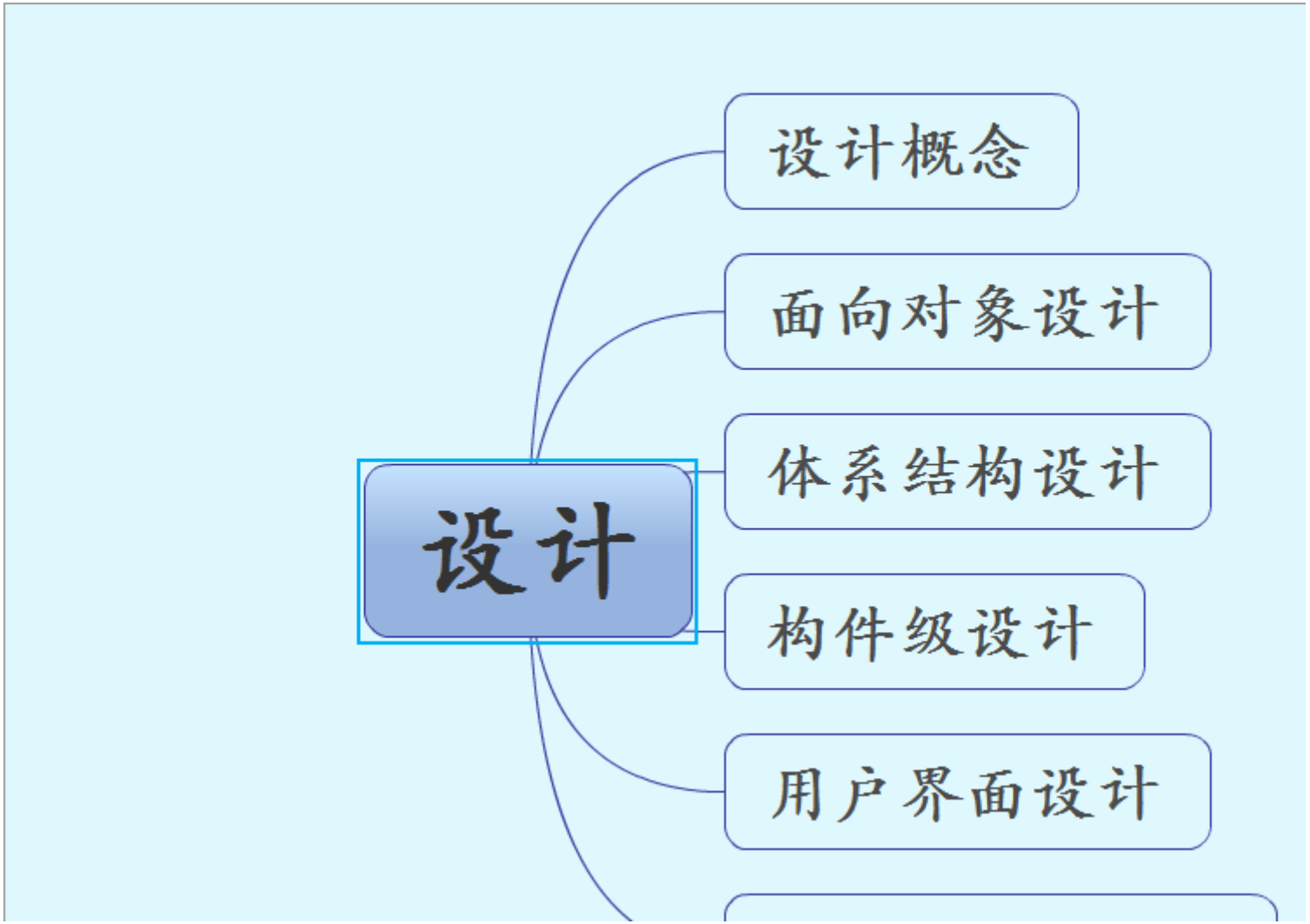
outline



- 构件定义
- 构件设计过程
- 软件重用
- 基本设计原则
- 构件详细设计
- 基于构件的软件工程

构件定义

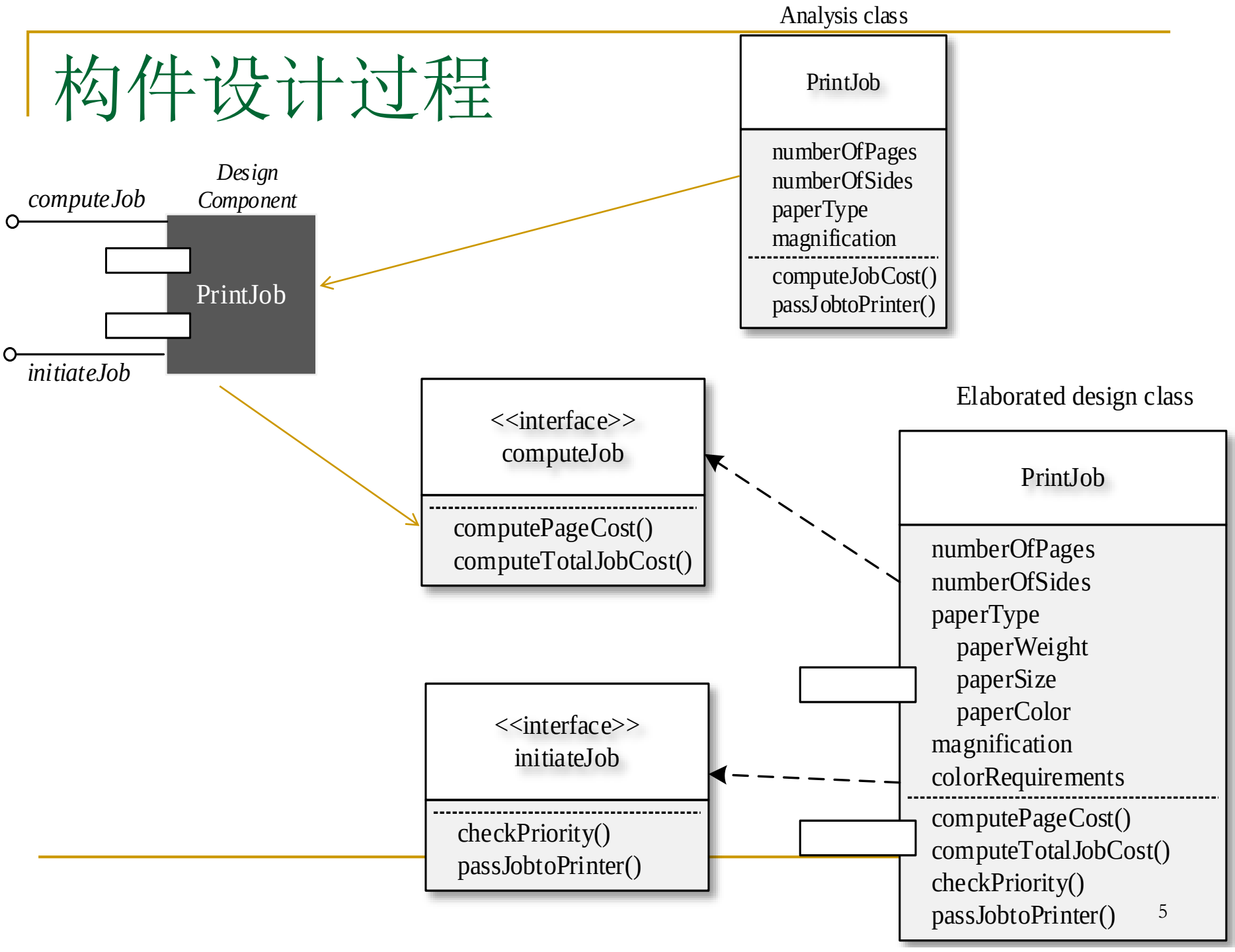
“a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.”

OMG Unified Modeling Language Specification

构件设计过程

- 在面向对象软件工程中，一个构件由一系列相互协作的类组成。
- 构件设计过程将分析模型和体系结构模型转变为指导构建活动的设计模型。
- 构件设计过程是一个迭代、不断精化的过程
  - 第一次迭代：构件中的每个类都包括和实现相关的所有属性和方法
  - 第二次迭代：为每个属性定义合适的数据结构；为每个方法设计算法（活动图、过程设计方法）
  - 定义每个类和其它类进行通信的所有接口

构件设计过程


Analysis class
PrintJob
numberOfPages
numberOfSides
paperType
magnification
computeJobCost()
passJobtoPrinter()
Design Component
computeJob
PrintJob
initiateJob
<<interface>>
computeJob
computePageCost()
computeTotalJobCost()
Elaborated design class
PrintJob
numberOfPages
numberOfSides
paperType
paperWeight
paperSize
paperColor
magnification
colorRequirements
computePageCost()
computeTotalJobCost()
checkPriority()
passJobtoPrinter()
<<interface>>
initiateJob
checkPriority()
passJobtoPrinter()

软件重用

1. 重用

- 重用也叫再用或复用，是指同一事物不作修改或稍加改动就多次重复使用。
- 广义地说，软件重用可分为以下3个层次：
  - 知识重用(软件工程知识的重用)。
  - 方法和标准的重用(面向对象方法或国家制定的软件开发规范的重用)。
  - 软件成分的重用。

软件重用

2. 软件成分的重用级别

- 代码重用：调用库中的模块
  - 源代码剪贴
  - 源代码包含　#include
  - 继承
- 设计结果重用：重用某个软件系统的设计模型。这个级别的重用有助于把一个应用系统移植到完全不同的软硬件平台上。
- 分析结果重用：重用某个系统的分析模型。这种重用特别适用于用户需求未改变，但系统体系结构发生了根本变化的场合。

软件重用

3. 典型的可重用软件成分

- 项目计划：软件质量保证计划。
- 成本估计：不同项目中类似功能的成本估算。
- 体系结构：事务类处理体系结构。
- 需求模型和规格说明：对象模型，数据流图。
- 设计：体系结构、数据、接口和过程设计。
- 源代码：兼容的程序构件。
- 用户文档和技术文档：部分重用。
- 用户界面：GUI 可占应用程序的60%代码量。
- 数据： 记录结构，文件和完整的数据库。
- 测试用例：与重用设计或代码相关的用例。

类构件

- 面向对象技术中的“类”，是比较理想的可重用软构件，称之为类构件。

1. 可重用软构件应具备的特点
   - 模块独立性强：具有单一、完整的功能，且经过反复测试被确认是正确的。
   - 具有高度可塑性：必须提供为适应特定需求而扩充或修改已有构件的机制，且使用起来非常简单方便。
   - 接口清晰、简明、可靠：详尽的文档说明。

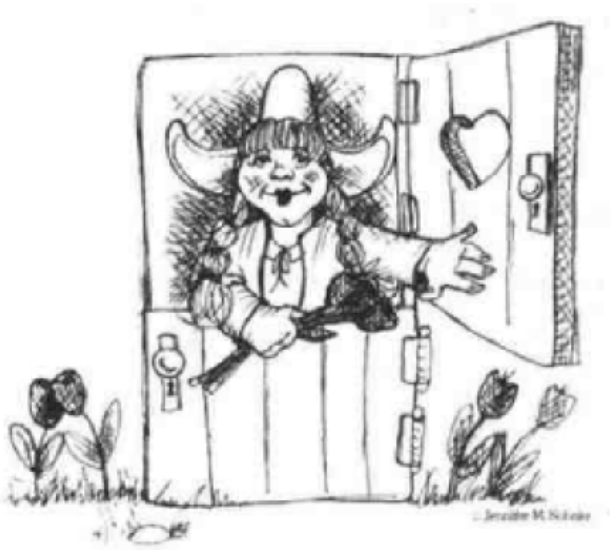
2. 类构件的重用方式

- 实例重用（封装性）
  - 使用适当的构造函数，按照需要创建类的实例。
  - 用几个简单的对象作为类的成员创建出一个更复杂的类。
- 继承重用（继承性）
  - 继承重用提供了一种安全地修改已有类构件，以便在当前系统中重用的手段。
  - 设计一个合理的、具有一定深度的类构件继承层次结构。
- 多态重用（多态性）
  - 使对象的对外接口更加一般化，降低消息连接的复杂程度。
  - 系统运行时，根据接收消息的对象类型，由多态性机制启动正确的方法。

基本设计原则

- 开闭原则（OCP）
- 针对接口编程原则
- 类的单一职责原则（SRP）
- Liskov替换原则（LSP）
- 接口隔离原则（ISP）
- 迪米特法则（LoD）
- 合成/聚合复用原则 CARP

开闭原则



A component should be open for extension but closed for modification.

Robert C. Martin

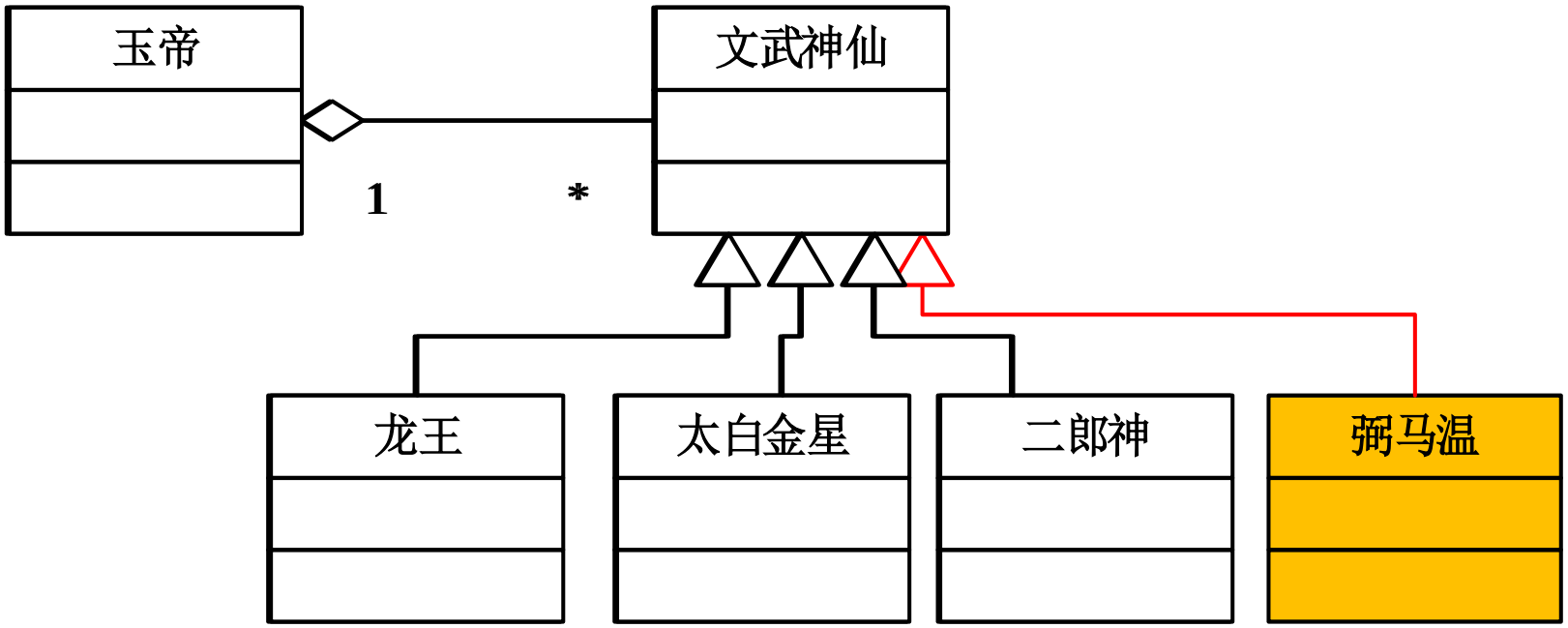
Design Principles and Design Patterns

开闭原则

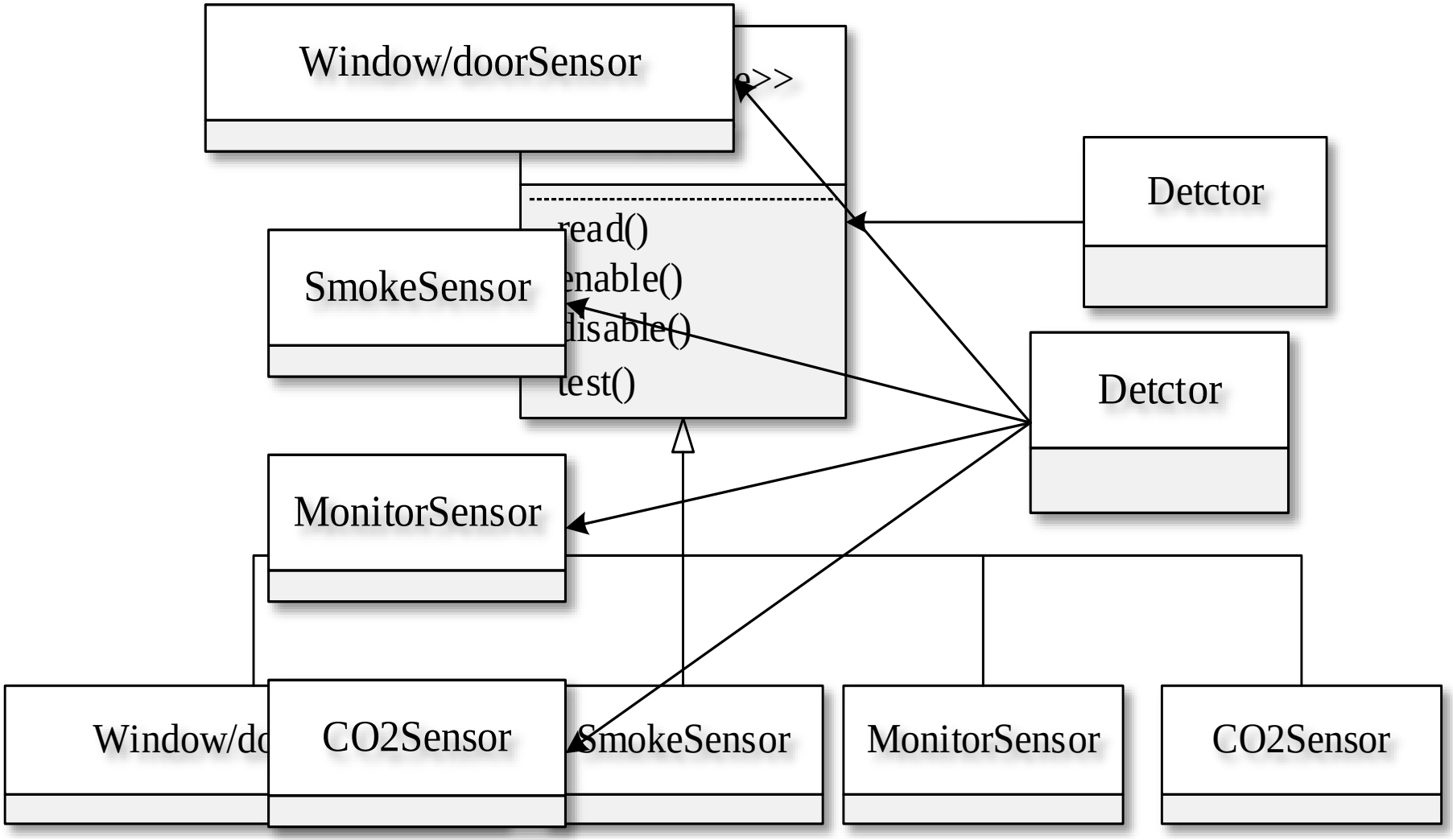
- 在设计一个构件时，应当让这个构件在不被修改（代码层）的前提下被扩展，即在不修改源代码的情况下改变这个模块的行为。
- 开闭原则的关键是抽象

例1

- 玉帝招安美猴王：不破坏天规是闭，收仙有道是开。招安之法便是天庭的开闭原则。通过给猴子一个职位，便可以满足了新的变化的需求，而不必破坏天庭已有的秩序



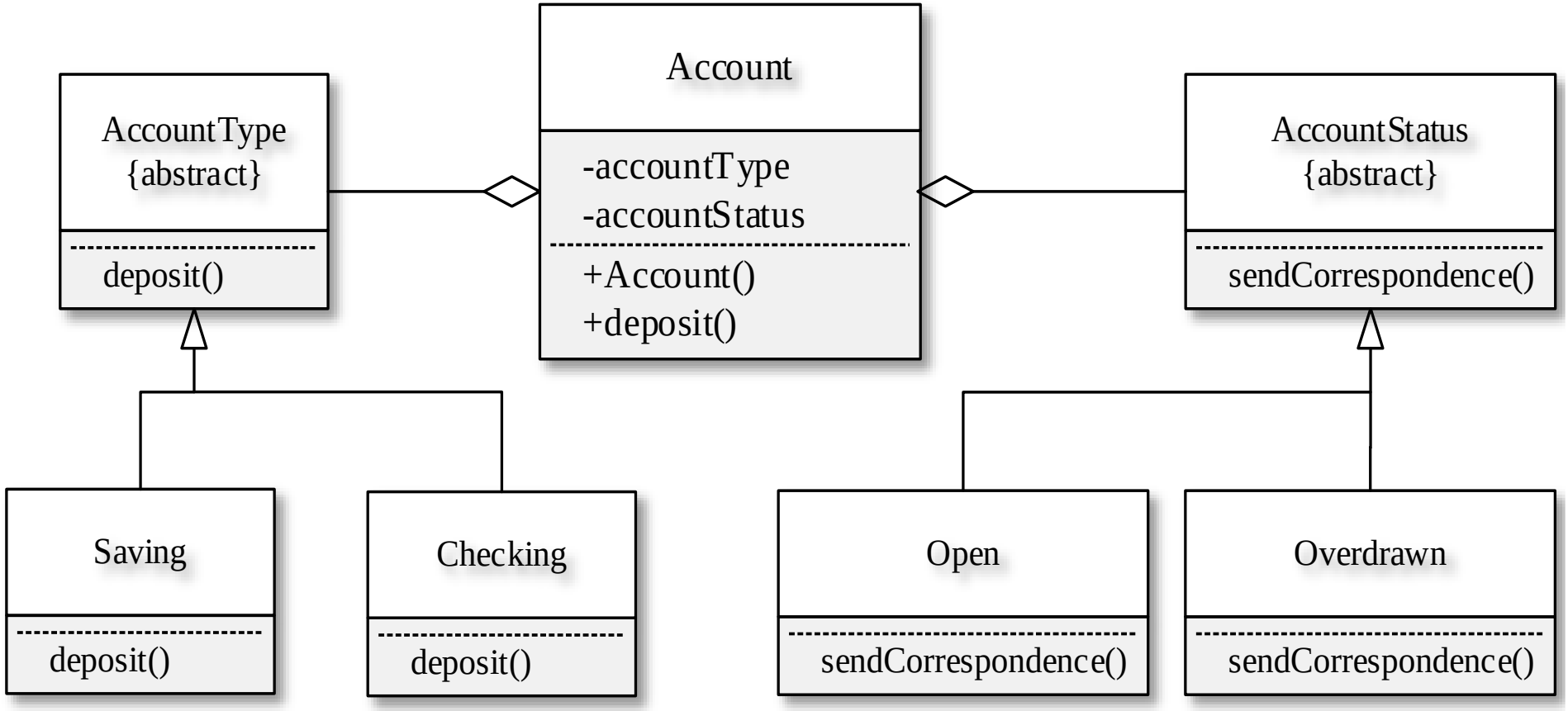
例2


Window/doorSensor
e>>
read()
enable()
disable()
test()
SmokeSensor
Detctor
Detctor
MonitorSensor
Window/dc
CO2Sensor
SmokeSensor
MonitorSensor
CO2Sensor

针对接口编程原则

- 针对抽象编程
  - 接口是高层的抽象
  - 接口可以隐藏实现的细节
  - 接口可以清晰指出对象的职责
- 抽象类是用来继承的,具体类不是用来继承的
- 所有的具体类不应该有子类
- 松散耦合
- 增加重用的可能性

例：账户


Account
-accountType
-accountStatus
+Account()
+deposit()
AccountType
{abstract}
deposit()
AccountStatus
{abstract}
sendCorrespondence()
Saving
deposit()
Checking
deposit()
Open
sendCorrespondence()
Overdrawn
sendCorrespondence()

例：账户—代码

```
public class Account
{
    private AccountType accountType;
    private AccountStatus accountStatus;
    public Account(AccountType acctType)
    {
        //write your code here
    }
    public void deposit(float amt)
    {
        //write your code here
    }
}
abstract public class AccountType
{
    public abstract void deposit(float amt);
}
```

```
abstract public class AccountStatus
{
    public abstract void sendCorrespondence();
}

public class Savings extends AccountType
{
    public void deposit(float amt)
    {
        //write your code here
    }
}
 public class Open extends AccountStatus
 {
     public void sendCorrespondence()
     {
         //write your code here
     }
 }
```

例：账户

- Account类并不依赖于具体类，因此当有新的具体类型添加到系统中时，Account类不必改变。
- 系统引进了一个新型的账号：MoneyMarket类型，Account类以及系统里面所有其他的依赖于AccountType抽象类的客户端均不必改变。

```
public class MoneyMarket extends AccountType
{
    public void deposit(float amt)
    {
        //write your code here
    }
}
```

类的单一职责原则

There should never be more than one reason for a class to change.

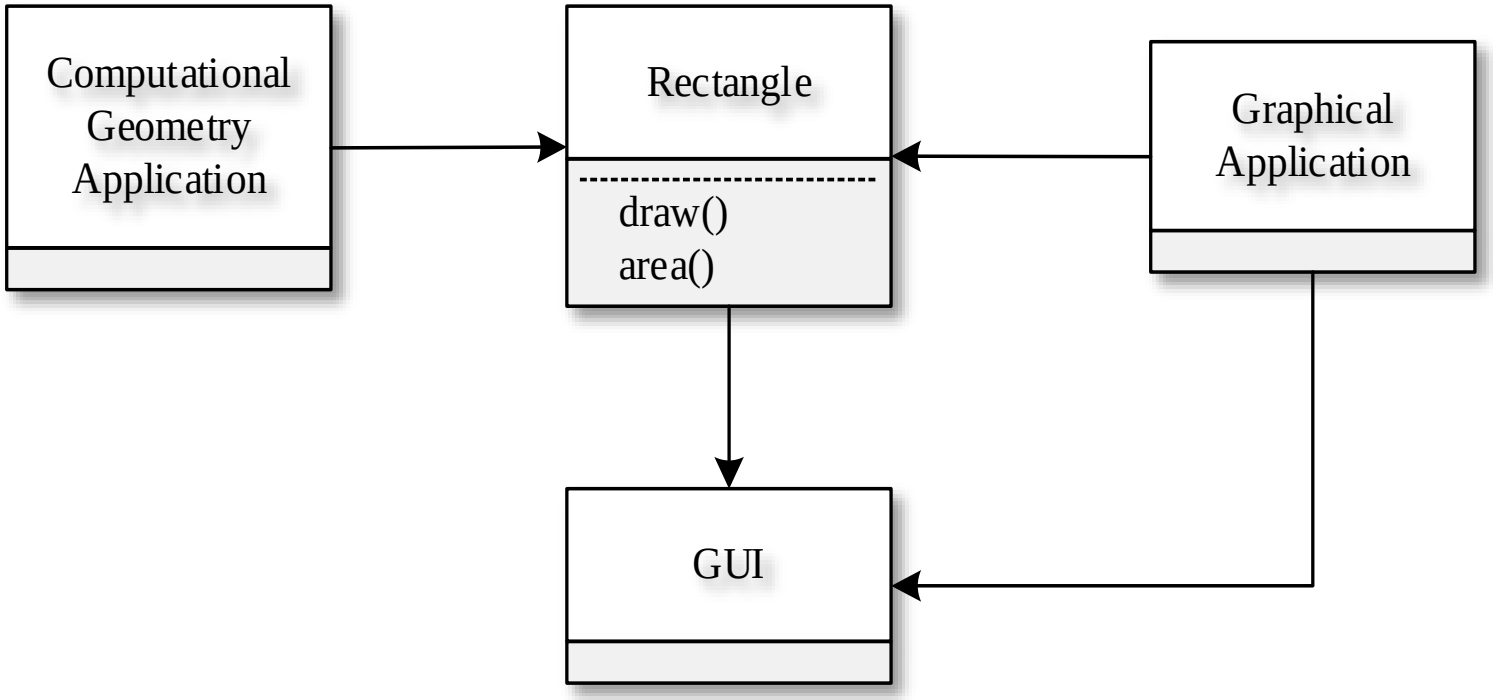
Robert C. Martin

Design Principles and Design Patterns

类的单一职责原则

- 类A负责两个不同的职责：R1，R2。当由于职责R1需求发生改变而需要修改类A时，有可能会导致原本运行正常的职责R2功能发生故障。
- 解决方案：一个类只负责一项职责
  - 分别建立两个类C1、C2，使C1完成职责R1功能，C2完成职责R2功能。这样，修改类C1时，不会使职责R2发生故障风险；修改C2时，也不会使职责R1发生故障风险。
- 例：如一个界面展示类夹杂业务逻辑代码或者数据库连接代码。

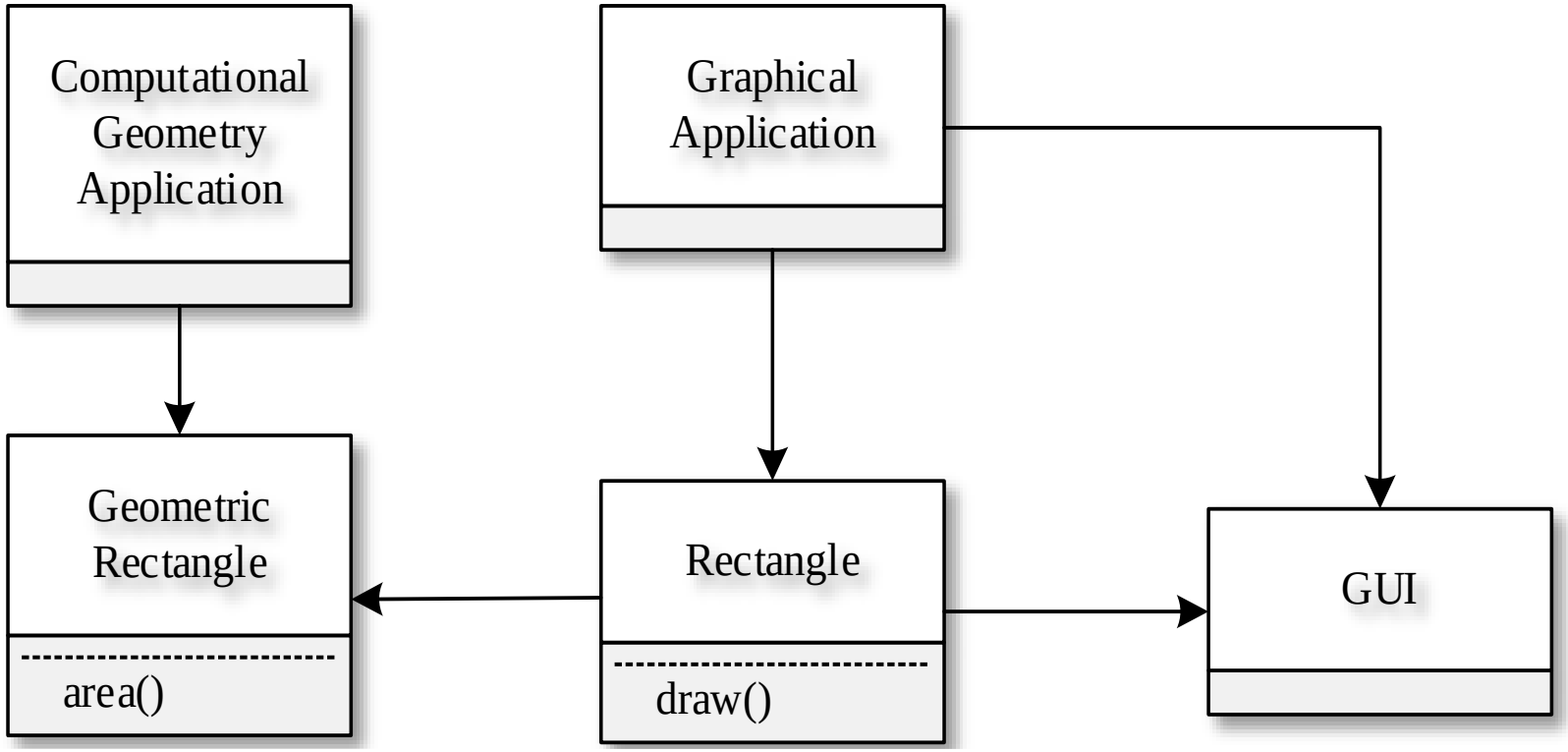
例



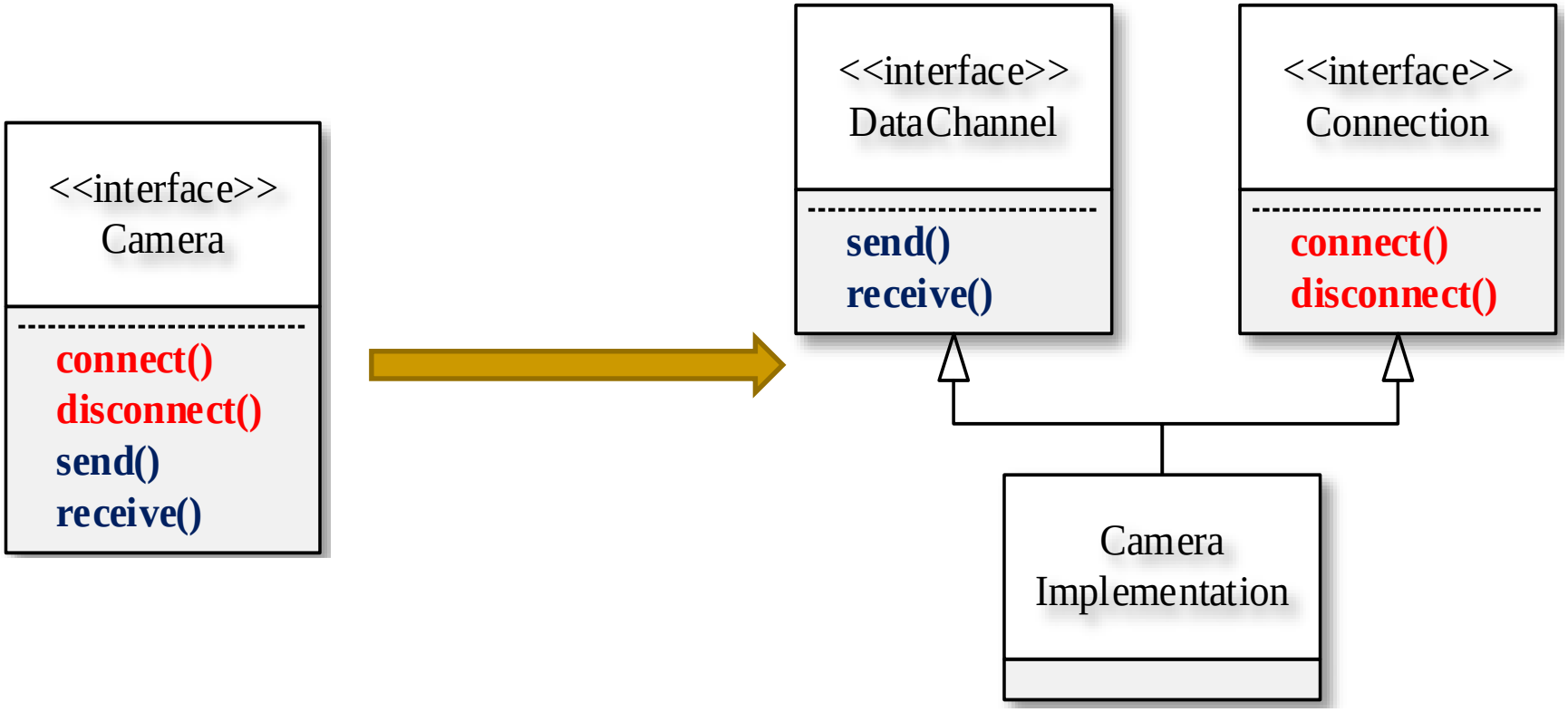
- Rectangle 类包含两个方法：draw方法负责在GUI上绘制矩形，area方法负责计算矩形图形面积。
- 违反了 SRP 原则，带来了如下问题：
  - 必须在计算几何应用中包含对 GUI 库的引用，导致应用程序无谓的消耗了链接时间、编译时间、内存空间和存储空间等。
  - 如果因为某些原因对图形应用的一个更改导致 Rectangle 类也相应做了更改，会影响计算几何应用的功能。

例

- 解决方案：遵循SRP 原则，职责分离
  - 将 Rectangle 中关于几何计算的职责移到了 GeometricRectangle 类中，而 Rectangle 类中仅保留矩形渲染职责。



例


<<interface>>
Camera
connect()
disconnect()
send()
receive()
<<interface>>
DataChannel
send()
receive()
<<interface>>
Connection
connect()
disconnect()
Camera
Implementation

Liskov替换原则

What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

Barbara Liskov

Subclasses should be substitutable for their base classes.

Robert C. Martin

Design Principles and Design Patterns

Liskov替换原则

- 在所有引用基类的地方，都可以用此基类的子类替换，而不影响程序原来的功能。
- 违背Liskov替换原则： RTTI（Run Time Type Identification），根据对象类型选择函数执行。

```
void DrawShape(Shape s)
{
    if (typeid(s) == typeid(Square))
        DrawSquare(s);
    else if (typeid(s) == typeid(Circle))
        DrawCircle(s);
}
```

多态性

```
public class Shapes {
  public static Shape randShape() {
    switch((int)(Math.random() * 3)) {
      default: // To quiet the compiler
      case 0: return new Circle();
      case 1: return new Square();
      case 2: return new Triangle();
    }
  }
  public static void main(String[] args) {
    Shape[] s = new Shape[9];
    // Fill up the array with shapes:
    for(int i = 0; i < s.length; i++)
      s[i] = randShape();
    // Make polymorphic method calls:
    for(int i = 0; i < s.length; i++)
      s[i].draw();
  }
} ///:~
```

根据父类对象＝子类对象，可以将Rectangle/Circle/Triangle类的实例赋值给S[i]

系统根据运行时刻s[i]所属的动态类型来决定执行哪个类的draw操作（动态绑定dynamic binding）

例：长方形和正方形

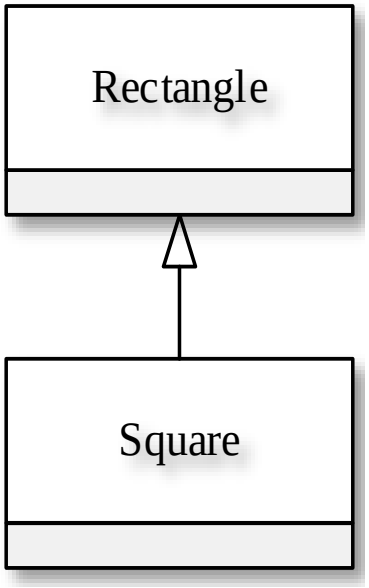
- 假设在一个应用程序中使用了 Rectangle 类
- 某一天用户提出新的需求，要求该应用程序除了能够处理Rectangle之外还要能够处理Square。

```
public class Rectangle
 {
   private double  width;
   private double  height;

   public void SetWidth(double w) {width = w; }
   public void SetHeight(double w) {height = w; }
   public double GetWidth() { return  width; }
   public double GetHeight() { return  height; }
 }
```

例：长方形和正方形

- 一个Square是一个Rectangle




```
public class Square extends Rectangle
{
  public void SetWidth(double w)
  {
    super.SetWidth(w);
    super.SetHeight(w);
  }
  public void SetHeight(double w)
  {
    super.SetWidth(w);
    super.SetHeight(w);
  }
}
```

例：长方形和正方形

```
void TestCase1()
{
  Square s = new Square();
  s.SetWidth(1);
  s.SetHeight(2);
}
```

OK

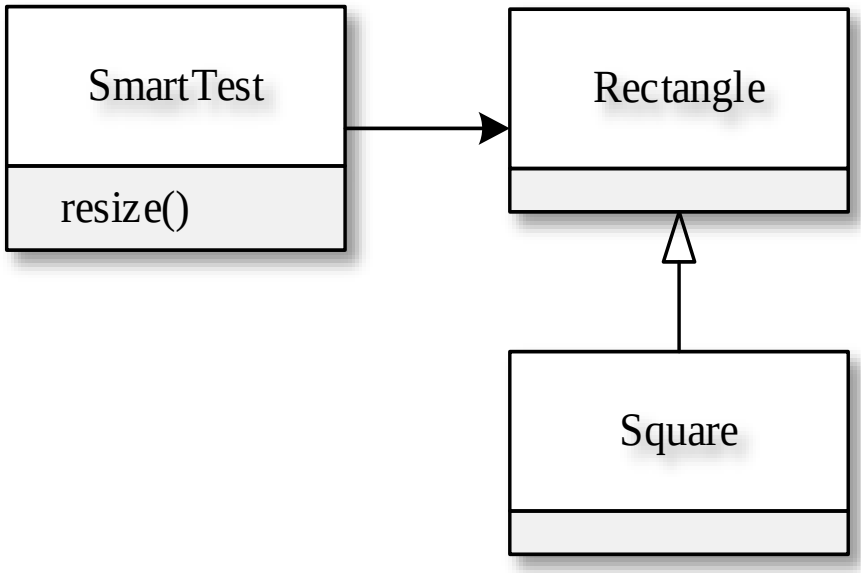
```
void f(Rectangle r)
{
  r.SetWidth(32);
}
```

OK

?

```
void g(Rectangle r)
{
  r.SetWidth(5);
  r.SetHeight(4);
  Assert.AreEqual(r.GetWidth() * r.GetHeight(), 20);
}
```

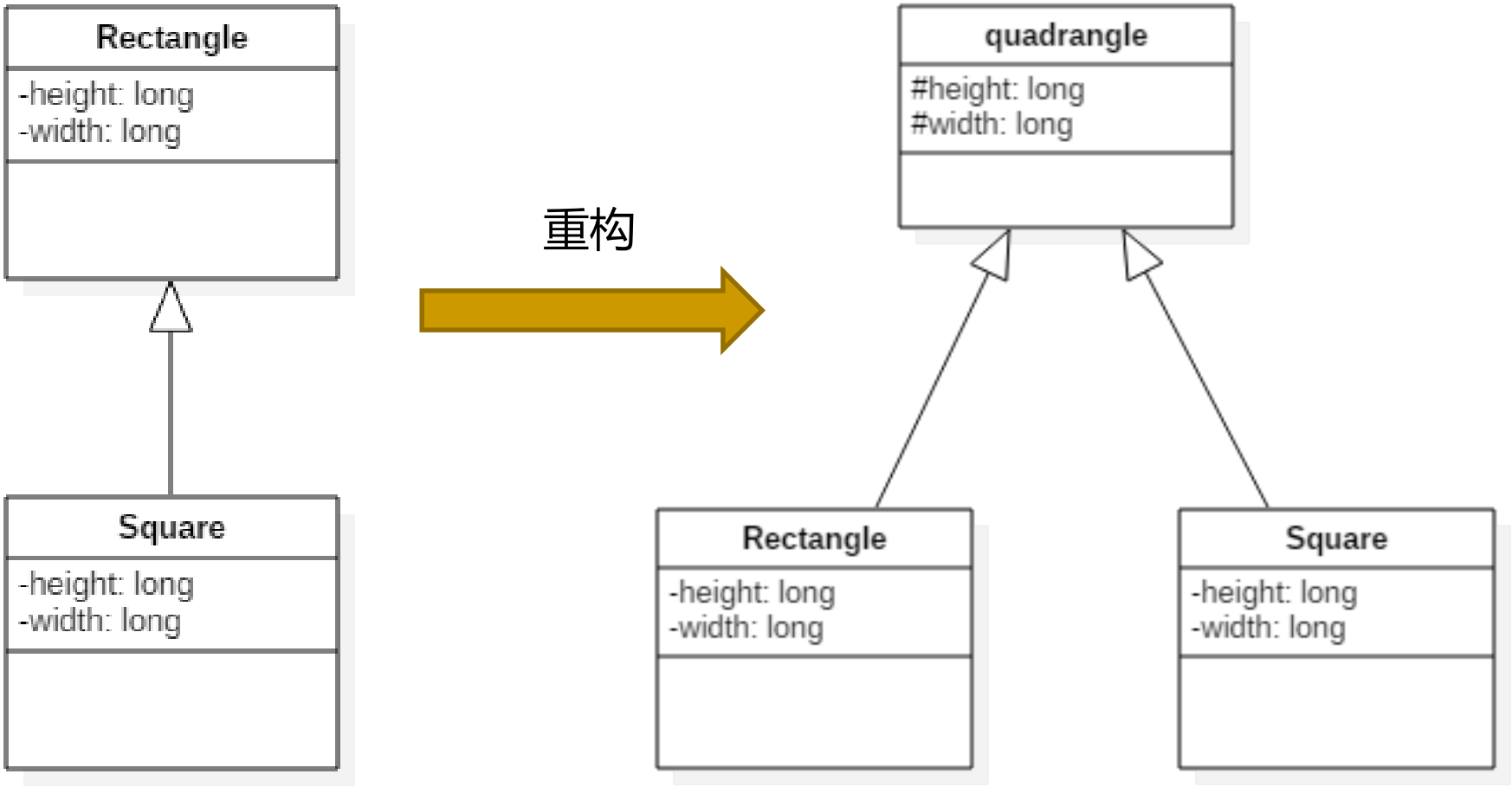
例：长方形和正方形




```
Public class SmartTest
{
  public void resize(rectangle r)
  {
    while (r.getHeight() <= r.getWidth())
      {
        r.setHeight (r.getHeight() + 1);
      }
  }
}
```

- resize()方法传入Rectangle类时，将宽度不断增加，直到超过长度为止
- resize()方法传入Square类时，将正方形的边不断增加，直到溢出。
- Liskov替换原则被破坏了，因此，正方形类不应当是长方形类的子类。

例：长方形和正方形


Rectangle
-height: long
-width: long
Square
-height: long
-width: long
重构
quadrangle
#height: long
#width: long
Rectangle
-height: long
-width: long
Square
-height: long
-width: long

接口隔离原则

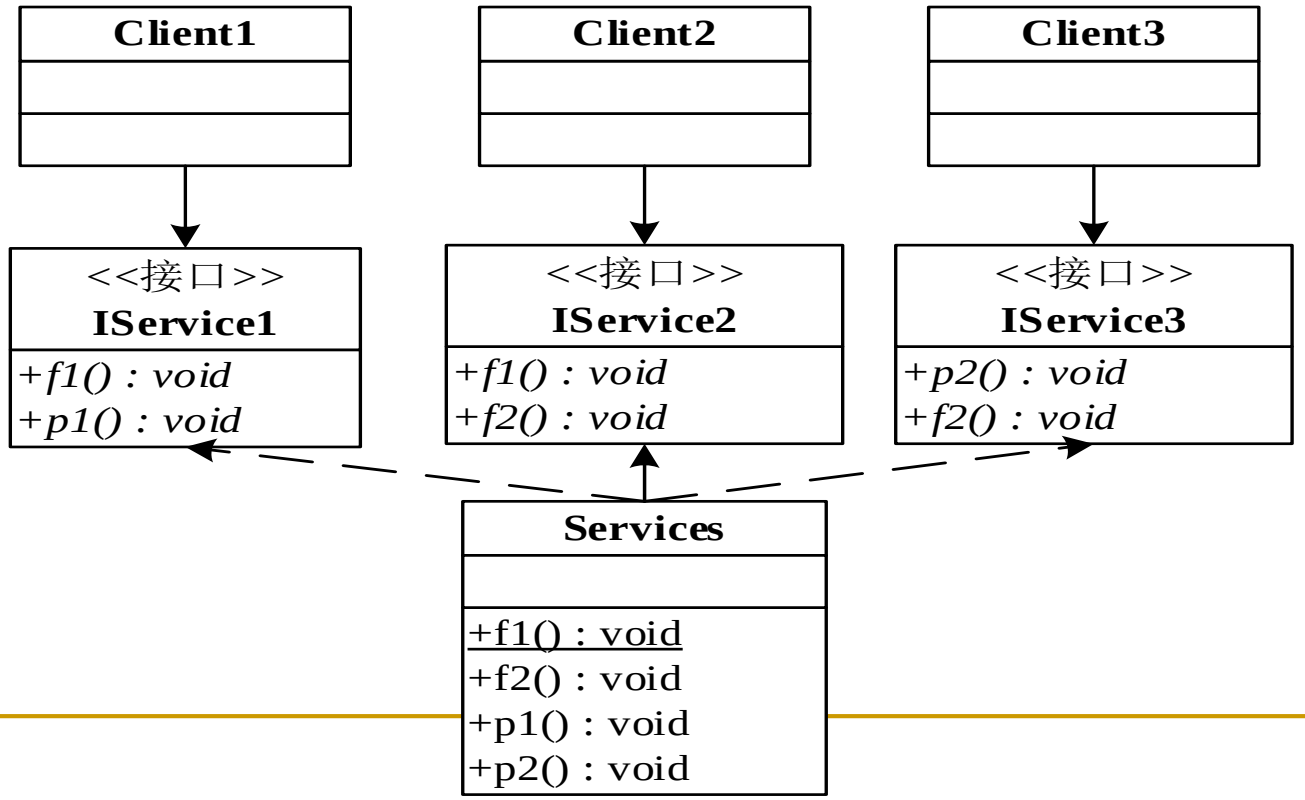
Many client-specific interfaces are better than one general purpose interface.

Robert C. Martin

Design Principles and Design Patterns

接口隔离原则

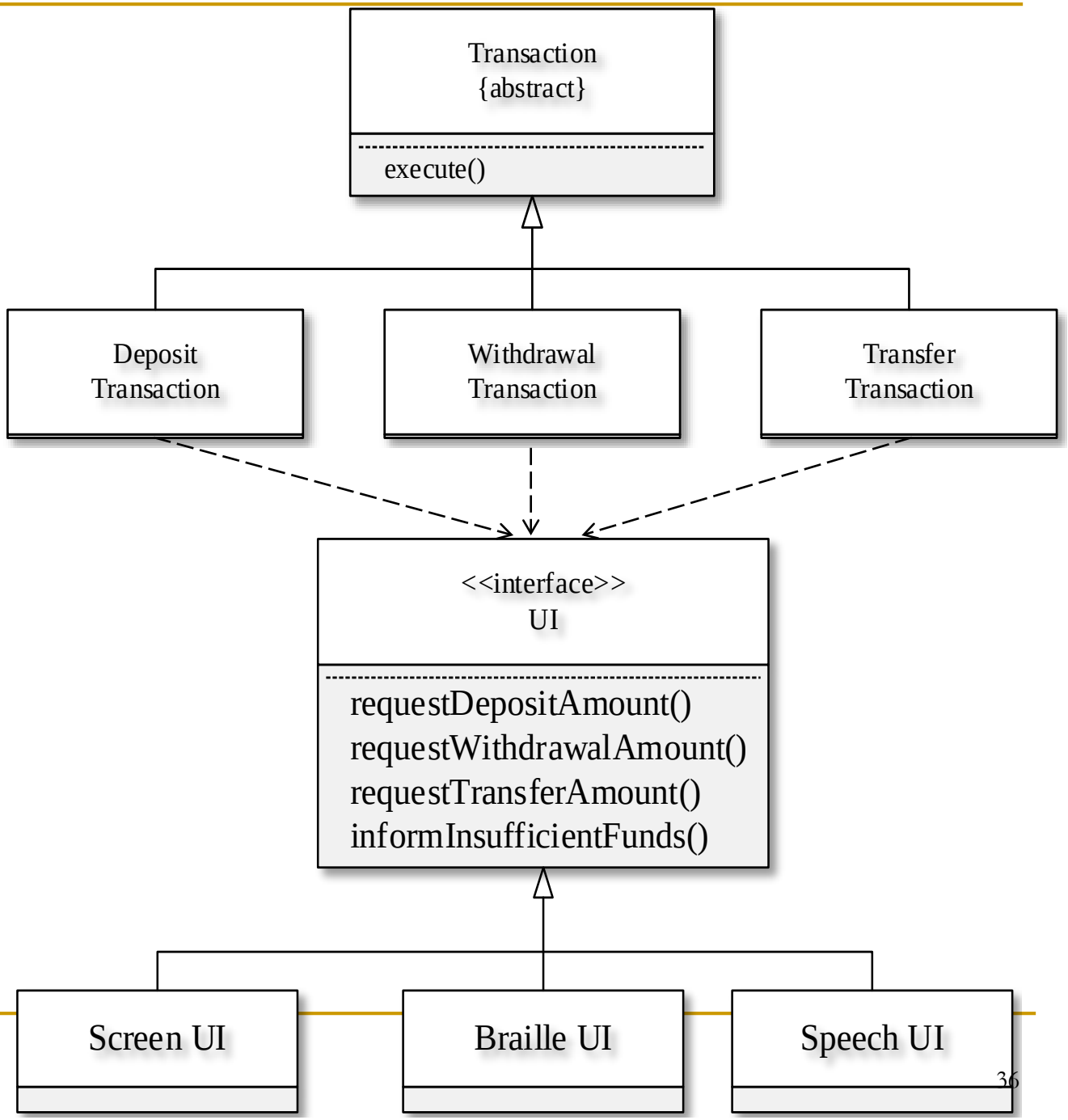
- 如果类的接口定义暴露了过多的行为，则说明这个类的接口定义内聚程度不够好。
- 类的接口可以被分解为多组功能函数的组合，每一组服务于不同的客户类。



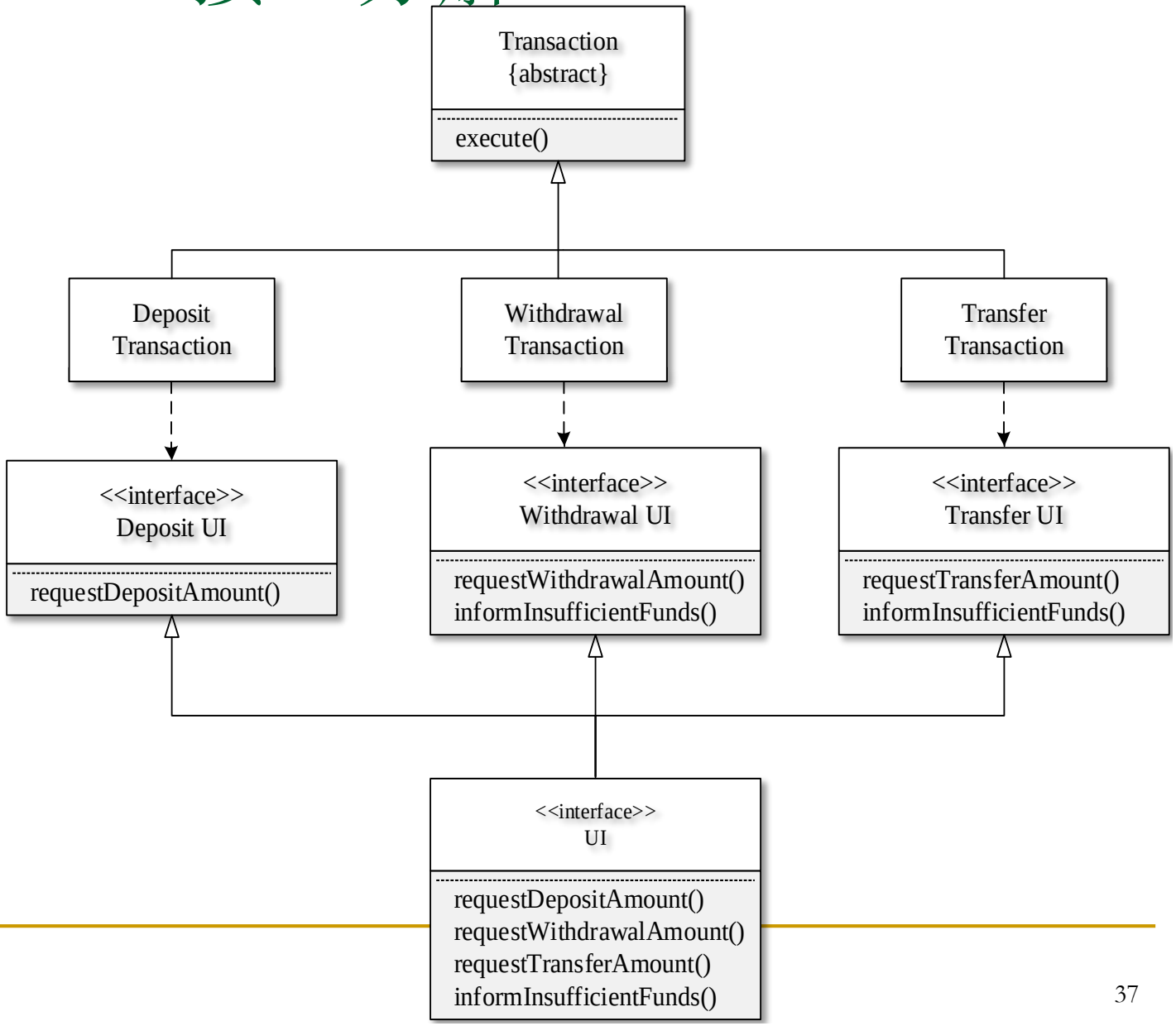
接口污染

- 接口污染：过于臃肿的接口
- 不应当把几个不同的角色都交给同一个接口，而应该交给不同的接口。
- 将没有关系的接口合并在一起，形成一个臃肿的大接口，是对接口的污染。
- 一个没有经验的设计师往往想节省接口的数目，因此，将一些看上去差不多的接口合并。

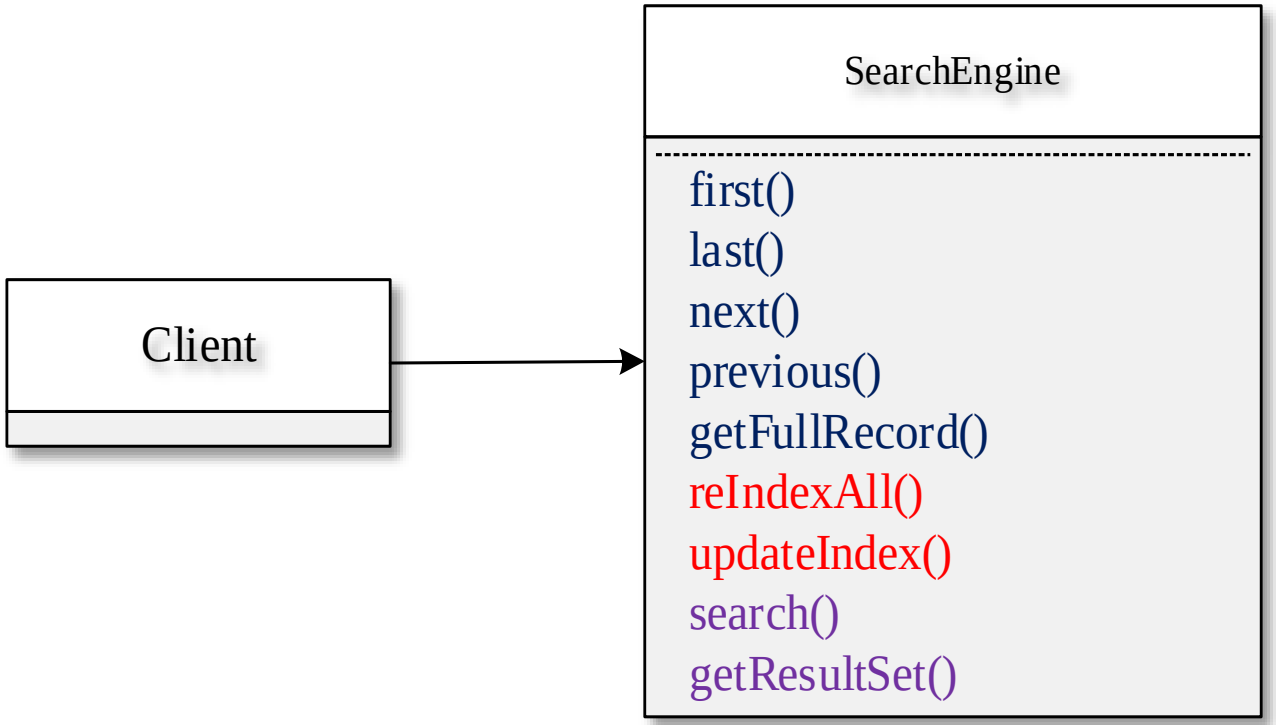
例：ATM


Transaction
{abstract}
execute()
Deposit
Transaction
Withdrawal
Transaction
Transfer
Transaction
<<interface>>
UI
requestDepositAmount()
requestWithdrawalAmount()
requestTransferAmount()
informInsufficientFunds()
Screen UI
Braille UI
Speech UI

例：ATM—接口分解

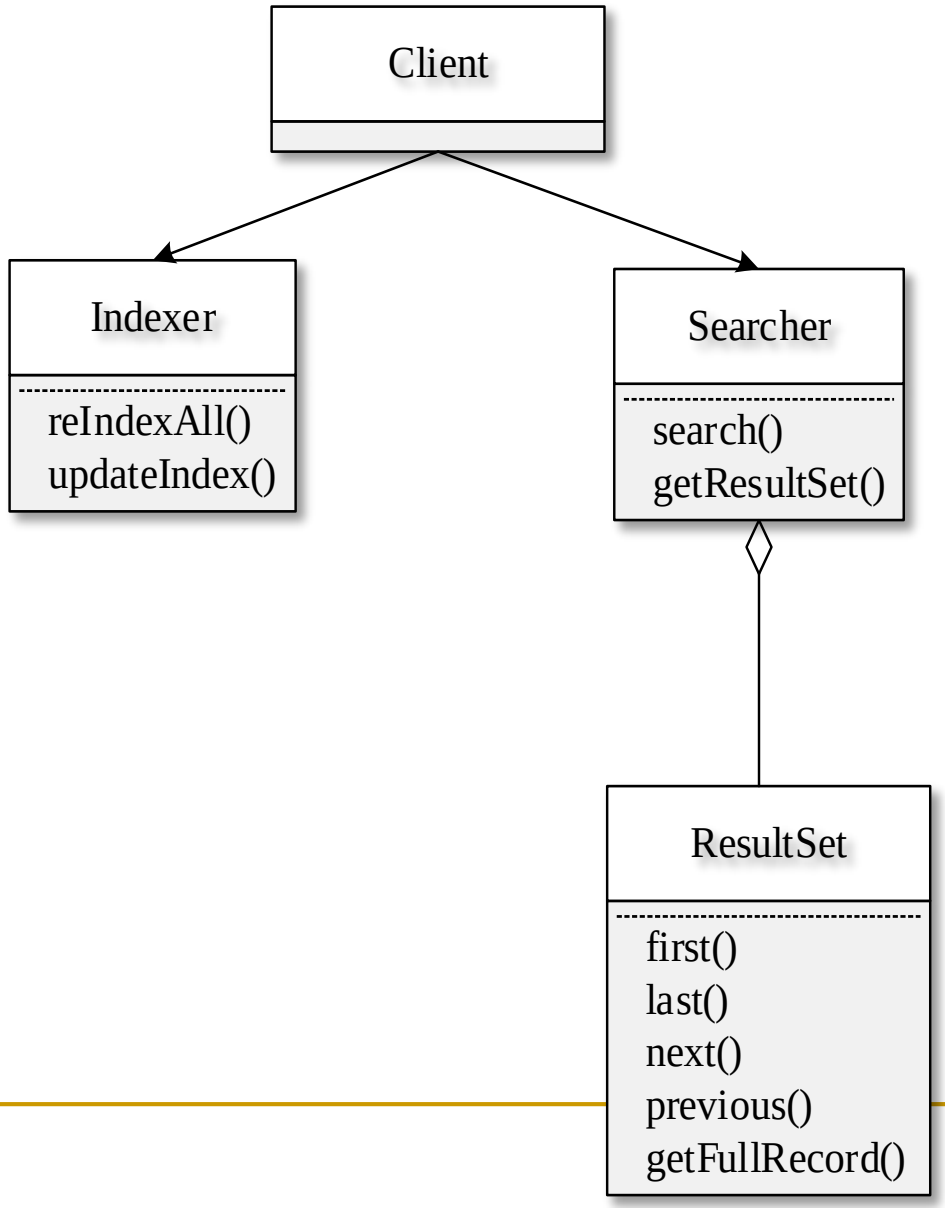

Transaction
{abstract}
execute()
Deposit
Transaction
Withdrawal
Transaction
Transfer
Transaction
<<interface>>
Deposit UI
requestDepositAmount()
<<interface>>
Withdrawal UI
requestWithdrawalAmount()
informInsufficientFunds()
<<interface>>
Transfer UI
requestTransferAmount()
informInsufficientFunds()
<<interface>>
UI
requestDepositAmount()
requestWithdrawalAmount()
requestTransferAmount()
informInsufficientFunds()

例：全文搜索引擎的系统设计--Bad



在这个设计中，一个接口负责所有的操作，违反了接口隔离原则，把不同功能的接口放在一起，由一个接口给出包括**搜索器角色**、**索引生成器角色**以及**搜索结果集角色**在内的所有角色。

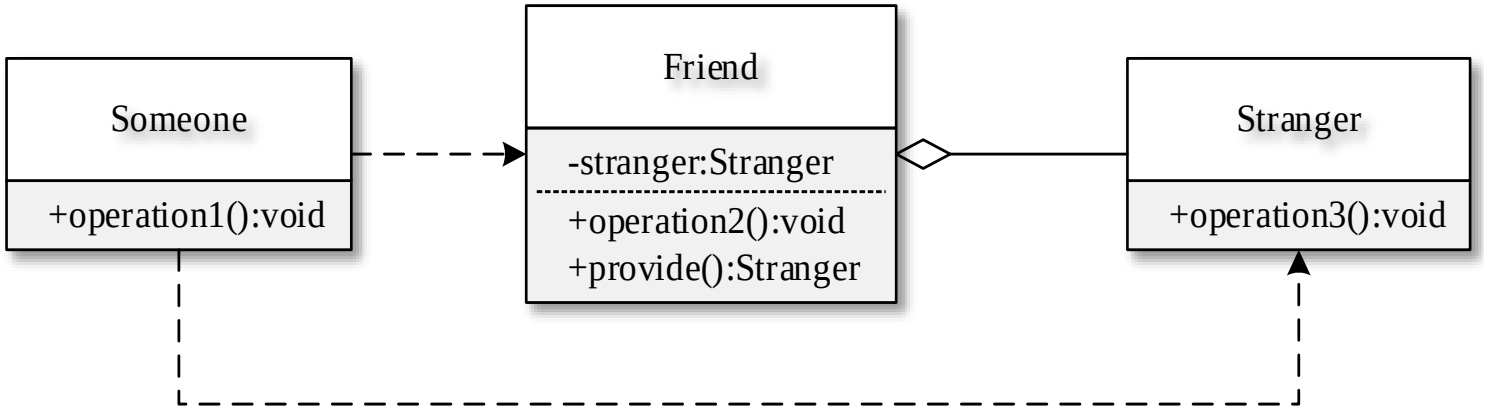
例：全文搜索引擎的系统设计--Good



迪米特法则

- 迪米特法则（Law of Demeter，LoD）又叫做最少知识原则（Least Knowledge Principle, LKP）。
  - Only talk to your immediate friends
  - Don’t talk to strangers
- 如果两个类不必彼此直接通信，那么这两个类就不应该发生直接的相互作用。如果其中的一个类需要调用另一个类的某一个方法的话，可以通过第三者转发这个调用。

不满足迪米特法则的系统



Someone与Friend是朋友，Friend与Stranger是朋友

```
public class Someone{
   public void operation1( Friend friend ){
      Stranger stranger = friend.provide() ;
      stranger.operation3() ;
   }
}
```

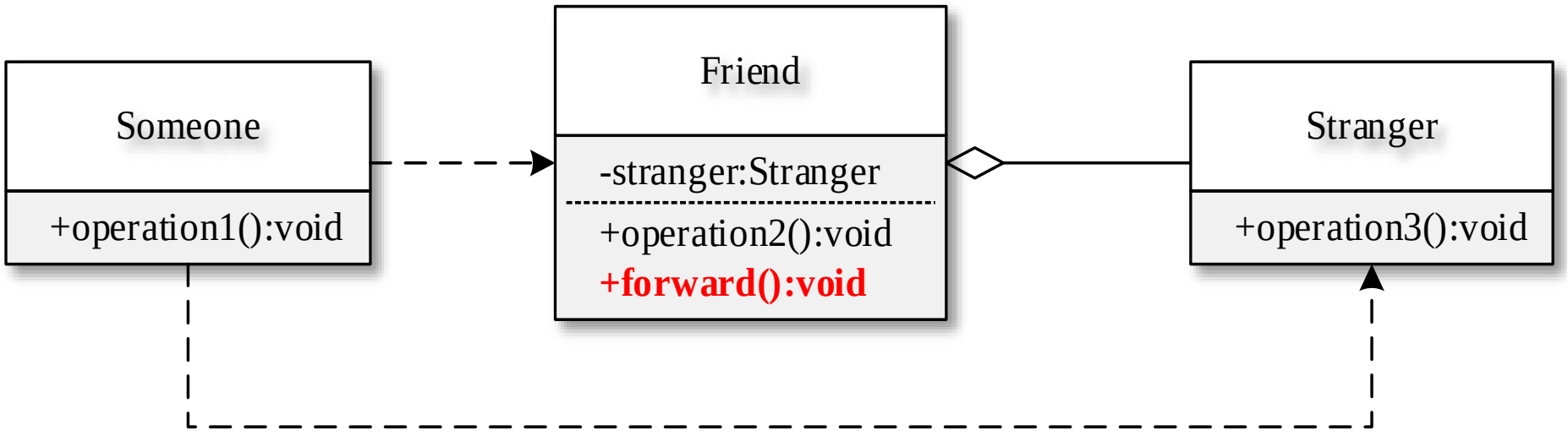
```
public class Friend{
   private Stranger stranger = new Stranger() ;
   public void operation2(){}
   public Stranger provide(){
      return stranger ;
    }
}
```

Someone的方法operation1()不满足迪米特法则。

- operation1()方法引用了Stranger对象，而Stranger对象不是Someone的朋友。

使用迪米特法则进行改造

- 迪米特法则建议“某人”不要直接与“陌生人”发生相互作用，而是通过“朋友”与之发生直接的相互作用。
- “朋友”实际上起到了将“某人”对“陌生人”的调用转发给“陌生人”的作用。这种传递叫做调用转发（Call Forwarding）。



使用迪米特法则进行改造—重构

```
public class Someone
 {
    public void operation1( Friend friend )
    {
       friend.forward() ;
    }
 }
```

Someone调用自己的朋友Friend对象的forward()方法，实现了对Stranger对象的访问。

```
public class Friend
 {
    private Stranger stranger = new Stranger() ;
    public void operation2(){}
    public void forward()
    {
       stranger.operation3() ;
    }
 }
```

forward()方法是转发方法，调用的细节被隐藏在Friend内部，使Someone与Stranger之间的直接联系被省略掉了，降低了系统内部的耦合度。

合成/聚合复用原则—CARP

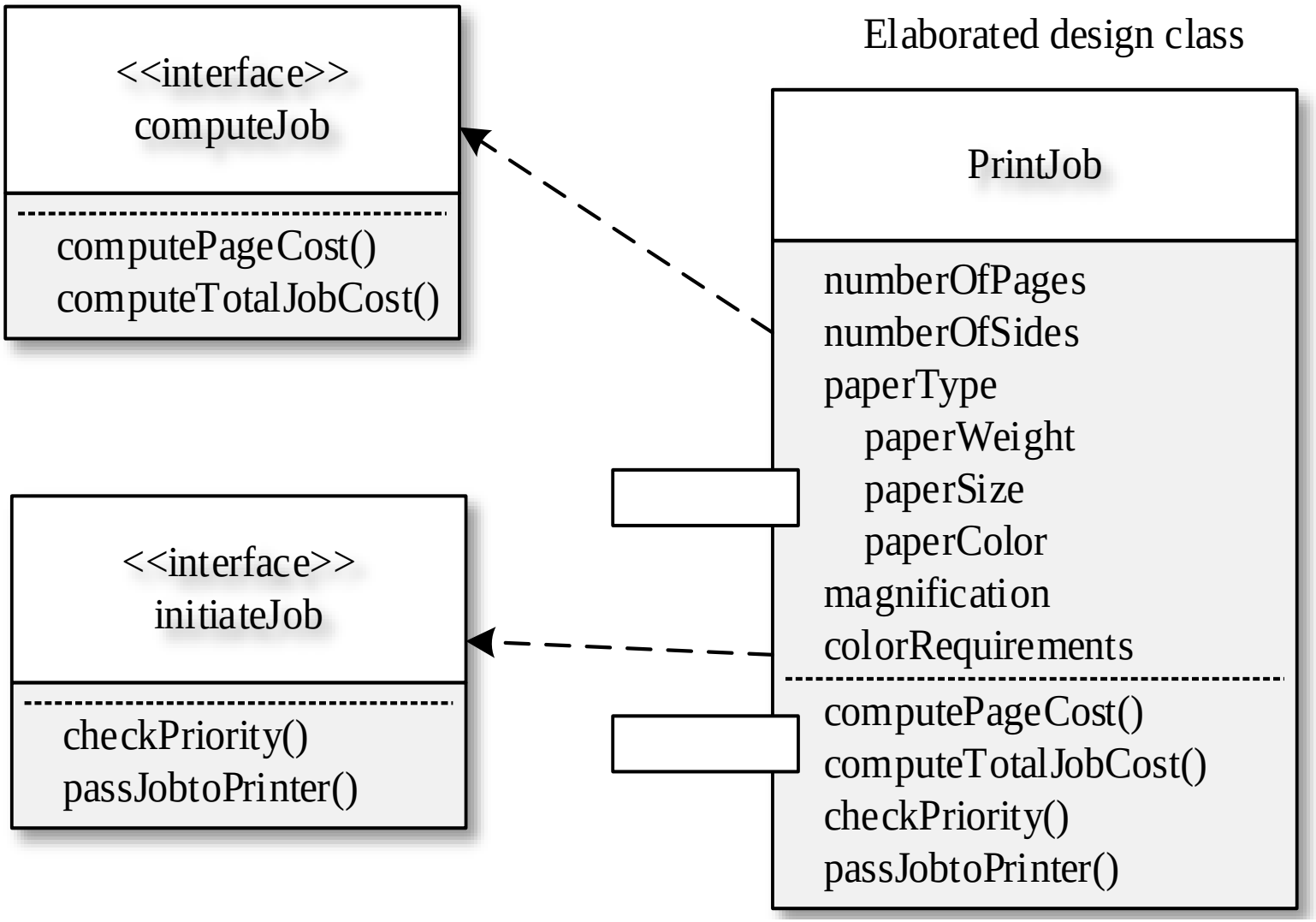
- 要尽量使用合成/聚合，尽量不要使用继承。
- CARP就是在一个新的对象里使用一些已有的对象，使之成为新对象的一部分，因此新的对象拥有原来的对象的功能。

- 构件定义
- 构件设计过程
- 软件重用
- 基本设计原则
- 构件详细设计
- 基于构件的软件工程

回顾：构件设计过程

- 构件设计过程是一个迭代、不断精化的过程
  - 第一次迭代：构件中的每个类都包括和实现相关的所有属性和方法
  - **第二次迭代：为每个属性定义合适的数据结构；为每个方法设计算法（活动图、过程设计方法）**
  - **定义每个类和其它类进行通信的所有接口**

构件详细设计实例


<<interface>>
computeJob
computePageCost()
computeTotalJobCost()
Elaborated design class
PrintJob
numberOfPages
numberOfSides
paperType
paperWeight
paperSize
paperColor
magnification
colorRequirements
computePageCost()
computeTotalJobCost()
checkPriority()
passJobtoPrinter()
<<interface>>
initiateJob
checkPriority()
passJobtoPrinter()

构件详细设计—属性

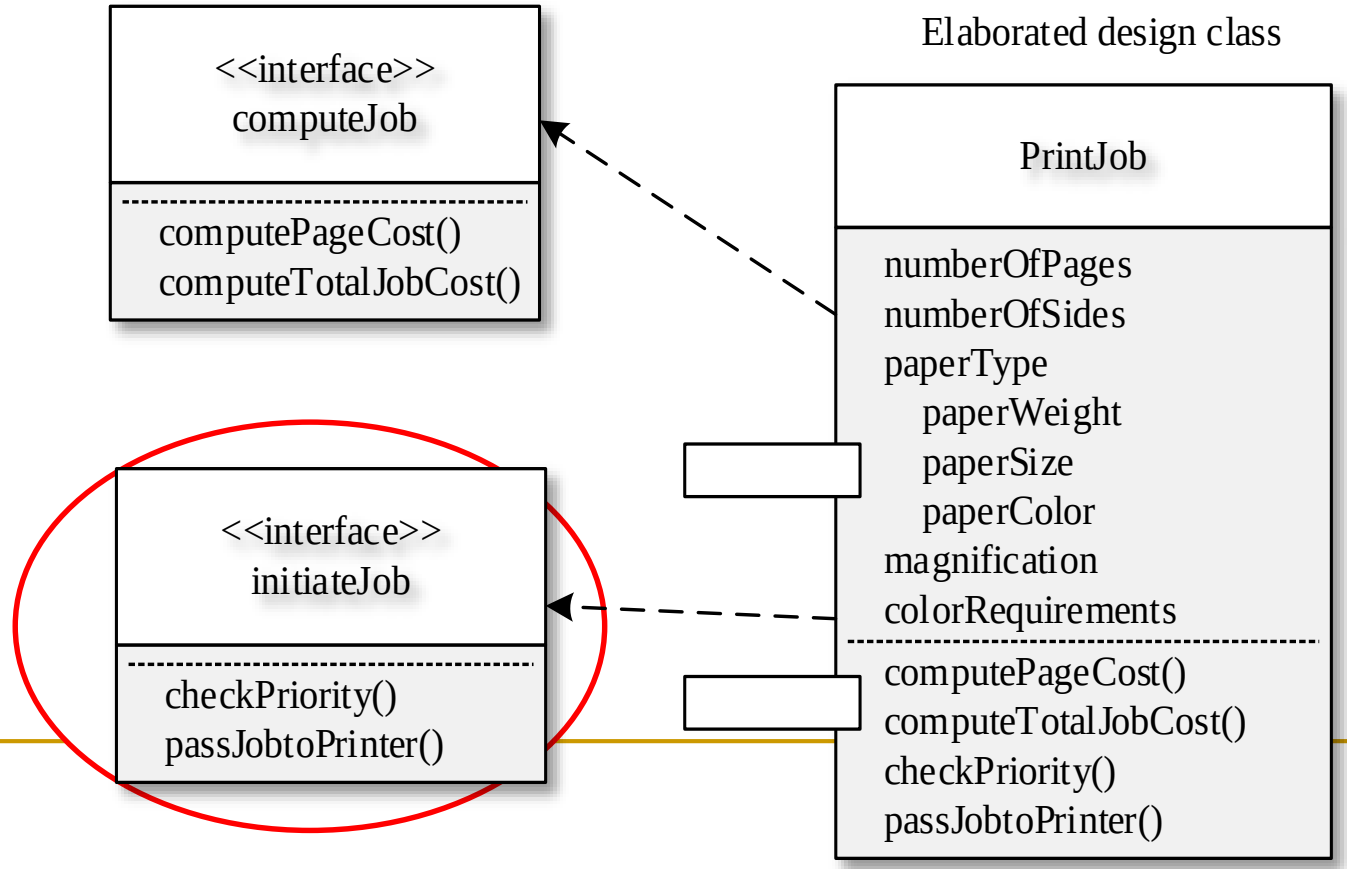
- 定义属性的数据结构和类型
  - 通常情况下，用实现阶段使用的编程语言来定义
  - 现阶段，使用UML属性格式定义属性的数据类型
    - 第一次迭代：属性只有名称
    - 第二次迭代：属性增加数据格式

name: type-express = initial-value {property string}

paperType-weight: string = “A” {contains 1 of 4 values – A, B, C, D}

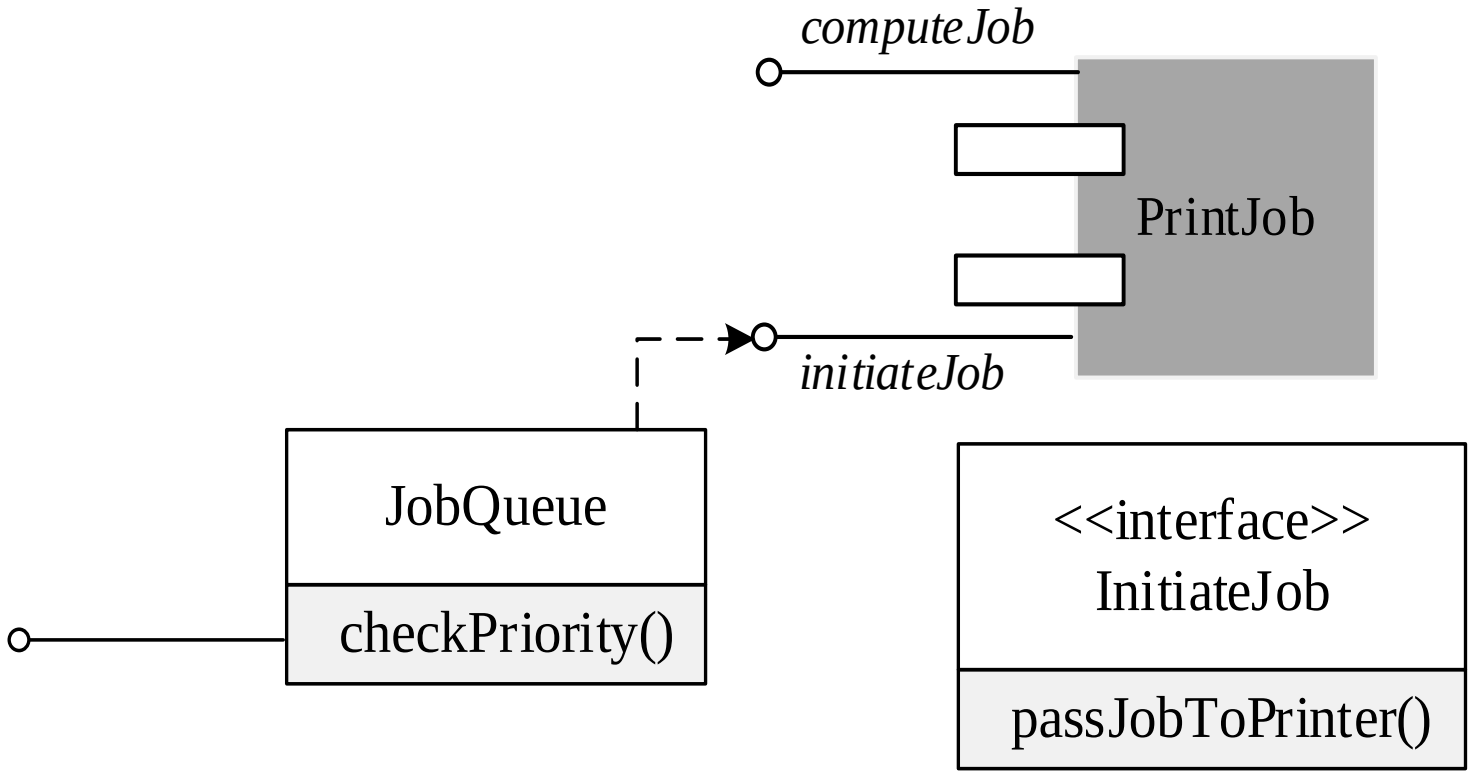
构件详细设计—接口

- 在构件级设计中，接口是一组外部可见的操作
- 接口不包含内部结构、属性等
- 类的方法被归类为一个或多个接口



构件详细设计—接口

- 对*initiateJob*接口进行重构
  - 定义一个新类**JobQueue**，包含*checkPriority()*操作
  - *initiateJob*接口只有一个功能，高内聚



构件详细设计—方法

- 定义方法的详细算法
  - 活动图
  - 伪代码
- 多次迭代、逐步求精
  - 第一次迭代：方法只有名称，高内聚
  - 第二次迭代：扩展方法的名称
  - 第三次迭代：算法描述

computePaperCost() → computePaperCost(weight, size, color): numeric

→ UML活动图

传统的构件设计方法

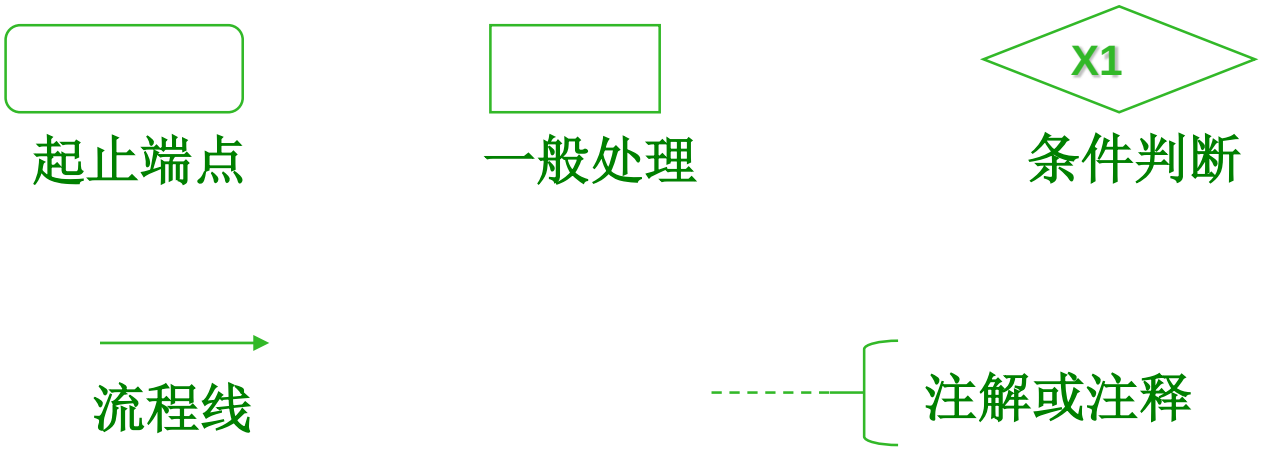
传统的构件设计使用以下三类工具：

- 图形工具： 将过程细节用图形来表示，在图中，逻辑结构用具体的图形表示。
- 列表工具：利用表来表示过程细节，表列出了各种操作和相应的条件。
- 语言工具：用类语言（伪码）表示过程的细节，很接近编程语言。

程序流程图

- 程序流程图又称为程序框图，它是历史最悠久、使用最广泛的描述过程设计的方法。
- 它的主要优点是对控制流程的描绘很直观，便于初学者掌握。
- 程序流程图历史悠久，至今仍在广泛使用着。

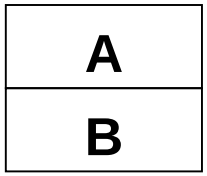
程序流程图



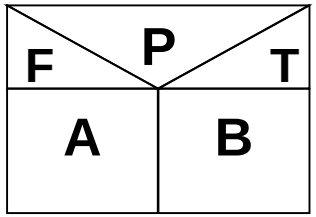
程序流程图中使用的符号

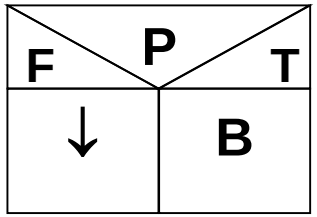
盒图(N-S图)

盒图(Box Diagram): Nassi &Shneiderman 1973年提出，又称为N-S Charts。Chapin 1974年作扩充，故也称为Chapin charts。

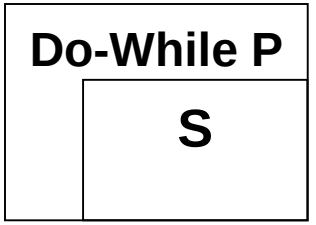


Sequential

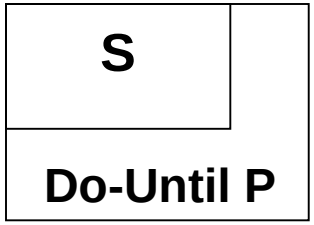




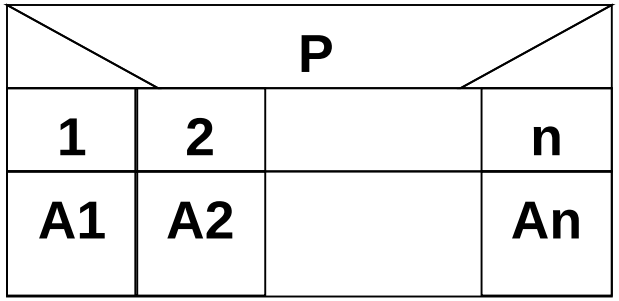
Selective



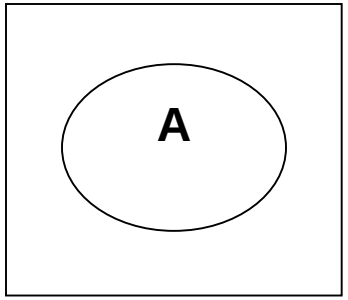
While



Until



Case

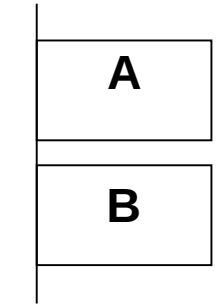


Call subroutine

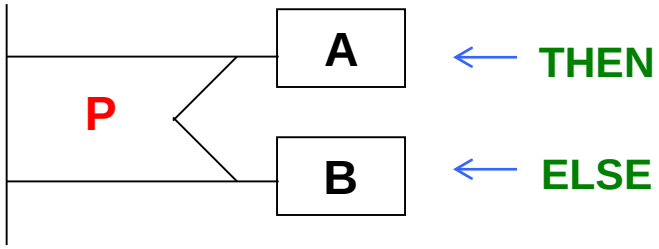
PAD图

- **PAD是问题分析图(problem analysis diagram)的英文缩写，自1973年由日本日立公司发明以后，已得到一定程度的推广。**
- **它用二维树形结构的图来表示程序的控制流，将这种图翻译成程序代码比较容易。**

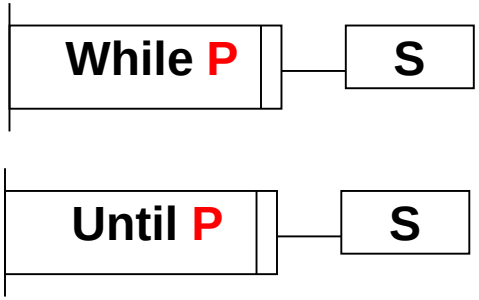
PAD图



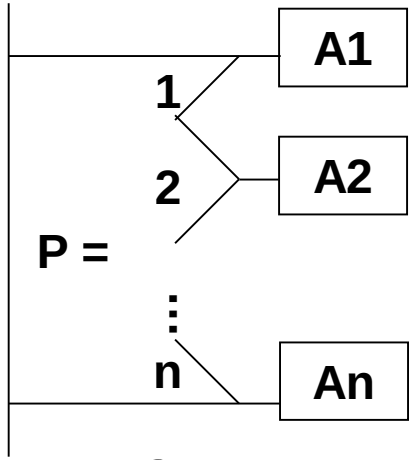
Sequential



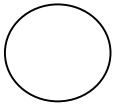
Selective



Loops



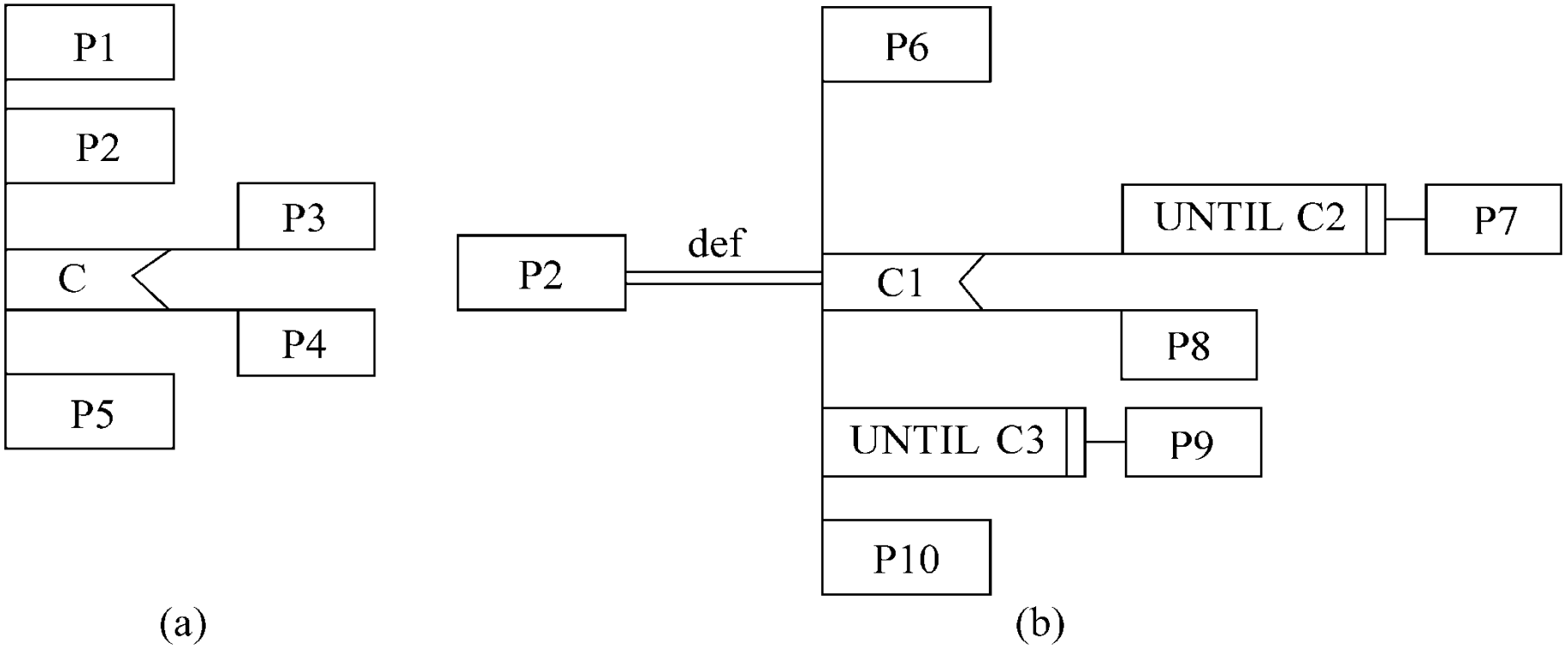
Case



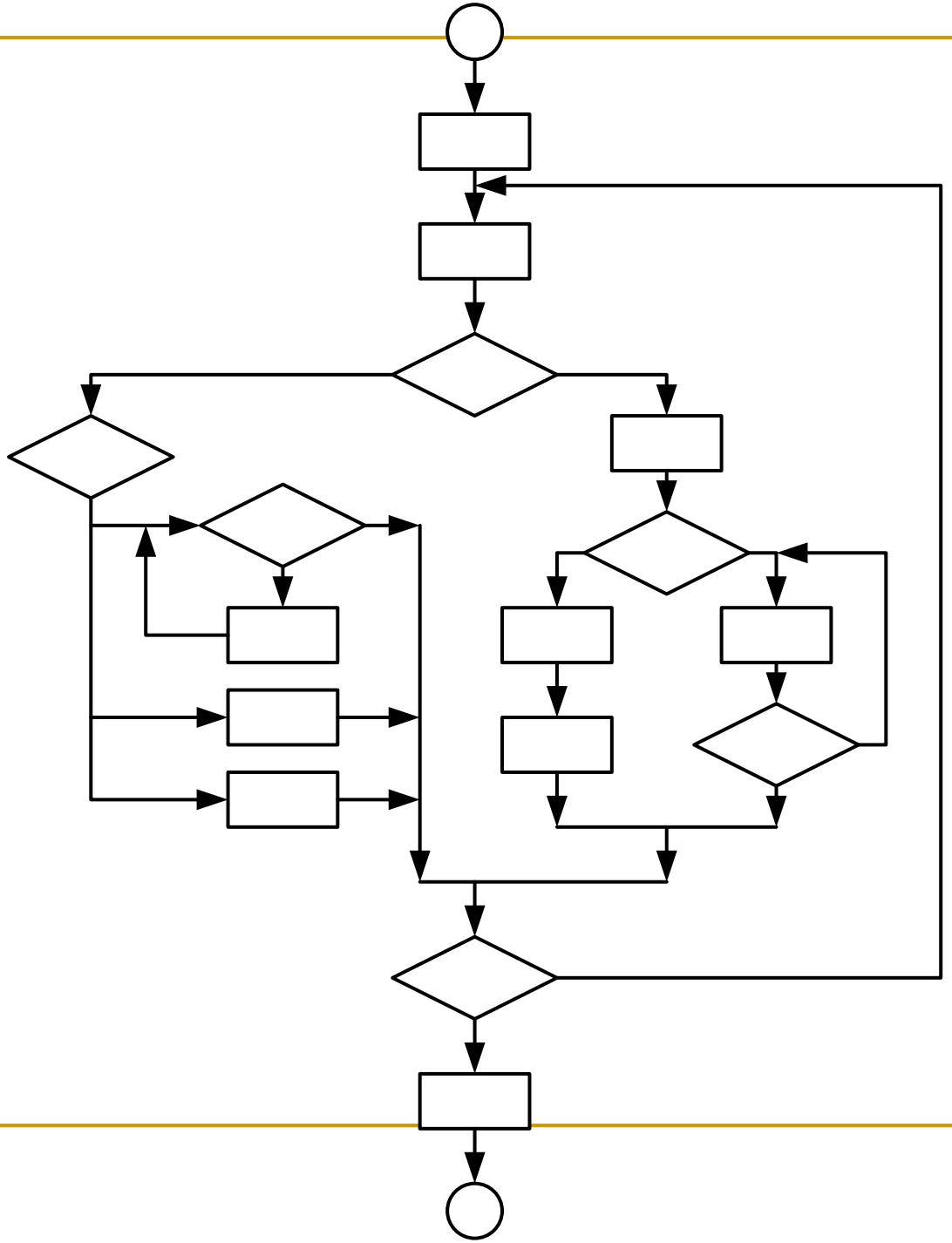
Statement Index

Definition

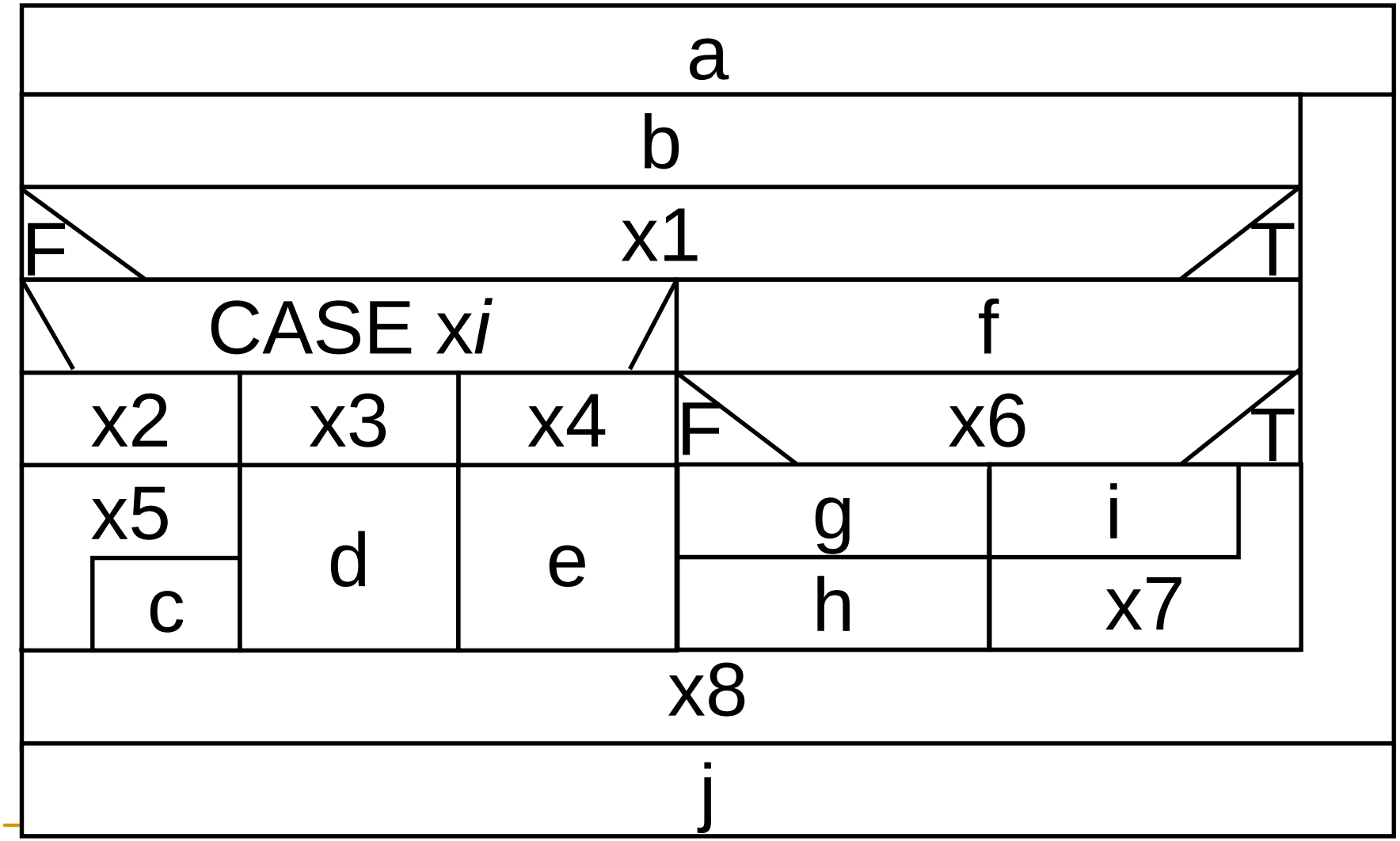
PAD图例



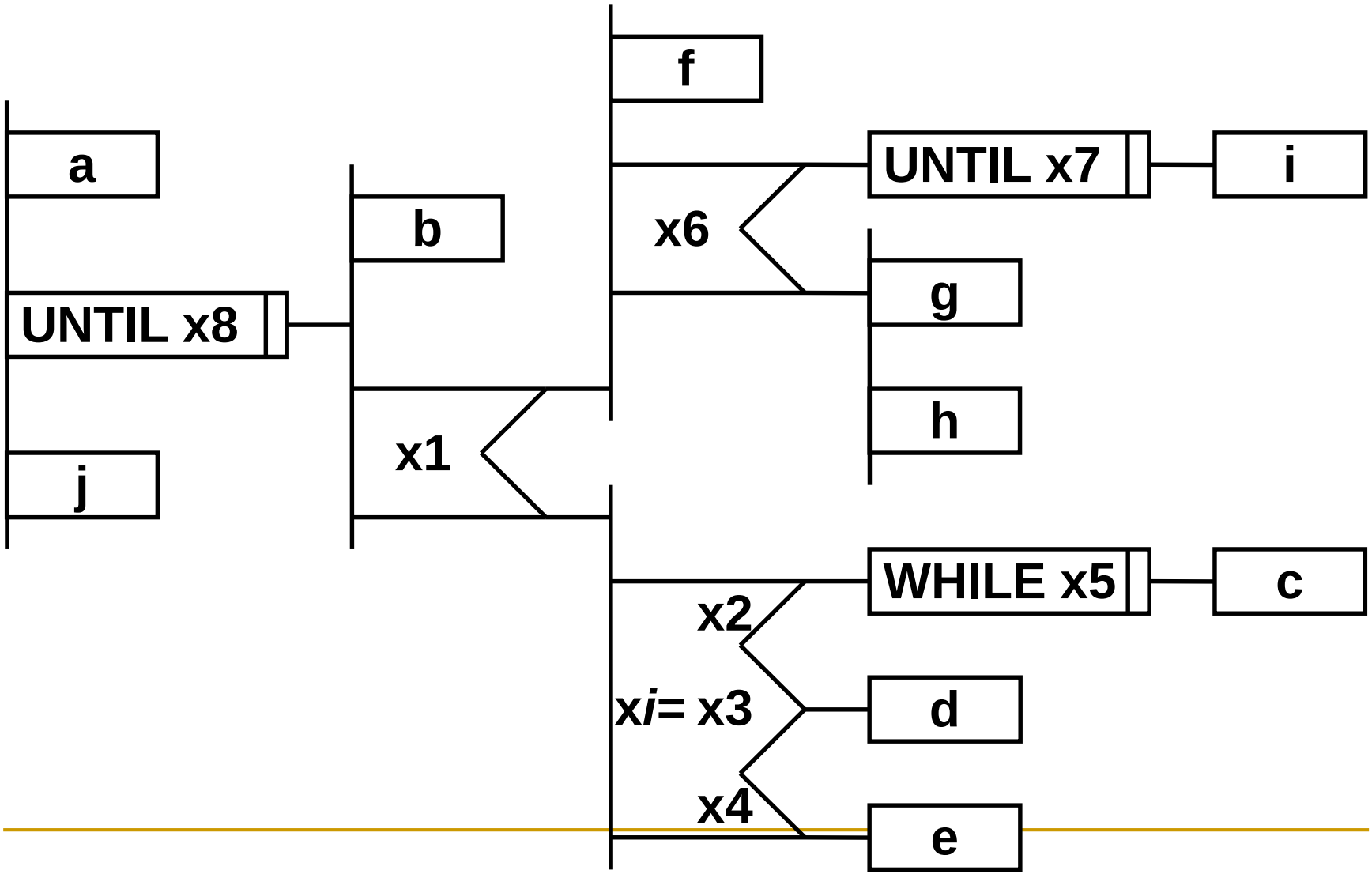
- **例题：某程序流程图如右图所示，请分别用N-S图和PAD图表示。**



N-S图：



PAD图



判定表

- 当算法中包含多重嵌套的条件选择时，用程序流程图、盒图、PAD图或后面即将介绍的过程设计语言(PDL)都不易清楚地描述。
- 判定表却能够清晰地表示复杂的条件组合与应做的动作之间的对应关系。

一张判定表由4部分组成：

- 左上部列出所有条件；
- 左下部是所有可能做的动作；
- 右上部是表示各种条件组合的一个矩阵；
- 右下部是和每种条件组合相对应的动作。

所有条件	条件组合矩阵
所有动作	条件组合 对应的动作

例题

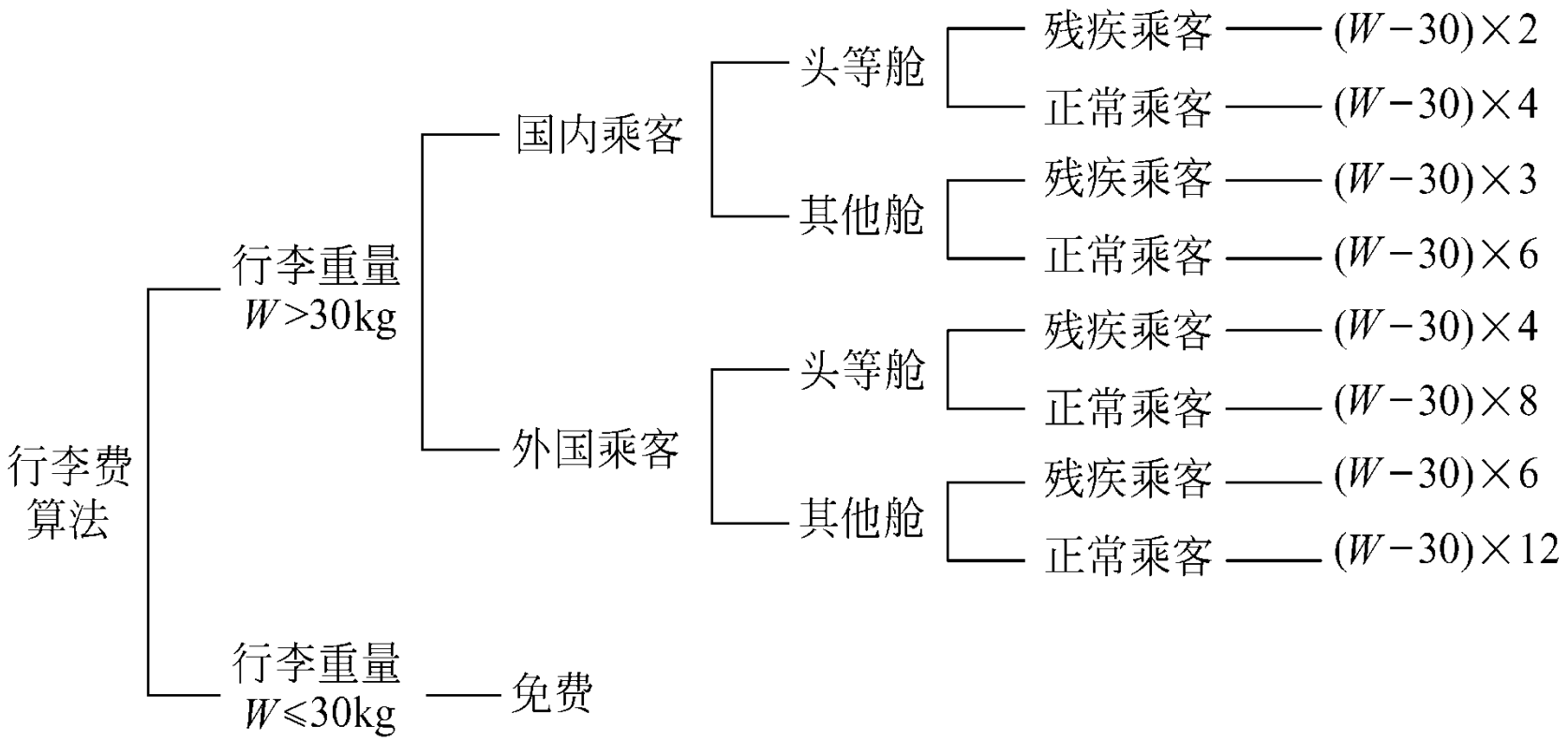
- 假设某航空公司规定，乘客可以免费托运重量不超过30kg的行李。
- 当行李重量超过30kg时，对头等舱的国内乘客超重部分每公斤收费4元，对其他舱的国内乘客超重部分每公斤收费6元。
- 对外国乘客超重部分每公斤收费比国内乘客多一倍，对残疾乘客超重部分每公斤收费比正常乘客少一半。

用判定表表示计算行李费的算法

	1	2	3	4	5	6	7	8	9
国内乘客		T	T	T	T	F	F	F	F
头等舱		T	F	T	F	T	F	T	F
残疾乘客		F	F	T	T	F	F	T	T
行李重量 $W \leq 30$	T	F	F	F	F	F	F	F	F
免费	√								
$(W-30)\times 2$				√					
$(W-30)\times 3$					√				
$(W-30)\times 4$		√						√	
$(W-30)\times 6$			√						√
$(W-30)\times 8$						√			
$(W-30)\times 12$							√		

判定树

- 判定树是判定表的变种，也能清晰地表示复杂的条件组合与应做的动作之间的对应关系。
- 多年来判定树一直受到人们的重视，是一种比较常用的系统分析和设计的工具。



用判定树表示计算行李费的算法

过程设计语言

- 过程设计语言(PDL)也称为伪码，它是用正文形式表示数据和处理过程的设计工具。
- PDL具有严格的关键字外部语法，用于定义控制结构和数据结构；另一方面，PDL表示实际操作和条件的内部语法通常又是灵活自由的，可以适应各种工程项目的需要。

例

```
void elevatorControllerEventLoop (void)
{
    while (TRUE)
     {
        if (an elevatorButton has been pressed)
          if (elevatorButton is off)
          {
              elevatorButton::turnOnButton;
              scheduler::newRequestMade;
          }
          ……
      }
}
```

- 构件定义
- 构件设计过程
- 软件重用
- 基本设计原则
- 构件详细设计
- 基于构件的软件工程

Component-based software engineering

- CBSE强调使用软件构件来设计和开发软件系统
- 包含两个子过程
  - 领域工程（domain engineering）：识别、开发、分类、传播面向特定应用领域的软件构件，形成可复用的构件库
  - 基于构件的开发
    - 构件合格性检验（qualification）确保候选的构件满足系统的功能需求和质量特性，完全适合系统的体系结构
    - 构件适应性修改（adaptation）确保构件库中的所有构件都有一致的资源管理方法，包括数据管理、接口等
    - 构件组装（composition）将合格的、适合的构件组装到体系结构中

构件标准

- Sun’s Enterprise Java Beans（EJB）
- Microsoft’s COM and .NET
- CORBA Component Model（CCM）