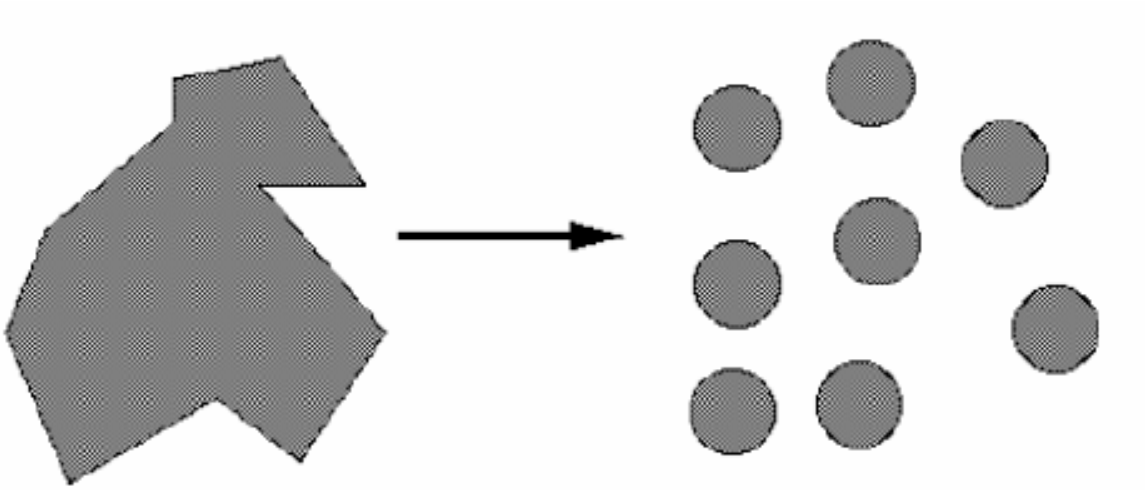
- **5.4.1 模块化的五大评价标准**
- 5.4.2 模块化的五大规则
- 5.4.3 模块化设计的基本原则

Five Criteria for Evaluating Modularity
（模块化的五大评价标准）

- Decomposability (可分解性)
  - Are larger components decomposed into smaller components?
- Composability (可组合性)
  - Are larger components composed from smaller components?
- Understandability (可理解性)
  - Are components separately understandable?
- Continuity (可持续性)
  - Do small changes to the specification affect a localized and limited number of components?
- Protection (出现异常之后的保护)
  - Are the effects of run-time abnormalities confined to a small number of related components?

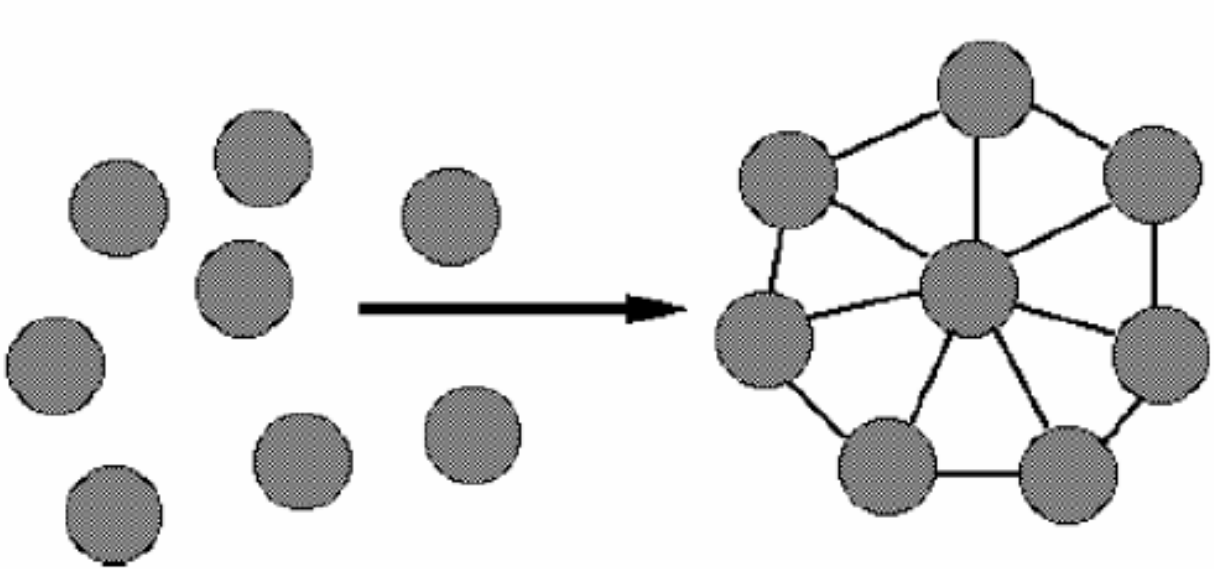
1. Decomposability

- Decompose problem into smaller sub-problems that can be solved separately (将问题分解为各个可独立解决的子问题)
  - Goal: keep dependencies explicit and minimal (目标：使模块之间的依赖关系显式化和最小化)



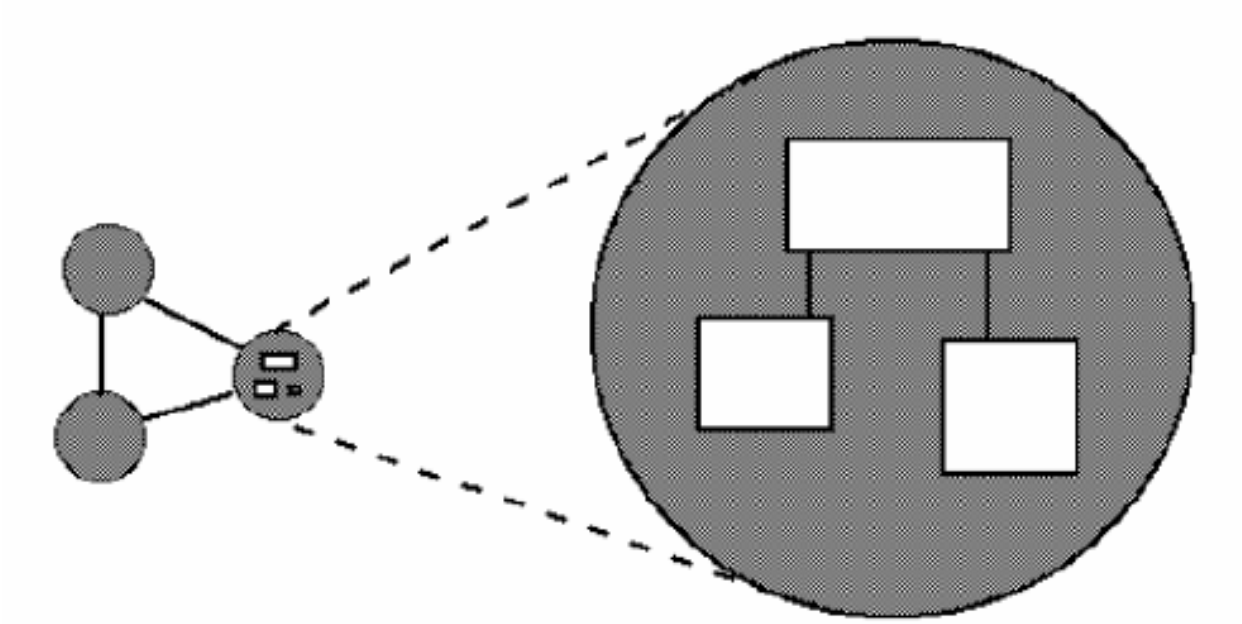
2. Composability

- Freely combine modules to produce new systems (可容易的将模块组合起来形成新的系统)
  - Goal: make modules reused in different environments (目标：使模块可在不同的环境下复用)
  - Example: Math libraries; UNIX command & pipes



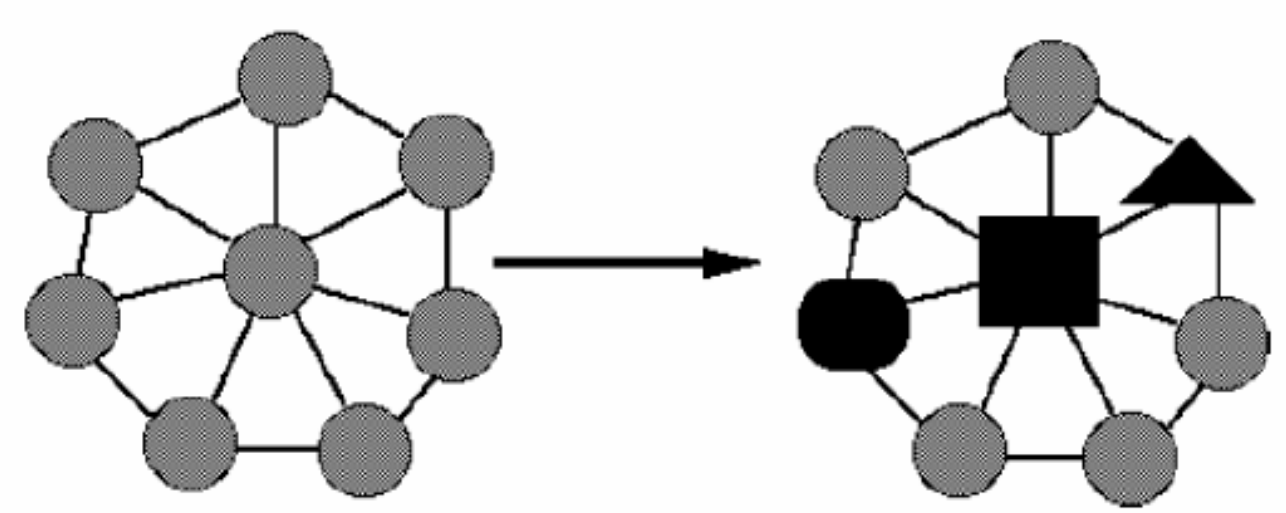
3. Understandability

- Individual modules understandable by human reader (每个子模块都可被系统设计者容易的理解)
  - Counter-example: Sequential Dependencies (A →B → C)



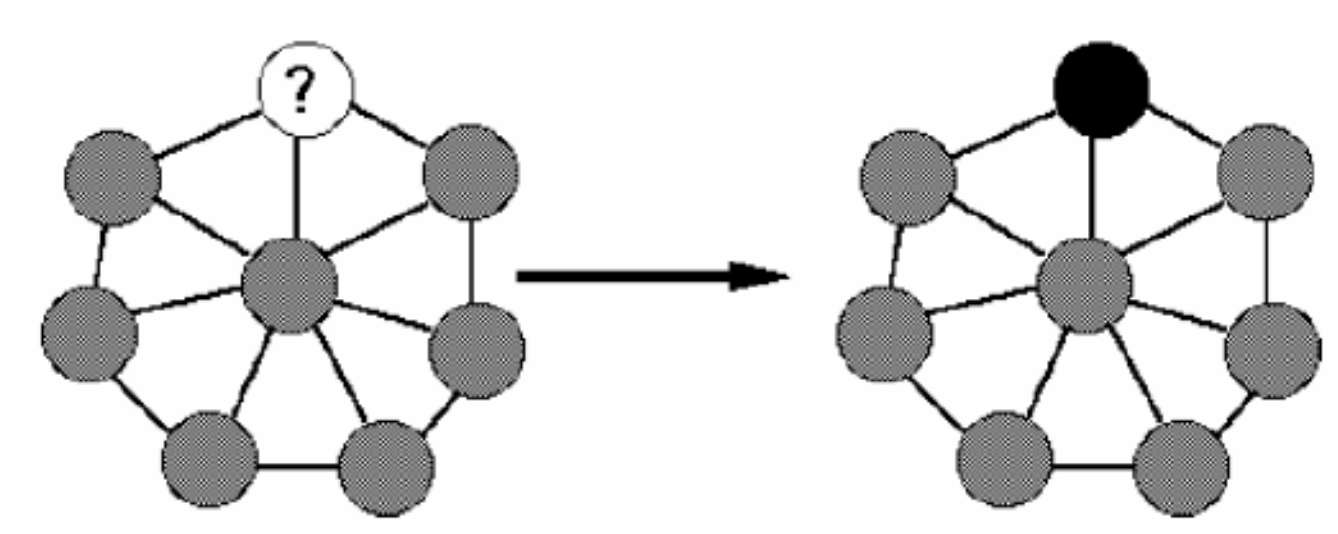
4. Continuity

- Small change in specification results in:
  - changes in only a few modules and does not affect the architecture (小的变化将只影响一小部分模块，而不会影响整个体系结构)
  - Example: Symbolic Constants (符号型变量)
    - const String PRODUCT_CODE = “PBS001291A”



5. Protection

- Effects of an abnormal run-time condition is confined to a few modules (运行时的不正常将局限于小范围模块内)
  - Example: Validating input at source



5.4 模块设计与评估方法

- 5.4.1 模块化的五大评价标准
- **5.4.2 模块化的五大规则**
- 5.4.3 模块化设计的基本原则

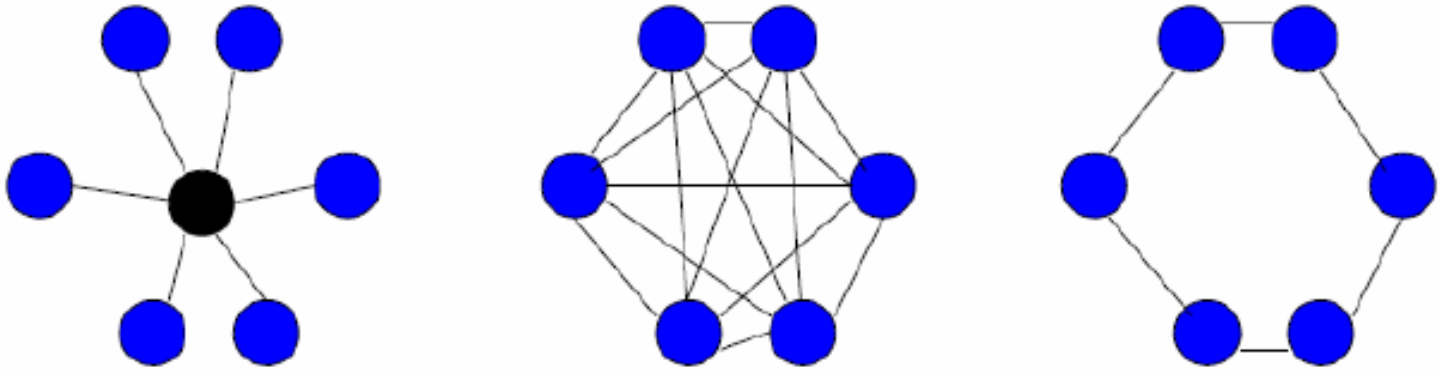
1. Direct Mapping（直接映射）

- Direct mapping: keep the structure of the solution compatible with the structure of the modeled problem domain (直接映射：模块的结构与现实世界中问题领域的结构保持一致)

- Impact on (对以下评价标准产生影响):
  - Continuity (持续性)
    - easier to assess and limit the impact of change
  - Decomposability (可分解性)
    - decomposition in the problem domain model as a good starting point for the decomposition of the software

2. Few Interfaces（少的接口）

- Every module should communicate with as few others as possible (模块应尽可能少的与其他模块通讯)
  - 通讯路径的数目: n-1, n(n-1)/ 2, n-1
  - affects Continuity, Protection, Understandability and Composability (对以下评价标准产生影响：可持续性、保护性、可理解性、可组合性)



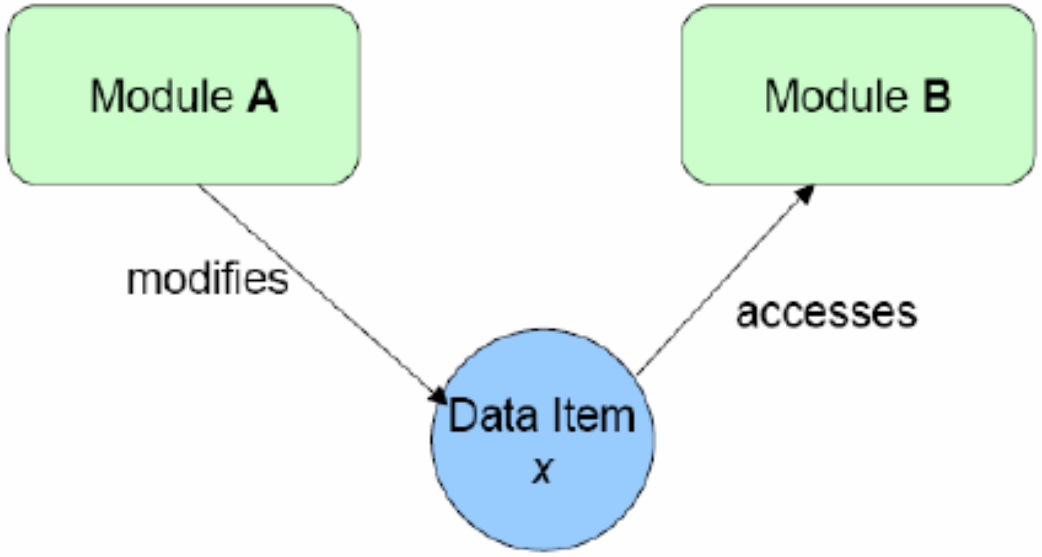
3. Small Interfaces（小的接口）

- If two modules communicate, they should exchange as little information as possible (如果两个模块通讯，那么它们应交换尽可能少的信息)
  - limited “bandwidth” of communication (限制模块之间通讯的“带宽”)
  - Continuity and Protection (对“可持续性”和“保护性”产生影响)

4. Explicit Interface（显式接口）

- Whenever two modules A and B communicate, this must be obvious from the text of A or B or both (当A与B通讯时，应明显的发生在A与B的接口之间)
  - Decomposability, Composability, Continuity, Understandability (受影响的评价标准：可分解性、可组合性、可持续性、 可理解性)

反例：

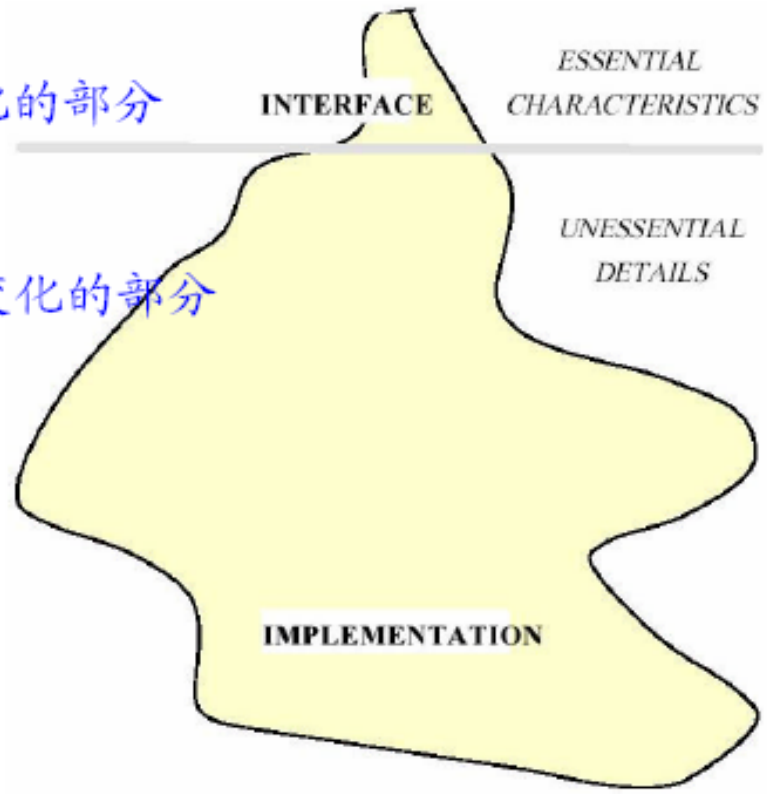


…再次强调Rule

- (1)Few Interfaces: “Don't talk to many!”
  尽可能少的接口：“不要对太多人讲话...”
- (2)Small Interfaces: “Don't talk a lot!”
  尽可能小的接口：“不要讲太多...”
- (3)Explicit Interfaces: “Talk loud and in public! Don't whisper!”
  显式接口：“公开的大声讲话...不要私下嘀咕...”

5. Information Hiding

- Motivation: design decisions that are subject to change should be hidden behind abstract interfaces (经常可能发生变化的设计决策应尽可能隐藏在抽象接口后面)
  - Impact on Continuity (影响"可持续性")



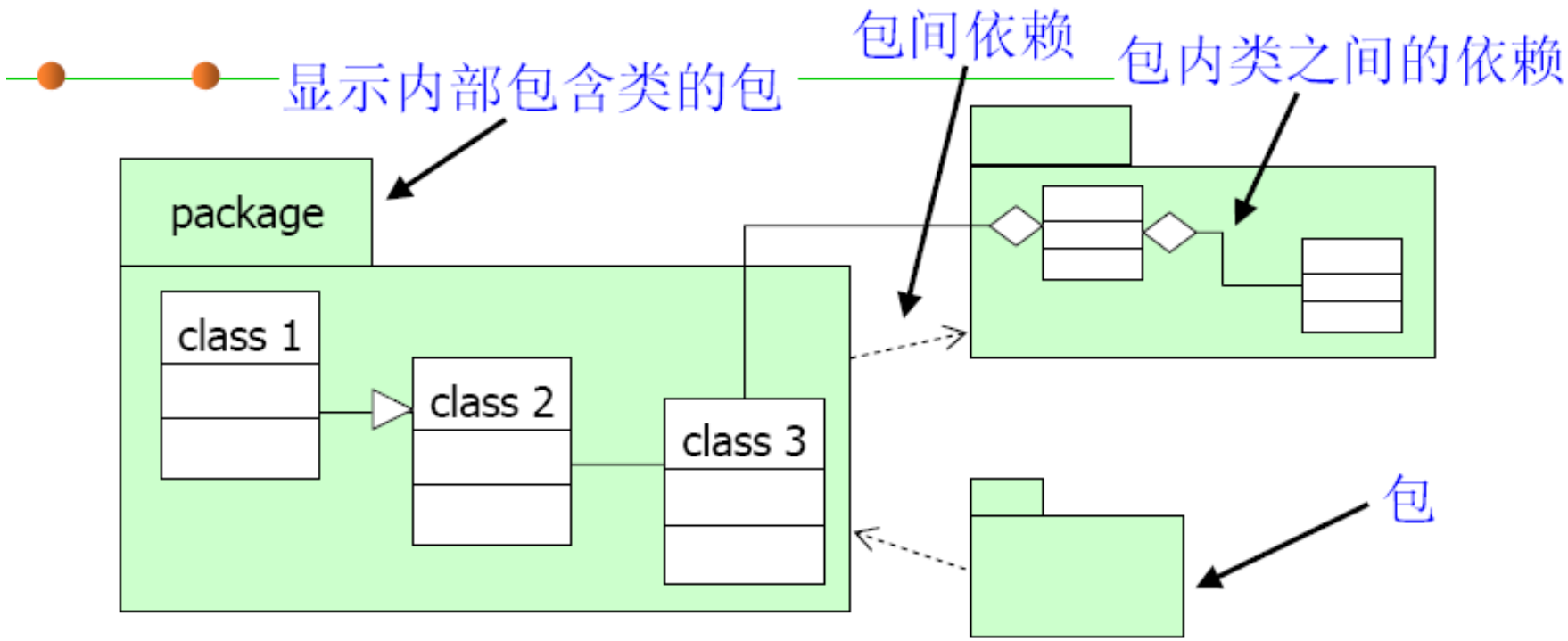
5.4 模块设计与评估方法

- 5.4.1 模块化的五大评价标准
- 5.4.2 模块化的五大规则
- 5.4.3 模块化设计的基本原则

5.4.3 模块化设计的基本原则

- 1.类设计原则
- 2.包聚合设计原则
- 3.包耦合设计原则

模块化设计的基本原则



- Class Design Principles (类设计原则)
- Principles of Package Cohesion (包聚合设计原则)
- Principles of Package Coupling (包耦合设计原则)

1. Class Design Principles (类设计原则)

- (SRP) The Single Responsibility Principle
  单一责任原则
- (OCP) The Open-Closed Principle
  开放-封闭原则
- (LSP) The Liskov Substitution Principle
  Liskov替换原则
- (DIP) The Dependency Inversion Principle
  依赖转置原则
- (ISP) The Interface Segregation Principle
  接口聚合原则

(SRP) The Single Responsibility Principle
单一责任原则

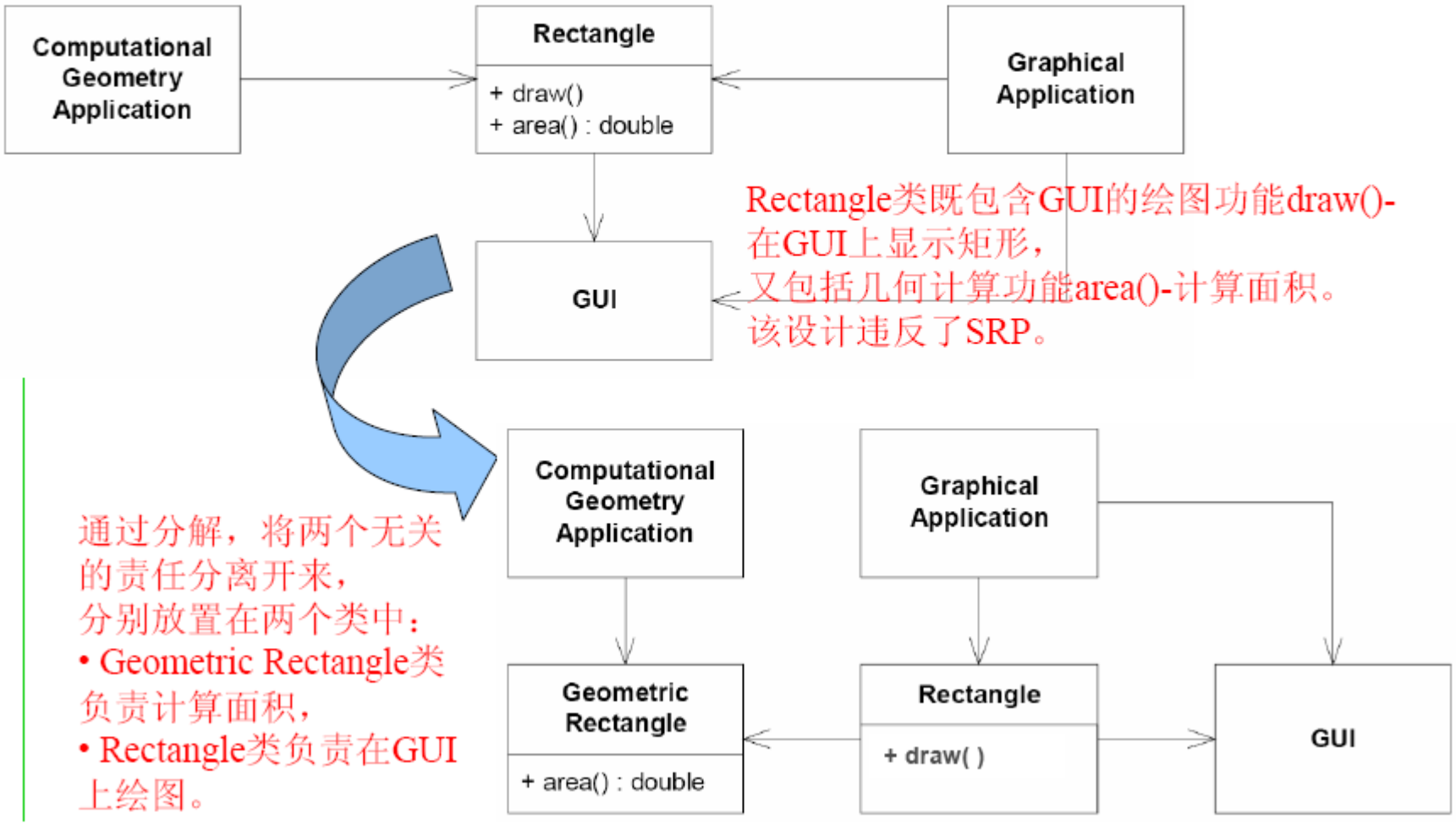
- Responsibility: “a reason for change.” (责任：变化的原因)
- SRP:
  - There should never be more than one reason for a class to change. (不应有多于1个的原因使得一个类发生变化)
  - One class, one responsibility. (一个类，一个责任)

- 如果一个类包含了多个责任，那么将引起不良后果：
  - 引入额外的包，占据资源
  - 导致频繁的重新配置、部署等

- The SRP is one of the simplest of the principle, and one of the hardest to get right. (最简单的原则，却是最难做好的原则)

SRP的一个反例

SRP的一个反例



(OCP) The Open-Closed Principle
开放封闭原则

- Software entities (classes, modules, functions, etc.) should be open for extension, (对扩展性的开放)
  - This means that the behavior of the module can be extended That we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications. (模块的行为应是可扩展的，从而该模块可表现出新的行为以满足需求的变化)

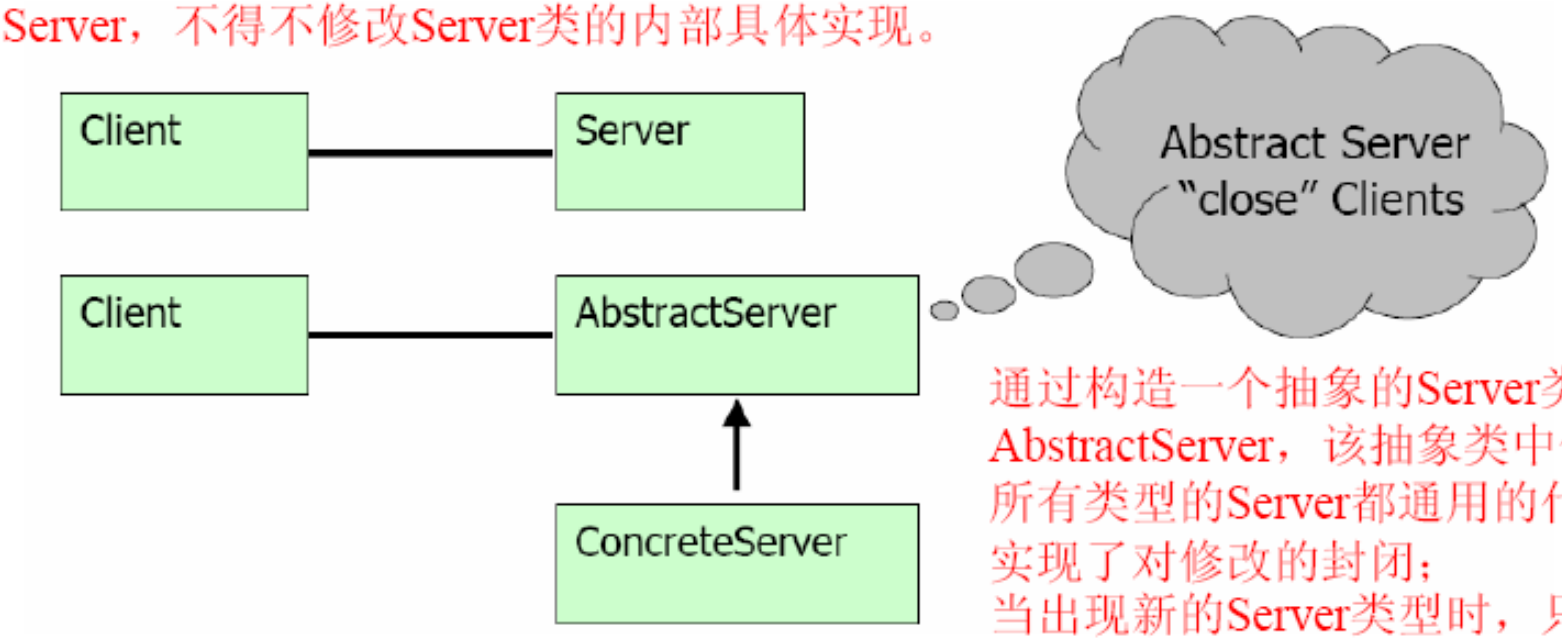
- but closed for modification. (对修改的封闭)
  - The source code of such a module is inviolate. No one is allowed to make source code changes to it. (但模块自身的代码是不应被修改的)

(OCP) The Open-Closed Principle
开放封闭原则（续）

- It would seem that these two attributes are at odds with each other. (二者看起来有些矛盾)
  - The normal way to extend the behavior of a module is to make changes to that module. (扩展模块行为的一般途径是修改模块的内部实现)
  - A module that cannot be changed is normally thought to have a fixed behavior. (如果一个模块不能被修改，那么它通常被认为是具有固定的行为)
- Key: abstraction (关键的解决方案：抽象技术)

OCP的一个反例

如果有多种类型的Server，那么针对每一种新出现的Server，不得不修改Server类的内部具体实现。



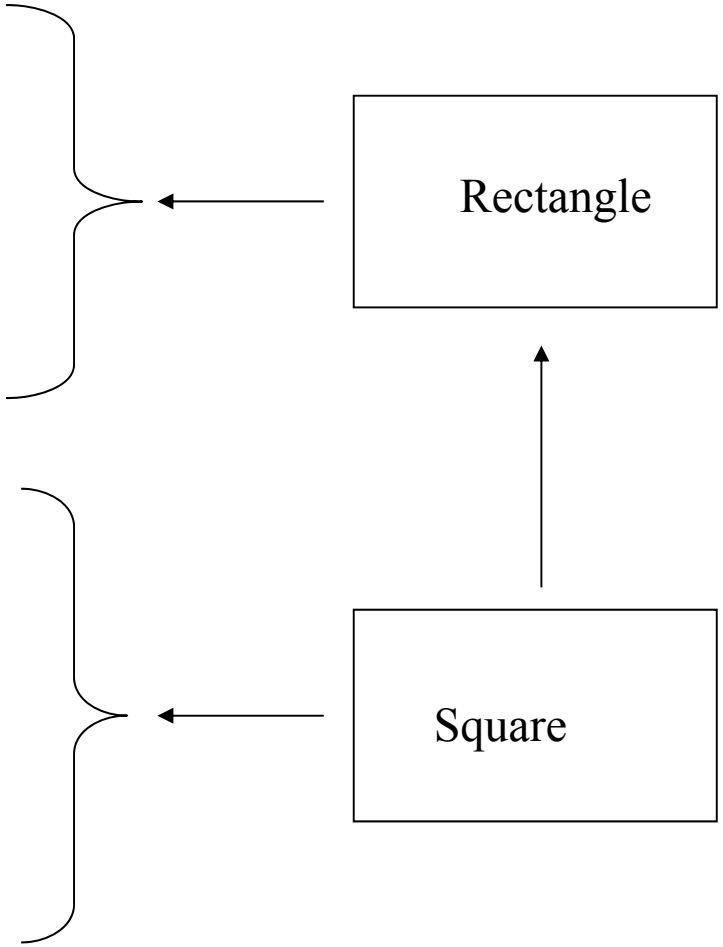
通过构造一个抽象的Server类：AbstractServer，该抽象类中包含针对所有类型的Server都通用的代码，从而实现了对修改的封闭；
当出现新的Server类型时，只需从该抽象类中派生出具体的子类ConcreteServer即可，从而支持了对扩展的开放。

(LSP) The Liskov Substitution Principle (Liskov替换原则)

- LSP: Subtypes must be substitutable for their base types. (子类型必须能够替换其基类型)
- Derived Classes must be usable through the base class interface without the need for the client to know the difference. (派生类必须能够通过其基类的接口使用，客户端无需了解二者之间的差异)

LSP的一个例子

```
class Rectangle
{
public:
void    SetWidth(double w) {itsWidth=w;}
void    SetHeight(double h) {itsHeight=h;}
double GetHeight() const {return itsHeight;
double GetWidth() const {return itsWidth;}
private:
double itsWidth;
double itsHeight;
};
class Square : Rectangle
{void Square::SetWidth(double w)
{       Rectangle::SetWidth(w);
        Rectangle::SetHeight(w);}
}
void Square::SetHeight(double h)
{
        Rectangle::SetHeight(h);
        Rectangle::SetWidth(h);
}}
```

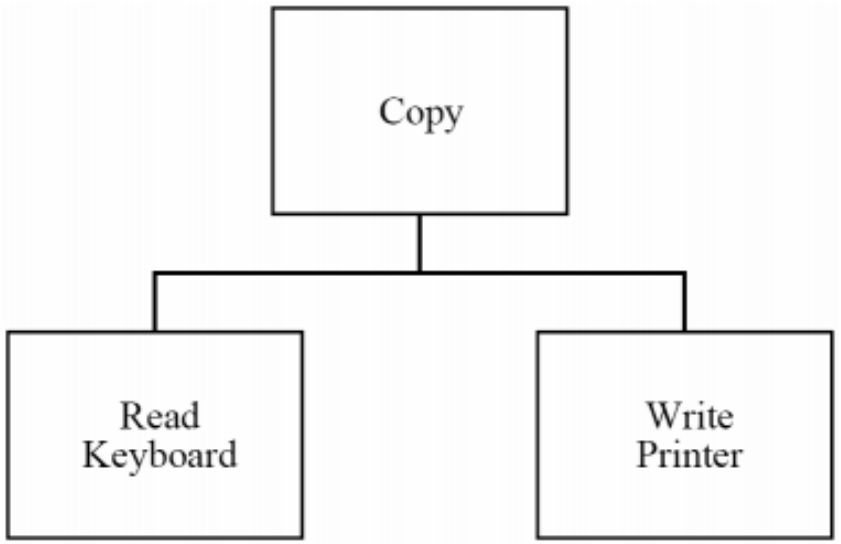


(DIP) The Dependency Inversion Principle
依赖倒置原则

- Abstractions should not depend upon details (抽象的模块不应依赖于具体的模块)
- Details should depend upon abstractions (具体应依赖于抽象)

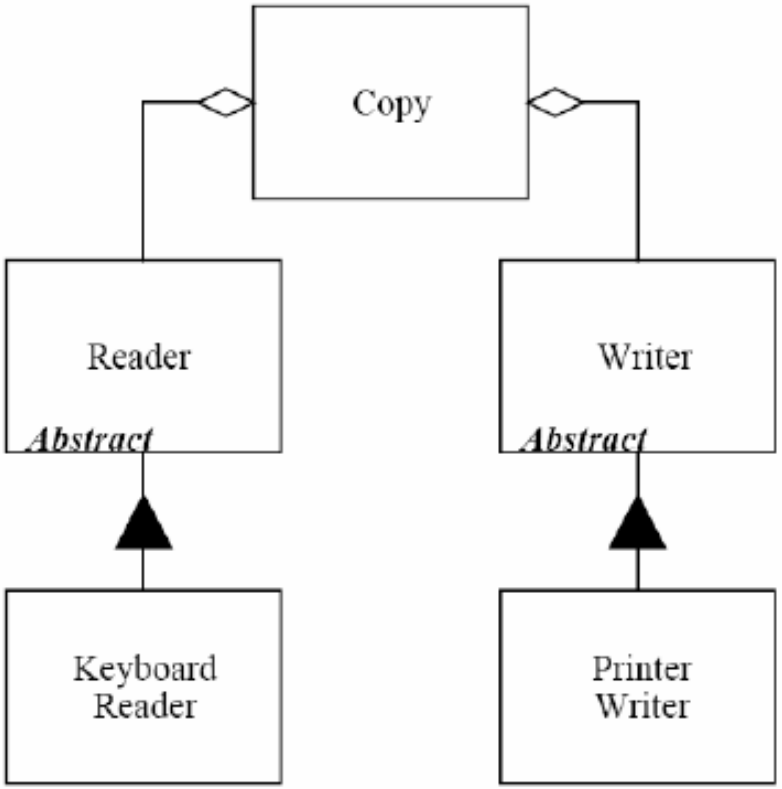
Example: the “Copy” program

```
void Copy()
{
      int c;
      while ((c = ReadKeyboard()) != EOF)
      WritePrinter(c);
}
```




```
enum OutputDevice {printer, disk};
void Copy(outputDevice dev)
{
      int c;
      while ((c = ReadKeyboard()) != EOF)
      if (dev == printer)
          WritePrinter(c);
      else
          WriteDisk(c);
}
```

New solution to the example



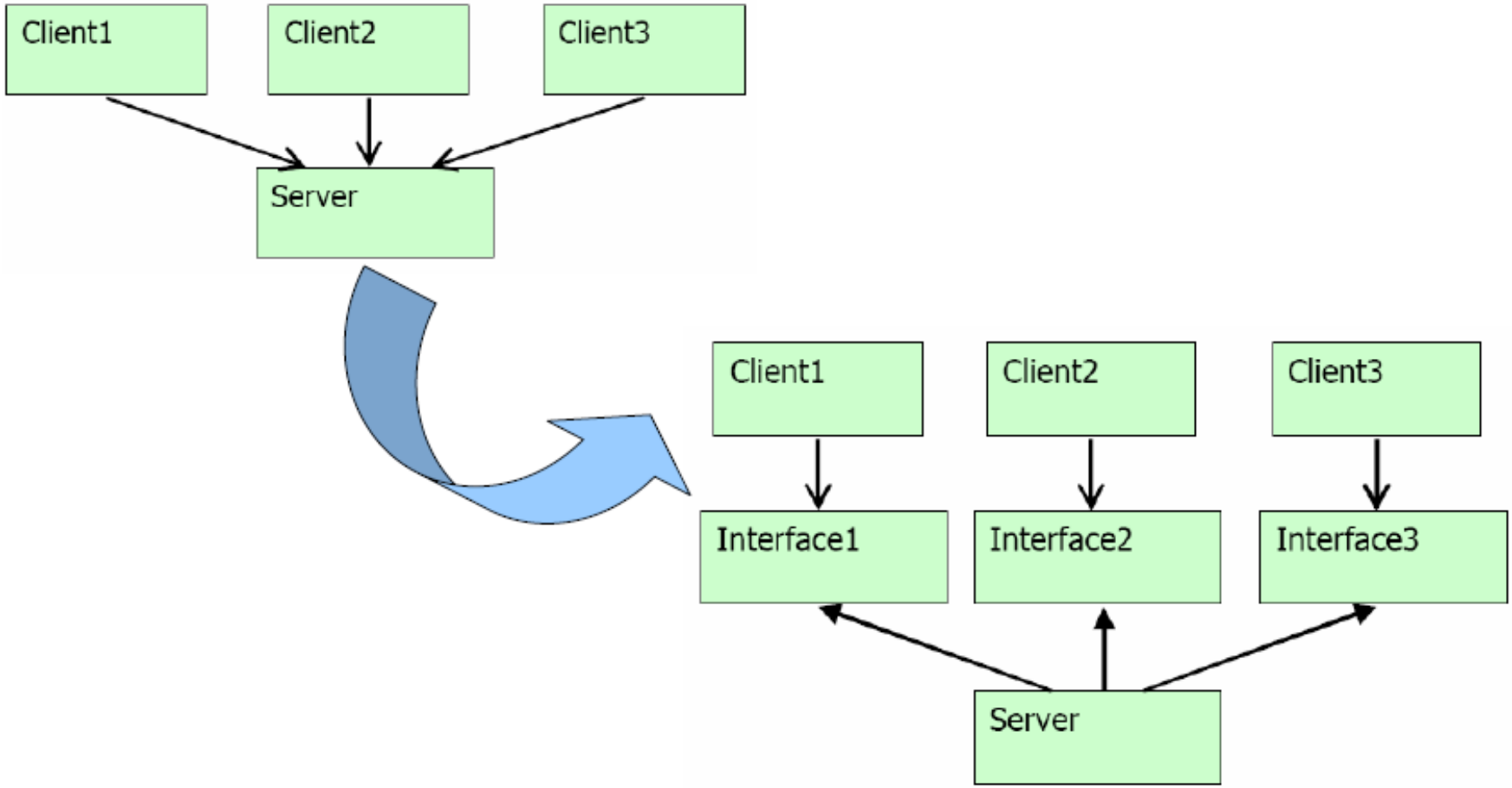

```
class Reader
{
  public:
    virtual int Read() = 0;
};

class Writer
{
  public:
    virtual void Write(char) = 0;
};

void Copy(Reader& r, Writer& w)
{
  int c;
  while((c=r.Read()) != EOF)
    w.Write(c);
}
```

(ISP) The Interface Segregation Principle
接口隔离原则

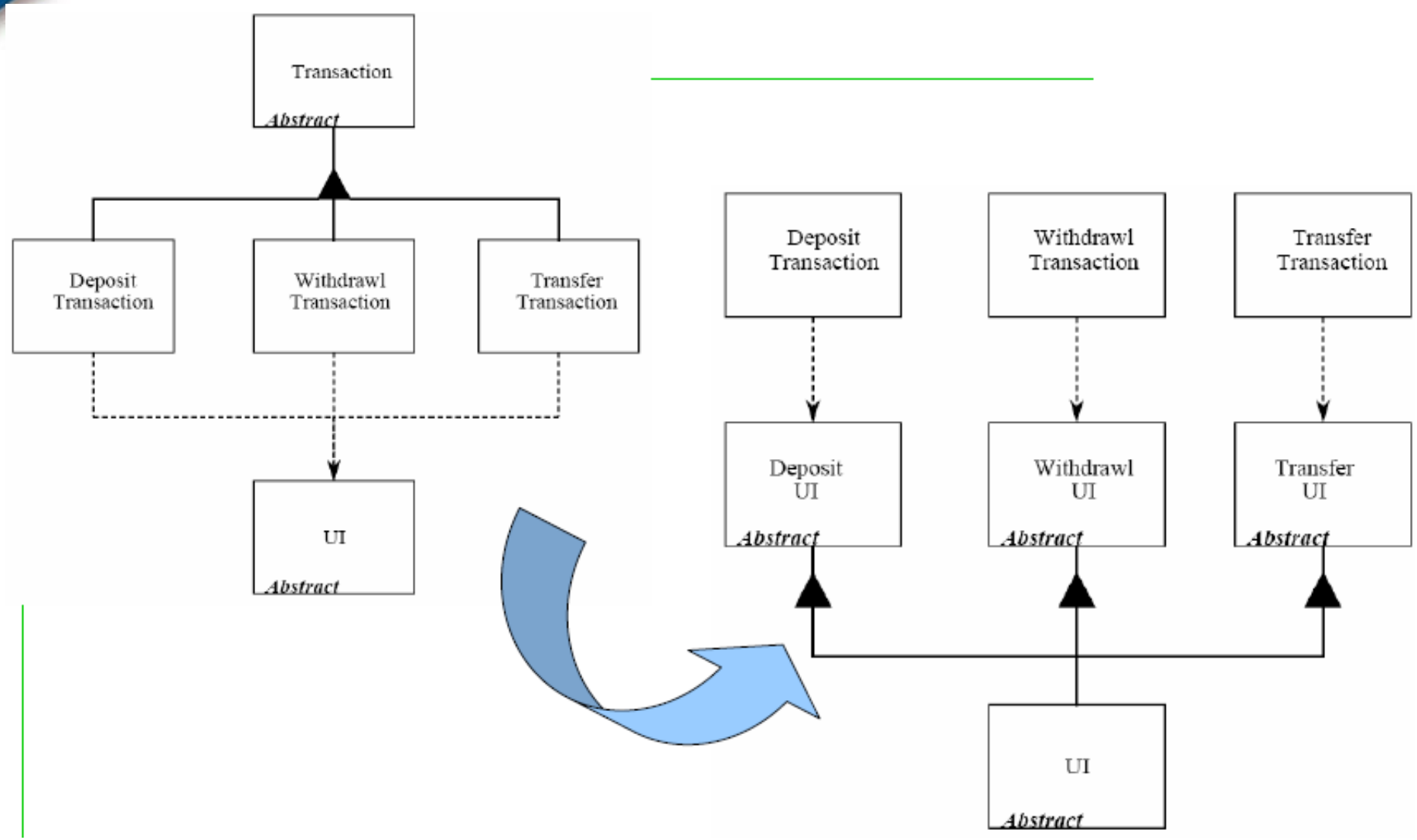
- Clients should not be forced to depend upon methods that they do not use. (客户端不应依赖于它们不需要的方法)
- Interfaces belong to clients, not to hierarchies.
- This principle deals with the disadvantages of "fat" interfaces. ("胖"接口具有很多缺点)
- Classes that have "fat" interfaces are classes whose interfaces are not cohesive. (不够聚合)
- The interfaces of the class can be broken up into groups of member functions. (胖接口可分解为多个小的接口)
- Each group serves a different set of clients (不同的接口向不同的客户端提供服务)
- Thus some clients use one group of member functions, and other clients use the other groups. (客户端只访问自己所需要的端口)

奥卡姆剃刀定律（**Occam's Razor, Ockham's Razor**）

(ISP) The Interface Segregation Principle
接口隔离原则



The ATM User Interface Example



2. Principles of Package Cohesion (包聚合设计原则)

- (REP) The Reuse/Release Equivalency Principle

复用/发布等价原则

- (CCP) The Common Closure Principle

共同封闭原则

- (CRP) The Common Reuse Principle

共同复用原则

(REP) The Reuse/Release Equivalency Principle
复用/发布等价原则

- 一个可重用的元件（组件、一个类、一组类等），只有在它们被某种发布（Release）系统管理以后，才能被重用。体系结构师应该将可重用的类都放在包中

• The granule of reuse is the granule of release. (复用的粒度应等价于发布的粒度)
• Single Classes are seldom reusable
• Unreleased modules cannot be reused
• So the granularity of reuse is the granularity of release

(CCP) The Common Closure Principle
共同封闭原则

- 尽量减少在产品的发布周期中被改动的包的数量，这就要求我们将一起变化的类放在一起（同一个包）

The classes in a package should be closed together against the same kinds of changes.（一个包中的所有类针对同一种变化是封闭的）
A change that affects a closed package affects all theclasses in that package and no other packages.（一个包的变化将会影响包里所有的类，而不会影响到其他的包）
If two classes are so tightly bound together, either physically or conceptually, such that the almost always change together; then they should belong to the same package.（如果两个类紧密耦合在一起，即二者总是同时发生变化，那么它们就应属于同一个包）

(CRP) The Common Reuse Principle
共同复用原则

- 对一个包的依赖就是对包里面所有东西的依赖。换句话说，避免因一个和我们无关的类的改变也产生包的一个新版本，从而我们被强迫升级这个包

• The classes in a package are reused together. (一个包里的所有类应被一起复用)

• If you reuse one of the classes in the package, you reuse them all. (如果复用了其中一个类，那么就应复用所有的类)

3. Principles of Package Coupling (包耦合设计原则)

- (ADP) The Acyclic Dependencies Principle

无圈依赖原则

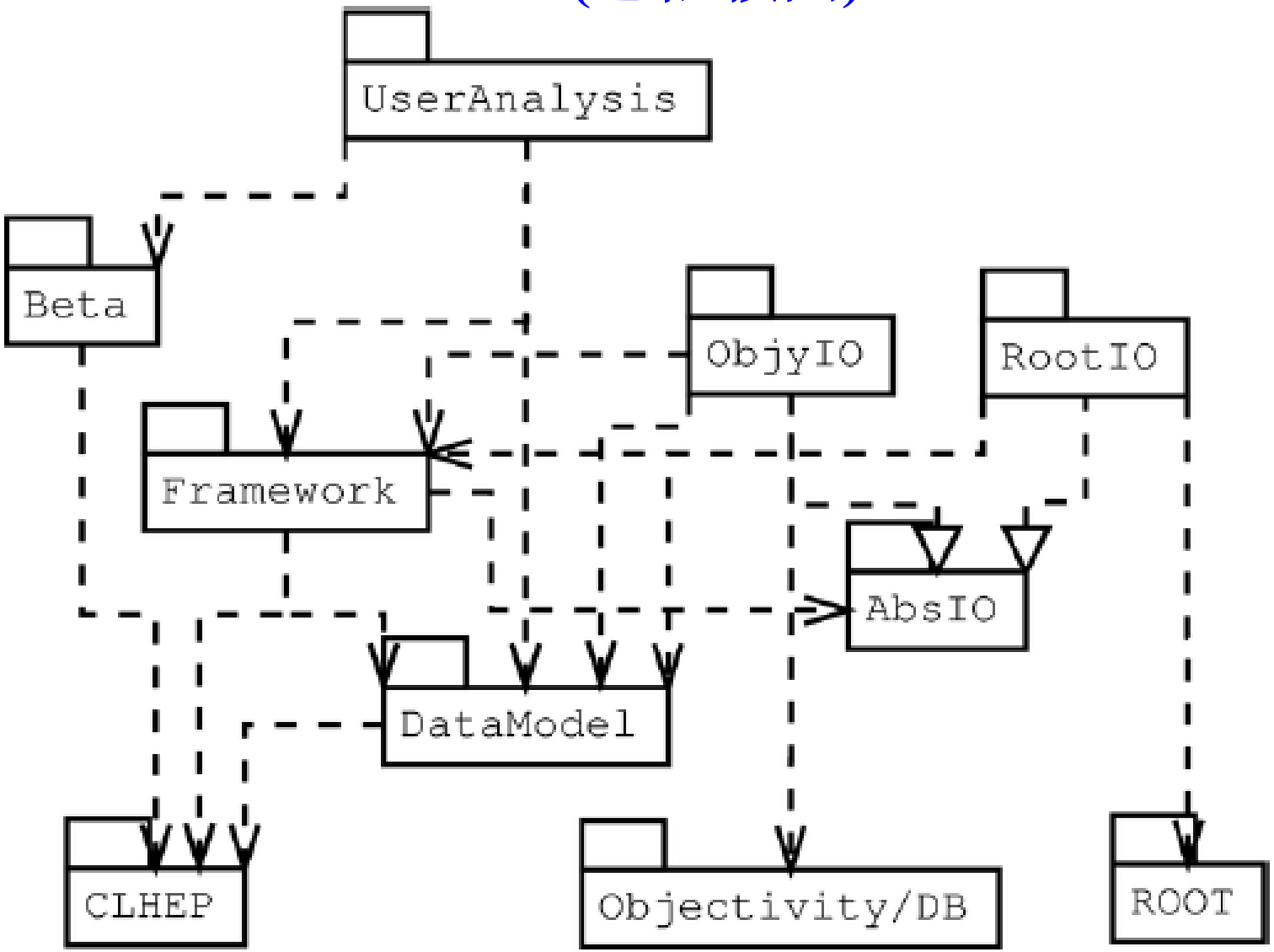
- (SDP) The Stable Dependencies Principle

稳定依赖原则

- (SAP) The Stable Abstraction Principle

稳定抽象原则

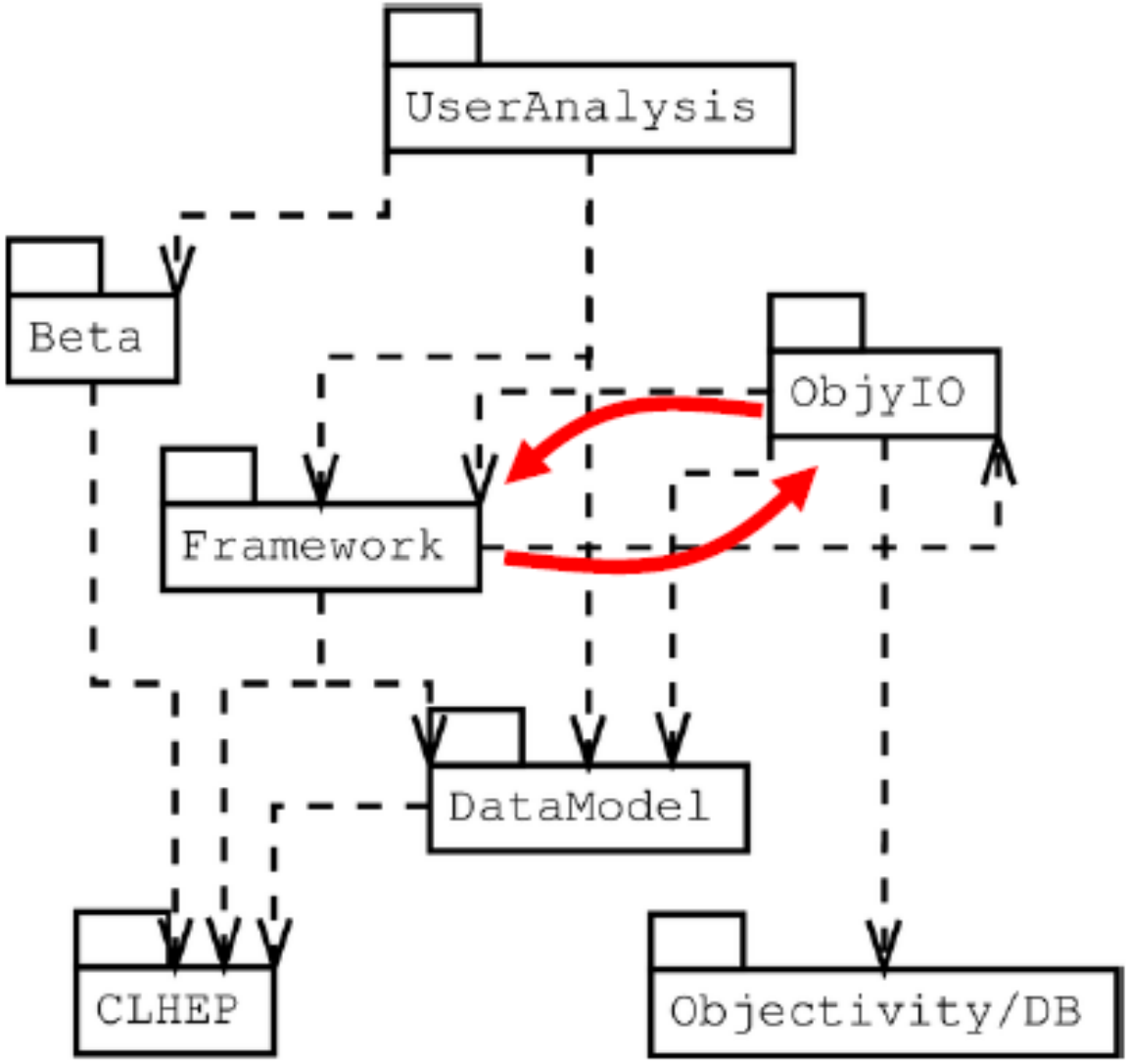
Dependency graph between packages (包依赖图)



(ADP) The Acyclic Dependencies Principle
无圈依赖原则

- Allow no cycles in the package dependency graph.
（不允许在包依赖图中出现任何圈/回路）
- Packages that adhere to the acyclic dependency principle are typically easier to unit test, maintain and understand.（无圈将容易进行测试、维护与理解）
- Cyclic dependencies make it more difficult to predict what the effect of changes in a package are to the rest of the system.（若存在回路依赖，很难预测该包的变化将会如何影响其他包，会出现“先有鸡，先有蛋”这样的问题）

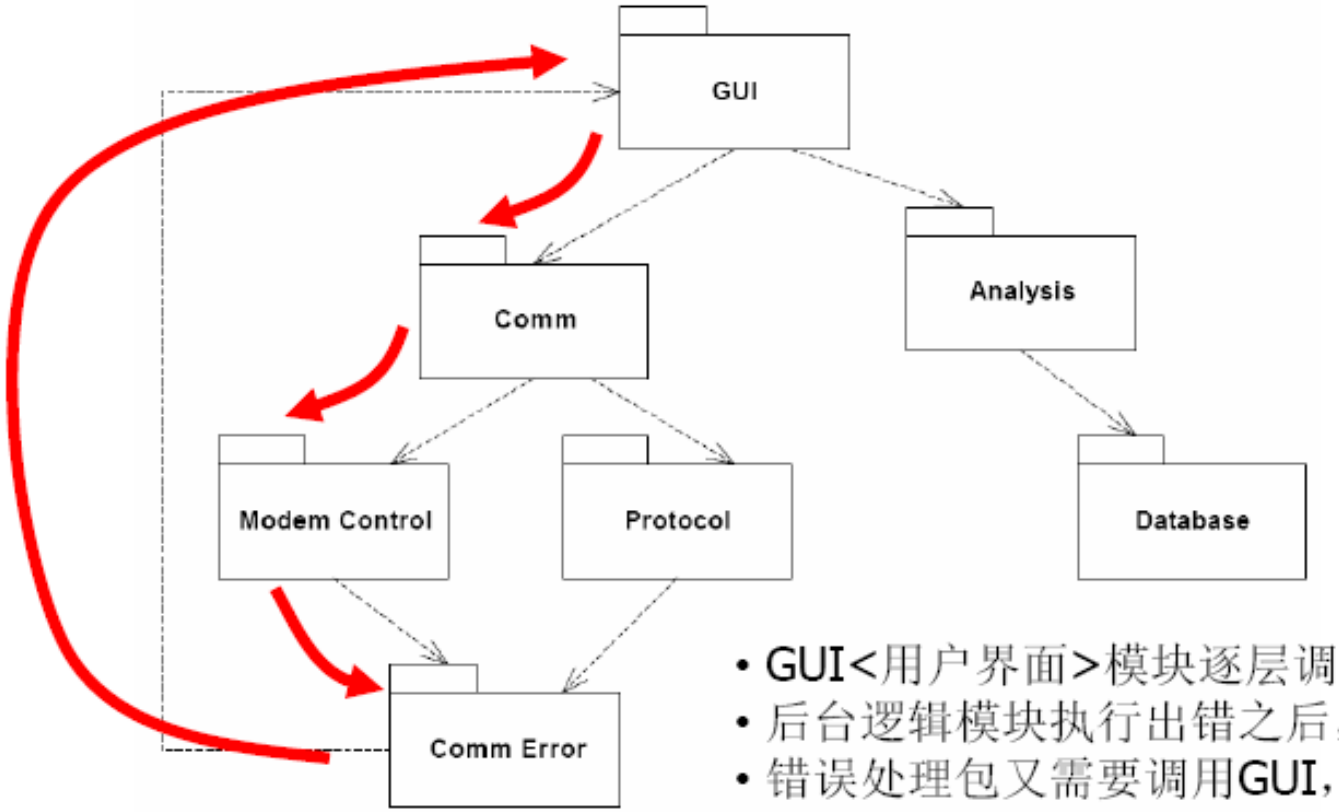
Dependency Cycles



Breaking a Cycle

- Cycles can be broken in two ways (消除圈的两种方式)
  - creating a new package (创建新包)
  - makes use of the DIP and ISP (利用DIP<依赖倒置原则>和ISP<接口隔离原则>)

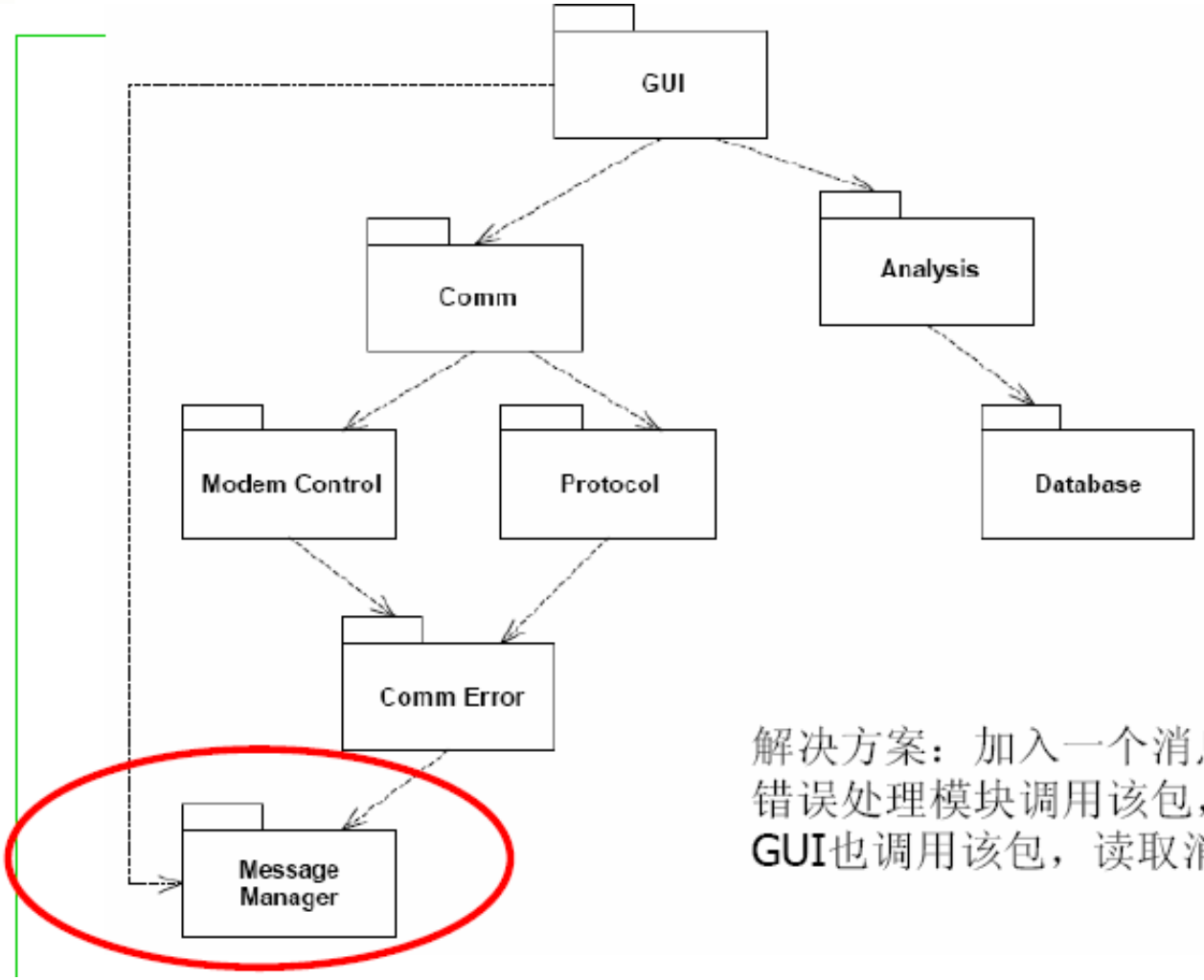
Approach (1):creating new packages



- GUI<用户界面>模块逐层调用后台逻辑模块；
- 后台逻辑模块执行出错之后，调用错误处理模块；
- 错误处理包又需要调用GUI，显示错误消息；

——从而形成包依赖图中的“圈”

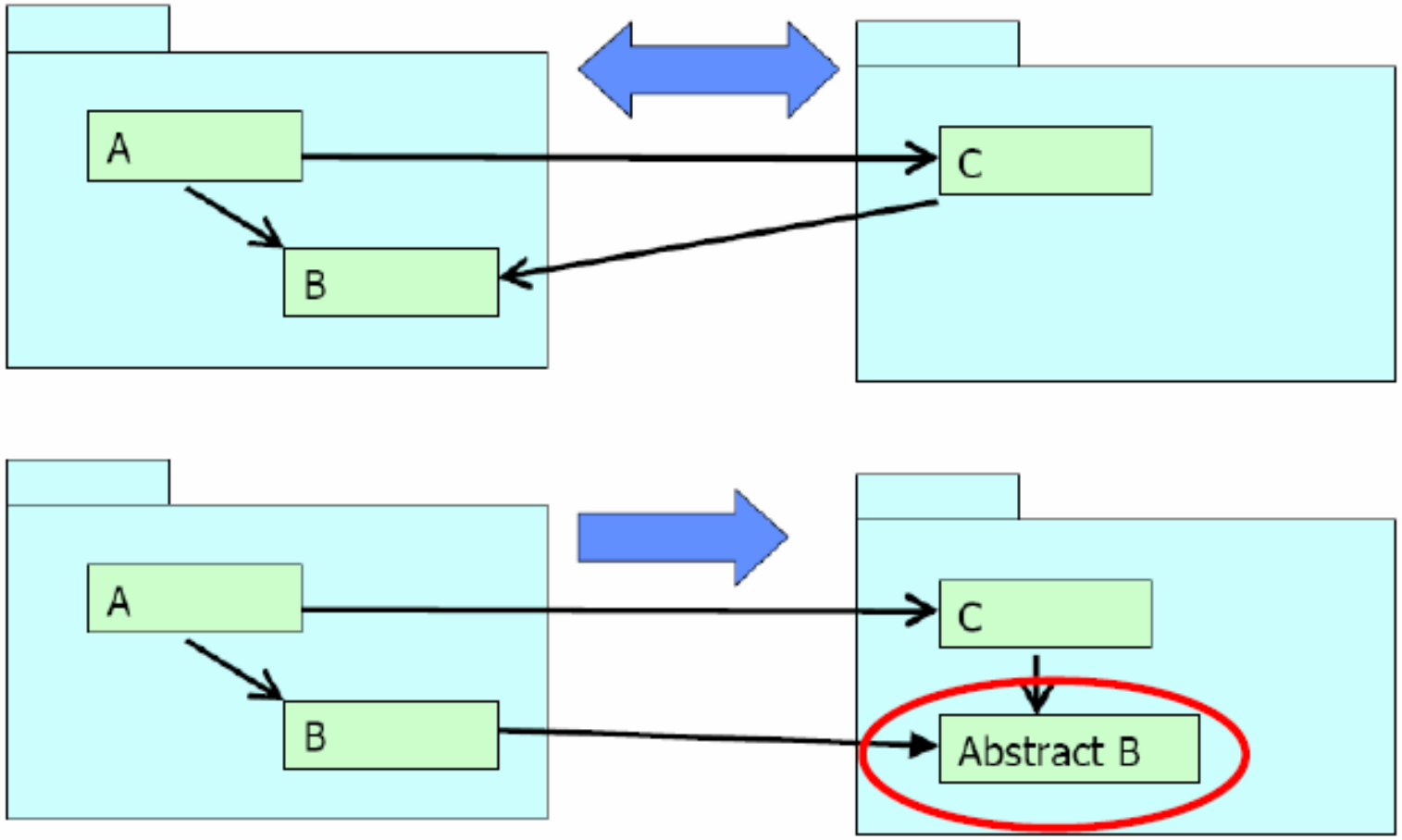
Approach (1):creating new packages



解决方案：加入一个消息管理的包，
错误处理模块调用该包，将错误消息发送过去；
GUI也调用该包，读取消息，并在UI上显示。

Approach (2):Using DIP and ISP

双向依赖→形成圈



创建抽象类，将依赖的方向加以翻转，从而消除圈

(SDP) The Stable Dependencies Principle
稳定依赖原则

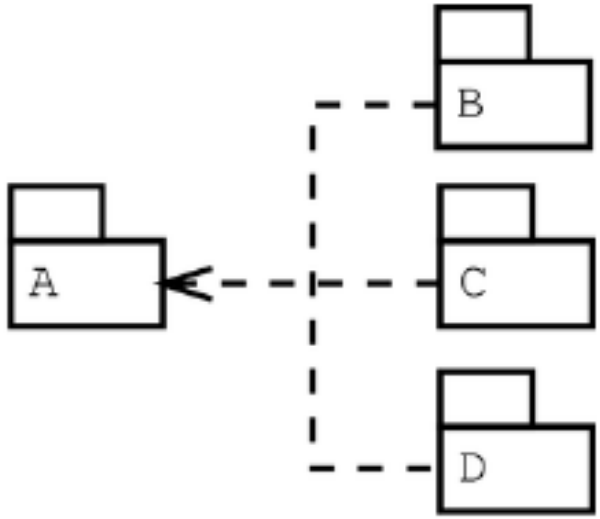
- Dependencies between released packages must run in the direction of stability.（包之间的依赖关系只能指向稳定的方向）
- The dependee must be more stable than the depender.（被依赖者应更稳定于依赖者）
- Stable packages are packages that are difficult to change.（稳定的包较难发生改变）
- Unstable packages that are used a lot by other packages are potential problem areas in a design.（如果不稳定的包却被很多其他包依赖，会导致潜在的问题）

Stability度量方法

- Ca (Afferent Coupling，传入依赖): the number of classes outside the package depending on classes within the package. (其他包中的类所依赖于该包内的类数目)
- Ce (Efferent Coupling，输出依赖): the number of classes outside the package that classes within the package depend on. (该包中的类依赖于其他包中的类的数目)

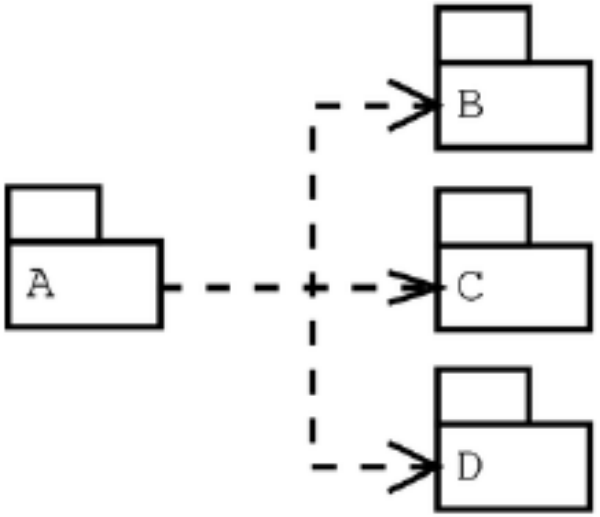
- I (Instability，不稳定度) = Ce / (Ce+Ca)
  - 0 —— Ultimately stable
  - 1 —— Ultimately instable

Stability度量方法



A is a stable package, many other packages depend on it

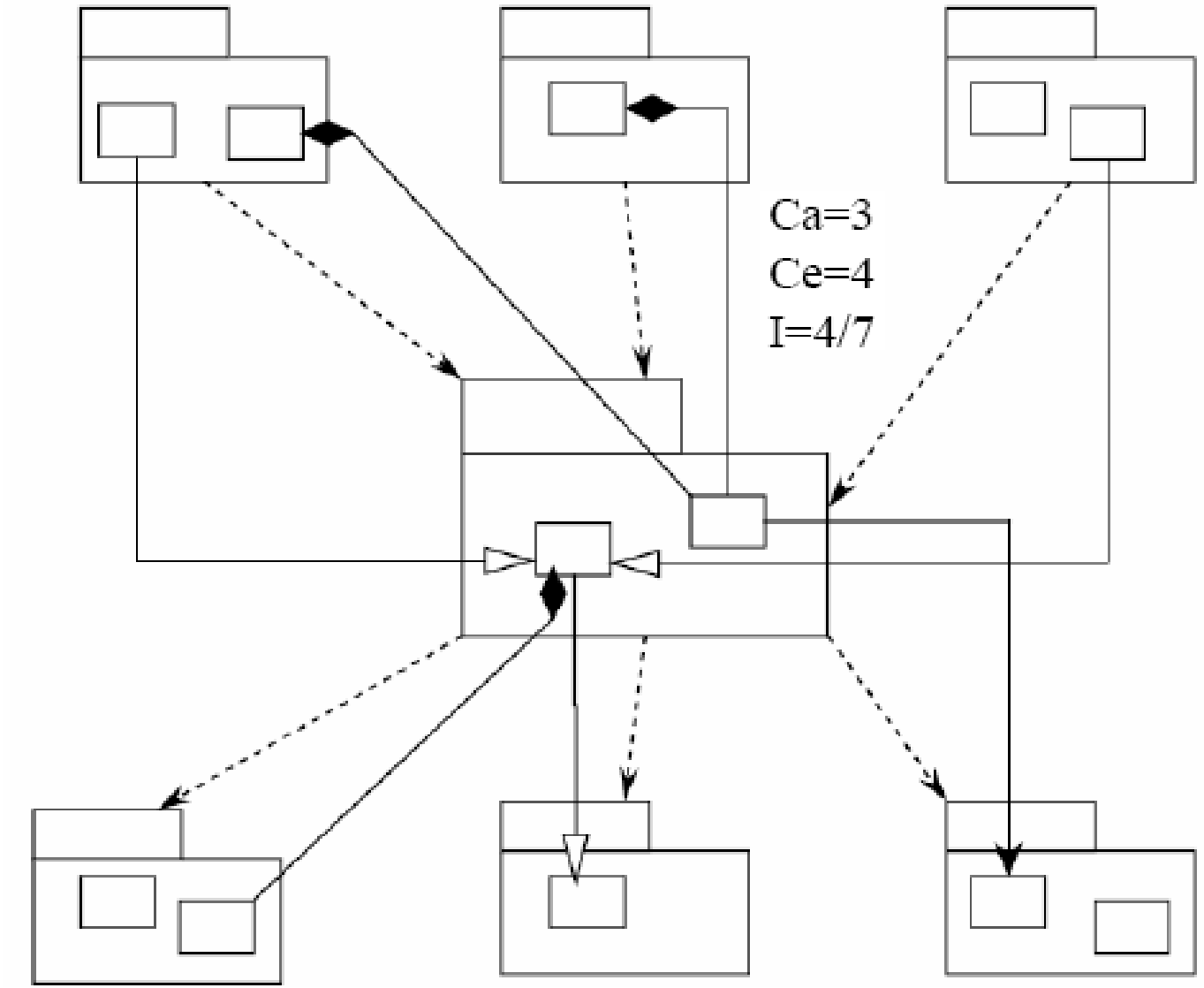
$I = 0$



A is unstable, it depends on many other packages

$I = 1$

An example

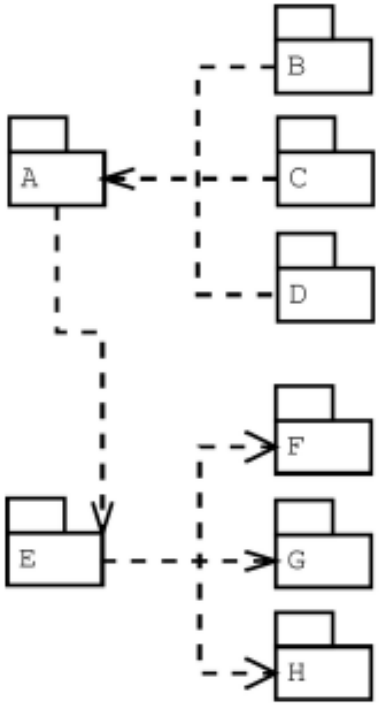


An example

Bad

A is responsible for B, C and D. It depends on E, → irresponsible

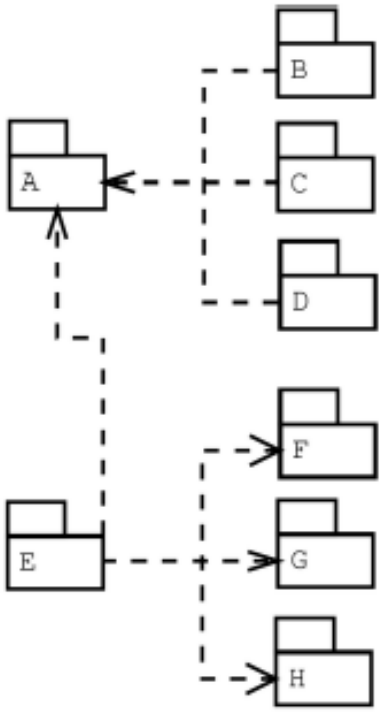
E depends on F, G and E. A depends on it. E is responsible and irresponsible.



Good

A is responsible for B, C, D and E. It will be hard to change

E depends on A, F, G and H. It is irresponsible and will be easy to modify.



Stability and Dependency

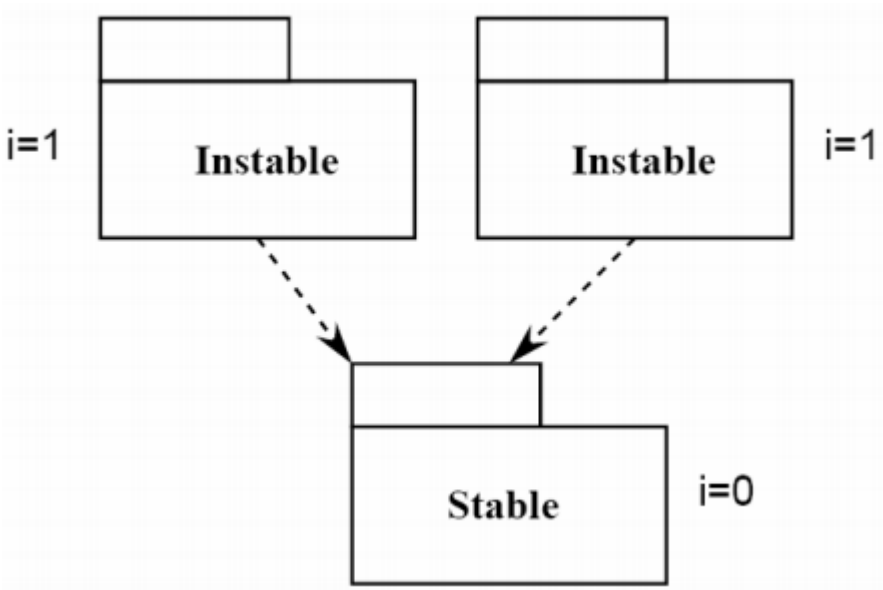
• When the I metric is 1 it means that no other package depends upon this package; and this package does depend upon other packages. (I=1：该包依赖于其他包，但不被其他包所依赖，因此它是最不稳定的)

• When the I metric is zero it means that the package is depended upon by other packages, but does not itself depend upon any other packages. (I=0：该包不依赖于其他包，但被其他包所依赖，因此它是最稳定的)

...并不是所有的包都是要稳定的

• If all the packages in a system were maximally stable, the system would be unchangeable. This is not a desirable situation.（如果所有的包都是很稳定的，那么系统将无法发生变化，这不是我们想要的）

• Indeed, we want to design our package structure so that some packages are instable and some are stable.（应有稳定的和不稳定的包）



(SAP) The Stable Abstraction Principle
稳定抽象原则

- This principle sets up a relationship between stability and abstractness. (在稳定性与抽象度之间建立关联)
- A package should be as abstract as it is stable.（一个包是稳定的，那么它就应该尽可能抽象）
- A completely stable package should consist of nothing but abstract classes（一个完全稳定的包中只应包含抽象类）
- An instable package should be concrete since it its instability allows the concrete code within it to be easily changed.（不稳定的包应是具体的，以便于容易的进行修改）

稳定抽象原则（SAP）和稳定依赖原则（SDP）

The SAP and the SDP combined amount to the Dependency Inversion Principle for Packages. (SAP和SDP共同构成了包之间的“依赖倒置原则DIP”)
SDP says that dependencies should run in the direction of stability, and the SAP says that stability implies abstraction. (SDP: 依赖应指向稳定的方向，SAP: 稳定性隐含着抽象)
Thus, dependencies run in the direction of abstraction (因此，依赖应指向抽象的方向)

Measuring Abstraction

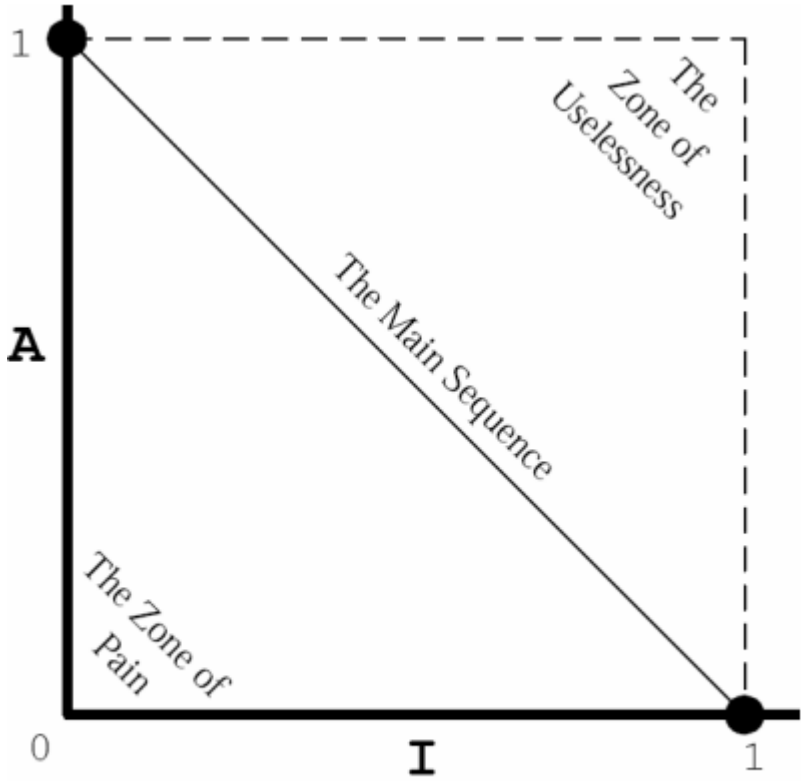
- The A metric is a measure of the abstractness of a package.
- Its value is simply the ratio of abstract classes in a package to the total number of classes in the package.（抽象度可通过计算包中包含的抽象类在所有类中所占的比例而得到）
- An abstract class is simply a class that has at least one pure virtual function（抽象类至少具有一个纯虚函数）.

A = (# of abstract classes) / (# of classes)

- $0 \leq A \leq 1$
- 0 - no abstract classes;
- 1 - all abstract classes

Stability vs. Abstraction

- A=1, I=0:
 maximally stable and abstract
- A=0, I=1:
maximally instable an concrete
- A=0, I=0:
highly stable and concrete
- A=1, I=1:
maximally abstract and instable

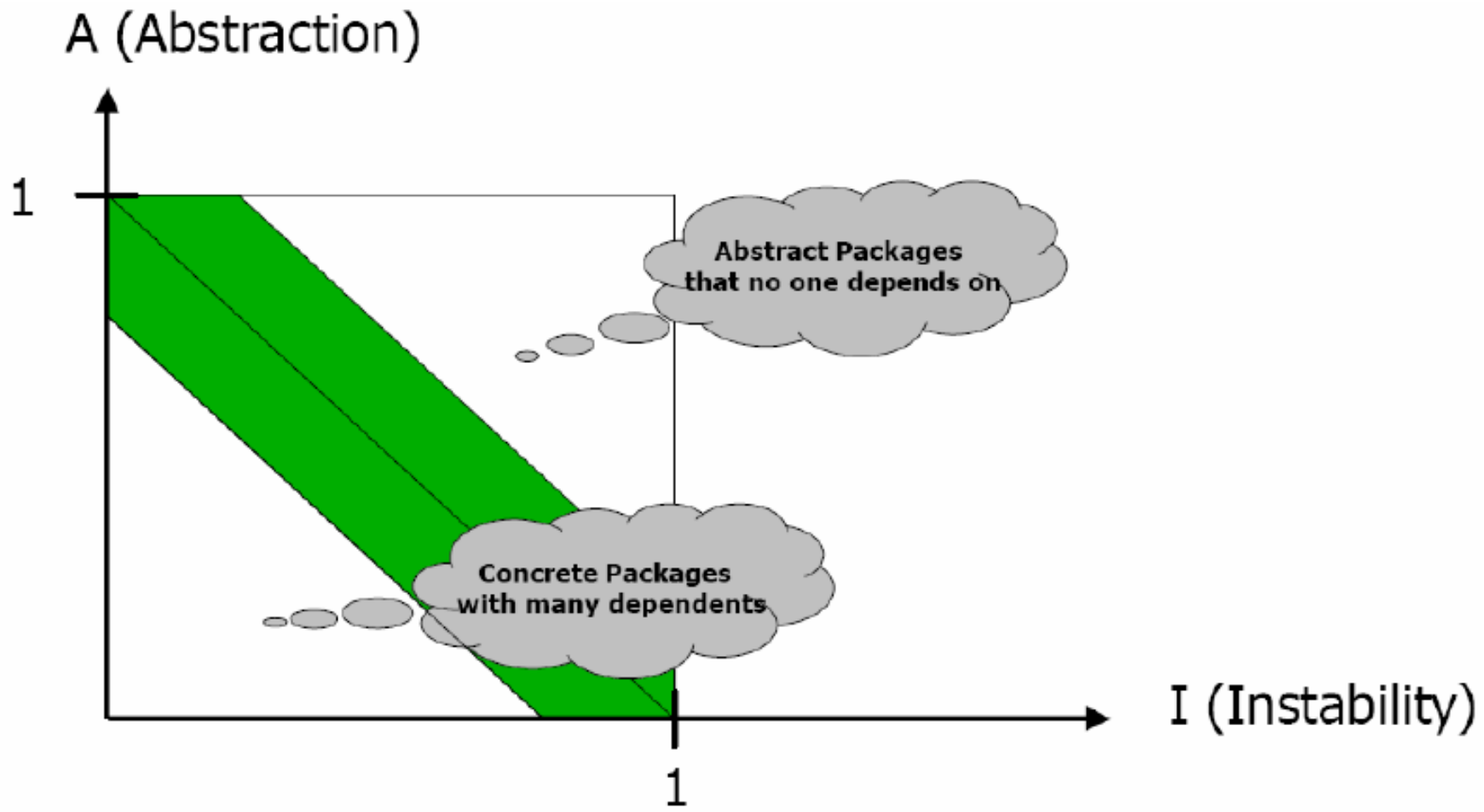


A-I graph

Main sequence of A-I graph

- Draw a line from (0,1) to (1,0).
- This line represents packages whose abstractness is "balanced" with stability. (稳定性与抽象度之间的平衡线)
- A package that sits on the main sequence is not "too abstract" for its stability, nor is "too instable" for its abstractness.
- It has the "right" number of concrete and abstract classes in proportion to its efferent and afferent dependencies.
- Clearly, the most desirable positions for a package to hold are at one of the two endpoints of the main sequence.

Main sequence of A-I graph



Distance from the Main Sequence

• It is desirable for packages to be on or close to the main sequence (理想情况是：应尽可能靠近主序列)

• Create a metric which measures how far away a package is from this ideal. (需要度量一个包理想情况有多远的距离)

$$D = \frac{|A + I - 1|}{\sqrt{2}}$$

• The perpendicular distance of a package from the main sequence.

• Ranges from [0~0.707].

Software Architecture Measurement and Analysis
软件架构度量与分析

汇报人：莫然

中国地质大学

2018-11-01

Architecture flaws（架构缺陷）

- Unstable Interface
- Modularity Violation
- Unhealthy Inheritance
- Clique
- Package Cycle
- Crossing

[1] *Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells.* WICSA 2016

[2] *A Case Study in Locating the Architectural Roots of Technical Debt.* ICSE 2015

Architecture flaws（架构缺陷）

1.Unstable Interface

不稳定接口

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	**StreamSession_java**	**(1)**	x,10	x,8	4	2	4	2	3	7	2	3	2	2	2	3	8	2	2
2	ConnectionHandler_java	**x,10**	(2)	2	x,3				3	5							4		
3	StreamTransferTask_java	**x,8**	2	(3)	3		x,4				2	3	2			2	5		
4	StreamMessage_java	**x,4**	3	3	(4)	x	x,2		2	2		3	2				2		
5	ReceivedMessage_java	**x,2**			x	(5)													
6	OutgoingFileMessage_java	**x,4**		4	x,2		(6)				2	2				2	3		
7	SessionInfo_java	**x,2**						(7)		2							2	2	2
8	StreamCoordinator_java	**x,3**	3		2				(8)	4				2			3		
9	StreamPlan_java	**x,7**	5		2			2	x,4	(9)							5	2	2
10	LegacySSTableTest_java	**x,2**		2			2			x	(10)						2		
11	StreamReader_java	**x,3**		3	3		2					(11)	4			6			
12	CompressedStreamReader_java	**x,2**		2	2							x,4	(12)			2			
13	StreamEvent_java	**x,2**						x	2					(13)					
14	StreamStateStoreTest_java	**x,2**												x	(14)				
15	StreamReceiveTask_java	**x,3**		2			2					6	2			(15)			
16	StreamTransferTaskTest_java	**x,8**	4	x,5	2		3	2	3	5	2						(16)	2	2
17	SessionInfoCompositeData_java	**x,2**						x,2		2							2	(17)	2
18	SessionInfoTest_java	**x,2**						x,2		2							2	2	(18)

Architecture flaws（架构缺陷）

Modeling architecture flaws among files

2.Modularity Violation

模块违约

Structurally independent files frequently change together

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	DropIndexStatement_java	(1)	14	13	12	10	8	7	6	6	5	5	5	5	5	5
2	CreateIndexStatement_java	14	(2)	16	11	10	8	7	6	6	12	7		4	5	6
3	AlterTableStatement_java	13	16	(3)	12	9	7	8	6	6	12	9		6	5	9
4	CreateKeyspaceStatement_java	12	11	12	(4)	12	7	10	5	5	4			3	5	6
5	DropKeyspaceStatement_java	10	10	9	12	(5)	4	7	5	5				3	5	5
6	CassandraServer_java	8	8	7	7	4	(6)					24	43			
7	AlterKeyspaceStatement_java	7	7	8	10	7		(7)	5				3	3	5	5
8	DropTriggerStatement_java	6	6	6	5	5		5	(6)	12				4	5	5
9	CreateTriggerStatement_java	6	6	6	5	5		5	12	(9)				4	5	5
10	SelectStatement_java	5	12	12	4	3	21				(10)	48	8			3
11	ModificationStatement_java	5	7	9	4	3	1				48	(11)				3
12	StorageService_java	5		3	3	3	3	3			8	0	(12)			
13	AlterTypeStatement_java	5	4	6	3	3		3	4	4				(13)	3	4
14	DropTableStatement_java	5	5	5	5	5		5	5	5				3	(14)	5
15	CreateTableStatement_java	5	6	9	6	5		5	5	5	3	3		4	5	(15)

High Co-changes

Architecture flaws（架构缺陷）

Modeling architecture flaws among files

3.Unhealthy Inheritance

继承问题

parent class depends on child class

		1	2	3	4	5	6	7	8	9
1	ProcessorDefinition_java	(1)	dp,8			,2	,2		,6	,3
2	AggregateDefinition_java	ex,8	(2)							
3	JmsEndpoint_java			(3)	dp,10		,2	,2	,2	
4	JmsQueueEndpoint_java			ex,10	(4)					
5	ManagedPerformanceCounter_java	,2				(5)	,3	,4	,5	
6	ManagedProcessor_java	dp,2		,2		ex,3	(6)	,7	,9	
7	ManagedCamelContext_java			,2		ex,4	,7	(7)	,21	,2
8	ManagedRoute_java	,6		,2		ex,5	,9	,21	(8)	
9	DefaultManagementObjectStrategy_java	,3				dp	dp	dp,2	dp	(9)

ex: extend
dp: depend

client class depends on both parent and child classes

Architecture flaws（架构缺陷）

Modeling architecture flaws among files

4.Clique

强耦合文件集

files form a **strongly connected** component

		1	2	3	4	5	6	7	8	9	10	11	12
1	ActivityRules_java	(1)	,5	dp,3	,3	,3	,4	,2	,2	,2	dp,4	dp,3	,3
2	ProcessInstance_java	dp,5	(2)	,3	dp,11	,4	,4	,6	,6	,2	,3	,8	,4
3	ProcessRules_java	dp,3	,3	(3)	,2		,3	,2	,2	,2	,3	,2	,2
4	ActivityState_java	dp,3	dp,11	,2	(4)	,3	,3	,7	,6	dp,2	,3	,7	,2
5	JpaBamProcessorSupport_java	dp,3	,4	dep	,3	(5)	,4	,2	,2	,2	,3	,6	,7
6	JpaBamProcessor_java	dp,4	dp,4	dp,3	dp,3	dp,4	(6)	,3	,3	dp,2	,4	,3	,5
7	TimeExpression_java						,3	(7)	dp,8	,2	dp,3	,7	,2
8	ActivityBuilder_java						,3	,8	(8)	,2	,3	dp,10	,2
9	ProcessContext_java						,2	,2	,2	(9)	,2	,3	,2
10	TemporalRule_java						,4	dp,3	dp,3	,2	(10)	,3	,4
11	ProcessBuilder_java	,3	dp,8	dp,2	,7	,6	dp,3	,7	dp,10	,3	,3	(11)	dp,5
12	ActivityMonitorEngine_java	,3	,4	dp,2	dp,2	,7	,5	,2	,2	,2	,4	,5	(12)

Cyclic Dependencies

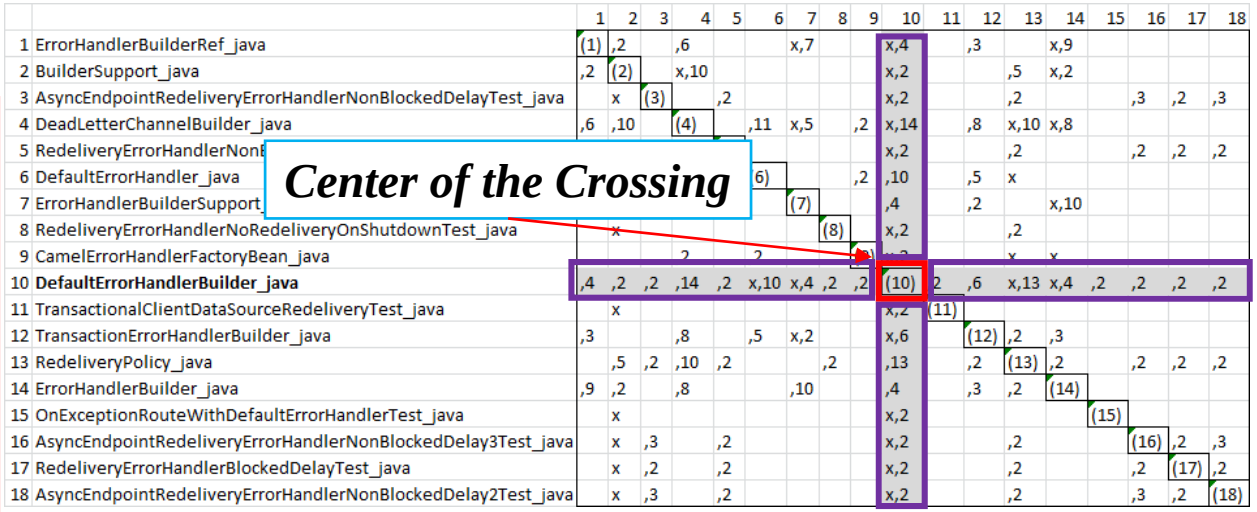
Architecture flaws（架构缺陷）

Modeling architecture flaws among files

5.Crossing

中心文件问题

A file with both **high fan-in** and **high fan-out** changed frequently with its dependents and dependees.



		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	ErrorHandlerBuilderRef_java	(1)	,2		,6			x,7			x,4		,3		x,9				
2	BuilderSupport_java	,2	(2)		x,10						x,2			,5	x,2				
3	AsyncEndpointRedeliveryErrorHandlerNonBlockedDelayTest_java		x	(3)		,2					x,2			,2			,3	,2	,3
4	DeadLetterChannelBuilder_java	,6	,10		(4)		,11	x,5		,2	x,14		,8	x,10	x,8				
5	RedeliveryErrorHandlerNon										x,2			,2			,2	,2	,2
6	DefaultErrorHandler_java						6)			,2	,10		,5	x					
7	ErrorHandlerBuilderSupport							(7)			,4		,2		x,10				
8	RedeliveryErrorHandlerNoRedeliveryOnShutdownTest_java		x						(8)		x,2			,2					
9	CamelErrorHandlerFactoryBean_java				2		2							x	x				
10	**DefaultErrorHandlerBuilder_java**	,4	,2	,2	,14	,2	x,10	x,4	,2	,2	(10)	2	,6	x,13	x,4	,2	,2	,2	,2
11	TransactionalClientDataSourceRedeliveryTest_java		x								x,2	(11)							
12	TransactionErrorHandlerBuilder_java	,3			,8		,5	x,2			x,6		(12)	,2	,3				
13	RedeliveryPolicy_java		,5	,2	,10	,2			,2		,13		,2	(13)	,2		,2	,2	,2
14	ErrorHandlerBuilder_java	,9	,2		,8			,10			,4		,3	,2	(14)				
15	OnExceptionRouteWithDefaultErrorHandlerTest_java		x								x,2					(15)			
16	AsyncEndpointRedeliveryErrorHandlerNonBlockedDelay3Test_java		x	,3		,2					x,2			,2			(16)	,2	,3
17	RedeliveryErrorHandlerBlockedDelayTest_java		x	,2		,2					x,2			,2			,2	(17)	,2
18	AsyncEndpointRedeliveryErrorHandlerNonBlockedDelay2Test_java		x	,3		,2					x,2			,2			,3	,2	(18)

Architecture flaws（架构缺陷）

Modeling architecture flaws among files

6.Package Cycle

包循环

Two packages depend on each other

作 业

- 1）属性驱动设计方法ADD、基于模式的设计方法中哪些步骤会涉及构件级（模块化）设计方法？
- 2）消除循环依赖通常有哪些方法？请举例说明。
- 3）回顾您曾经参与的或熟悉的一个软件项目，谈谈软件设计尤其是体系结构设计方面存在什么问题？针对软件的一部分或全部，选择属性驱动设计方法、基于模式的设计方法、模块化设计方法中的一种或多种的结合进行软件体系结构设计练习？