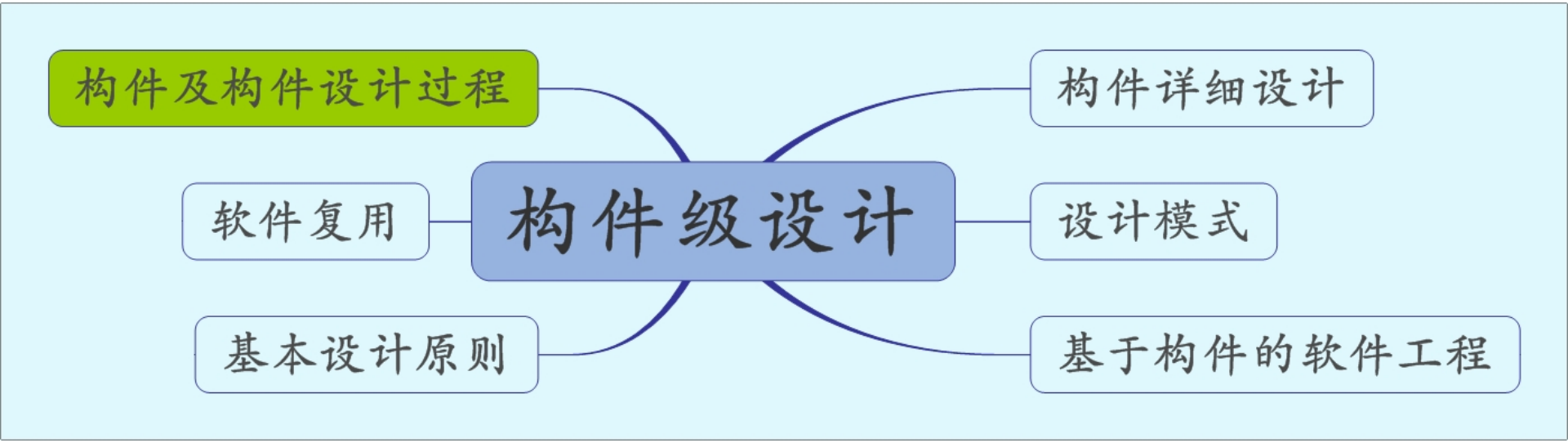
outline



构件定义

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

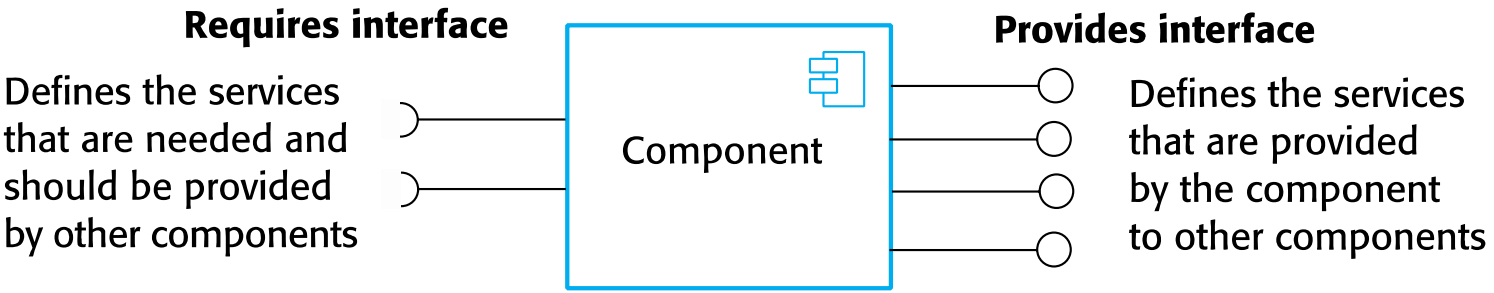
OMG Unified Modeling Language Specification

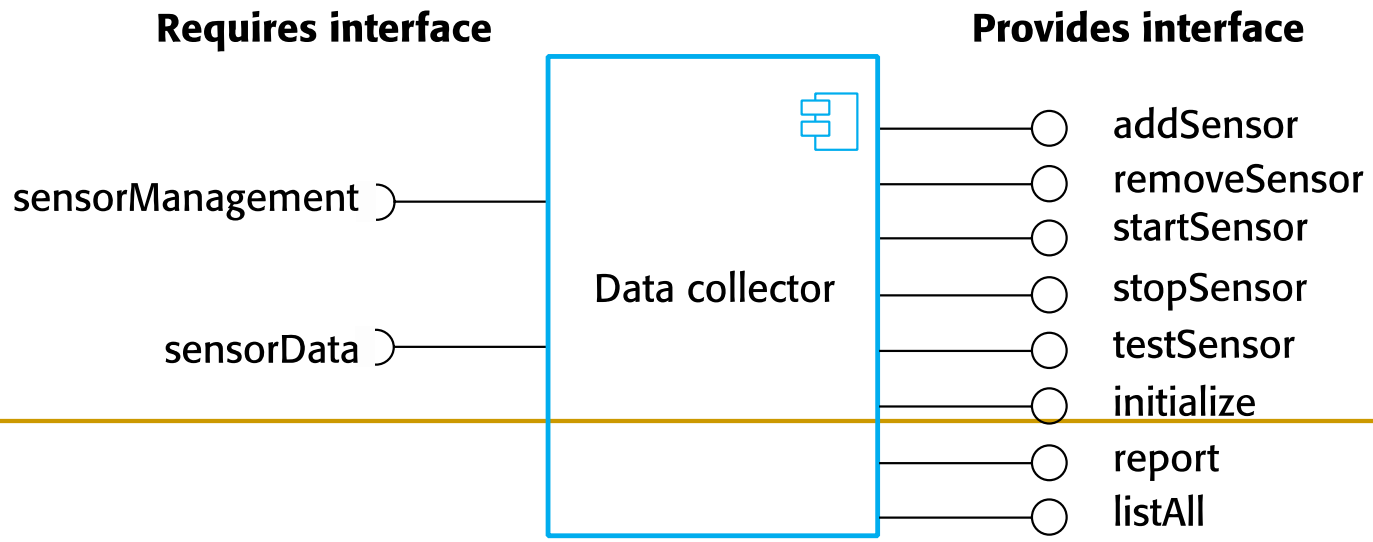
构件：服务提供者

- 构件是独立的服务提供者
- 当系统需要某一服务时，会调用提供相应服务的构件，而无须知道构件的位置和任何源代码信息
- 构件所提供的服务可以通过其接口得到，而且所有的交互都是通过接口实现的
- 构件接口表示为参数化的操作，其内部状态不会暴露

构件模型

- “提供”接口：定义了构件所提供的服务，API
- “请求”接口：定义了构件所需要的由其它构件提供的服务

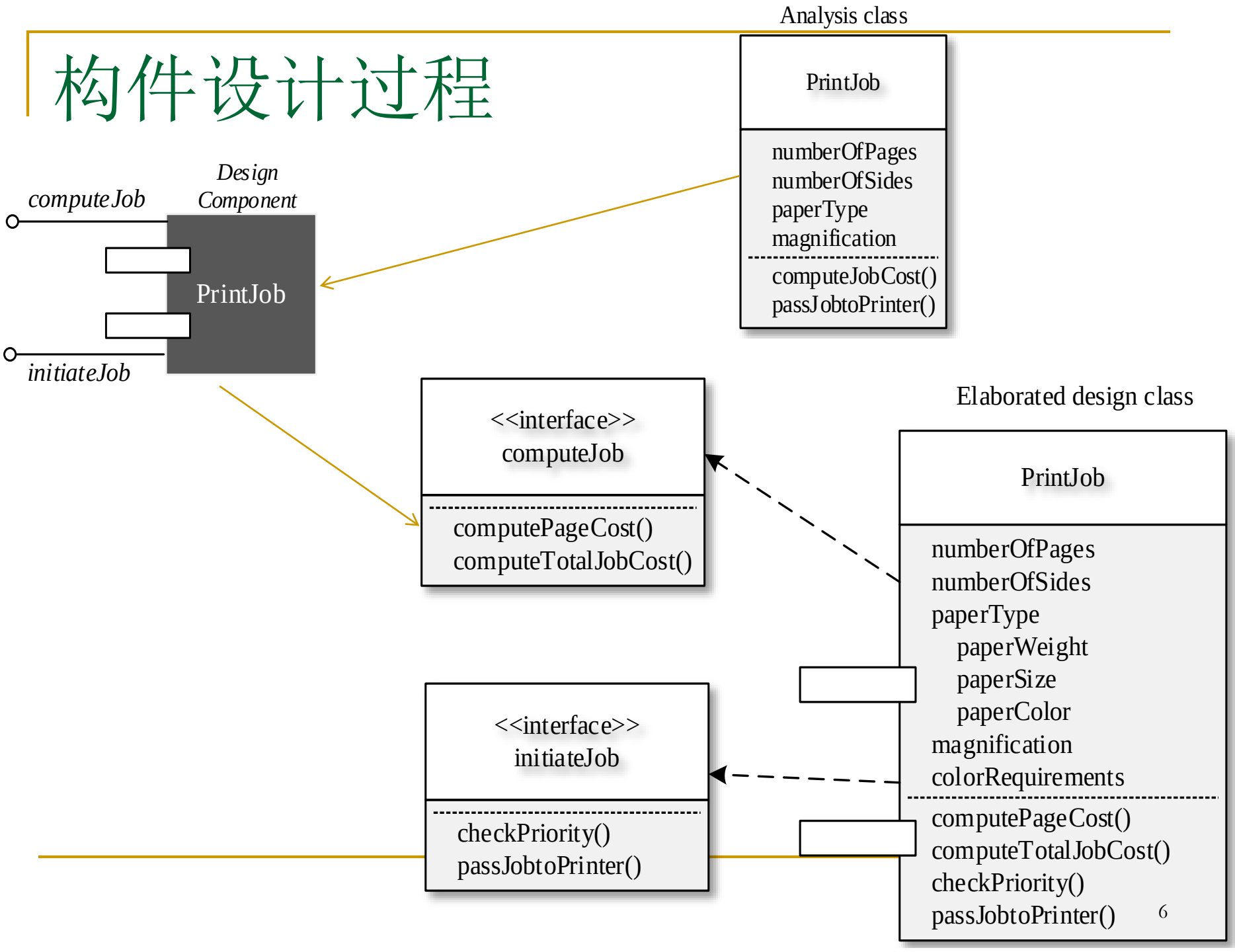




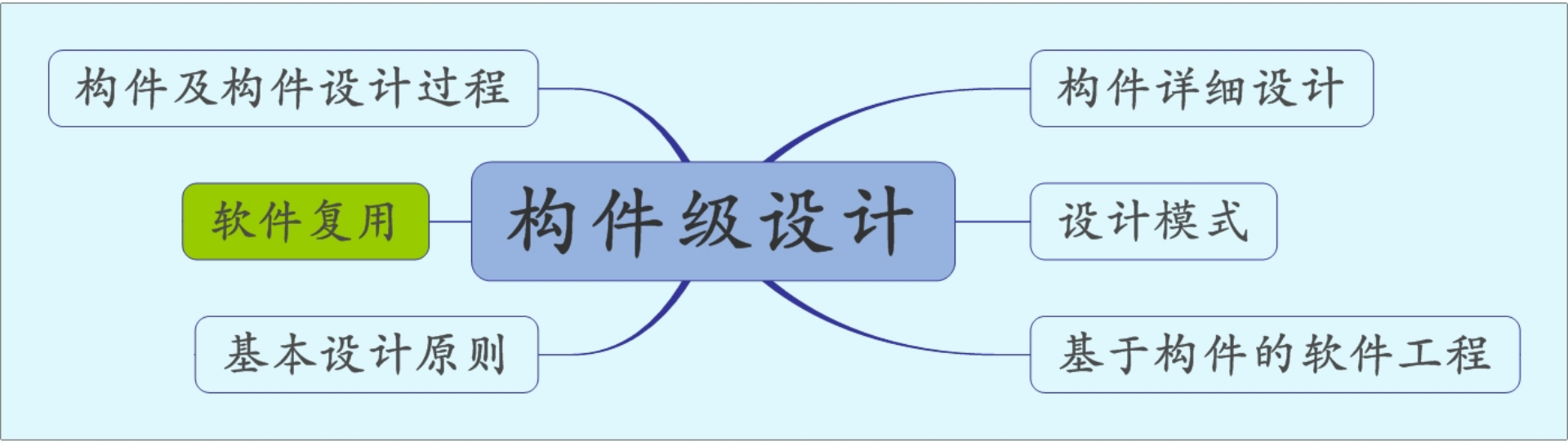
构件设计过程

- 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()

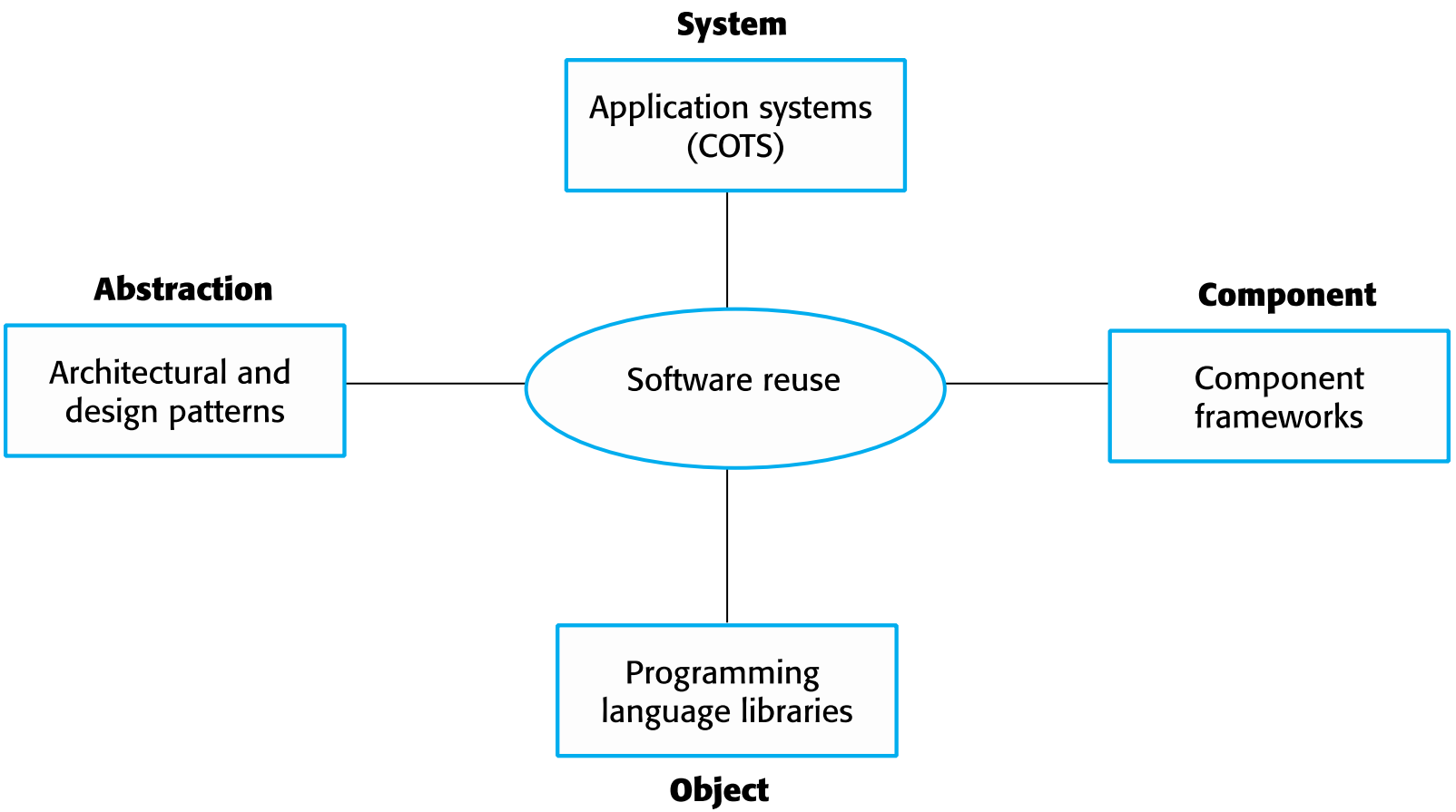
outline



软件复用

- 从1960s 到 1990s，大多数新软件都是从头开始开发
  - 唯一显著的软件复用是对编程语言库中的功能和对象的复用
- 成本和进度压力使这一方法越来越不可行，特别是对于商业系统和基于互联网的系统
- 基于复用已有软件的开发方法现在已经成为许多不同类型系统开发的基本准则
  - 基于Web的系统、科学软件、嵌入式系统

软件复用的级别


System
Application systems (COTS)
Abstraction
Architectural and design patterns
Software reuse
Component
Component frameworks
Programming language libraries
Object

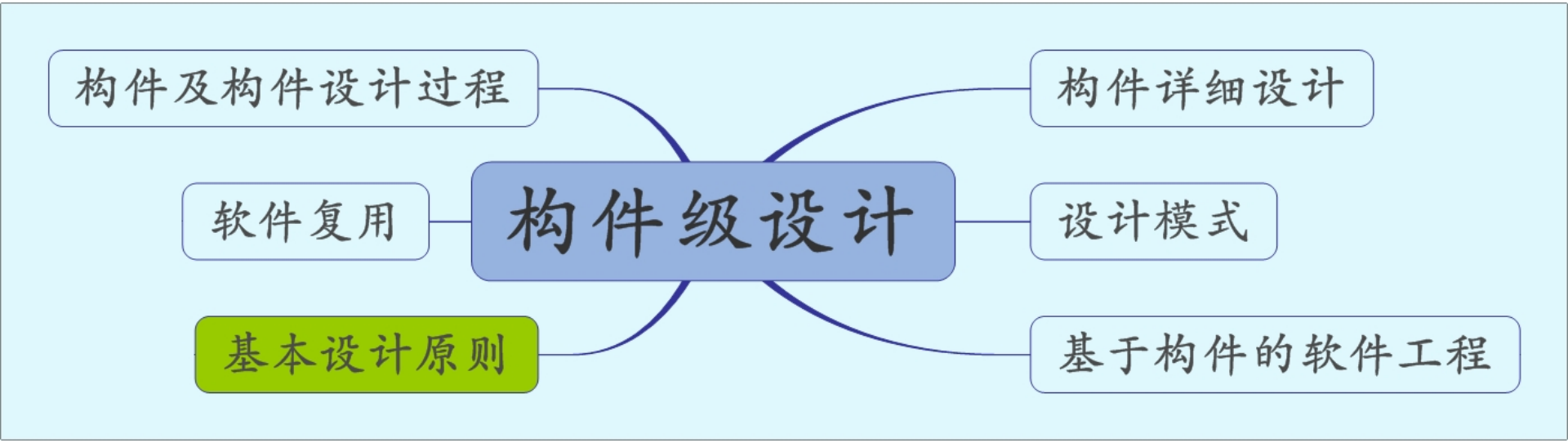
复用成本

- 通过复用已有的软件，可以更快地开发新系统，风险和成本都很低
- 存在一些与复用相关的成本
  - 寻找可复用的软件以及评价其是否满足需要所花费的时间成本
  - 在适用的情况下，购买可复用软件的成本
  - 适配和配置可复用软件构件或系统以反映正在开发的系统的需求的成本
  - 可复用软件元素相互之间集成以及与所开发的新代码相集成的成本

建议

- 如何复用已有的知识和软件应当是在开始一个软件开发项目时要考虑的第一件事
- 应当在详细设计之前考虑复用的可能性
- 在一个面向复用的开发过程中，要搜索可复用元素，然后修改需求和设计以充分利用这些可复用元素

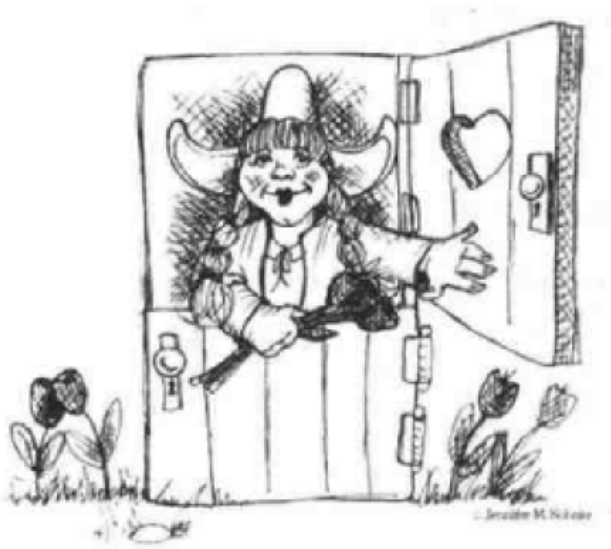
outline



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）

开闭原则



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

Robert C. Martin

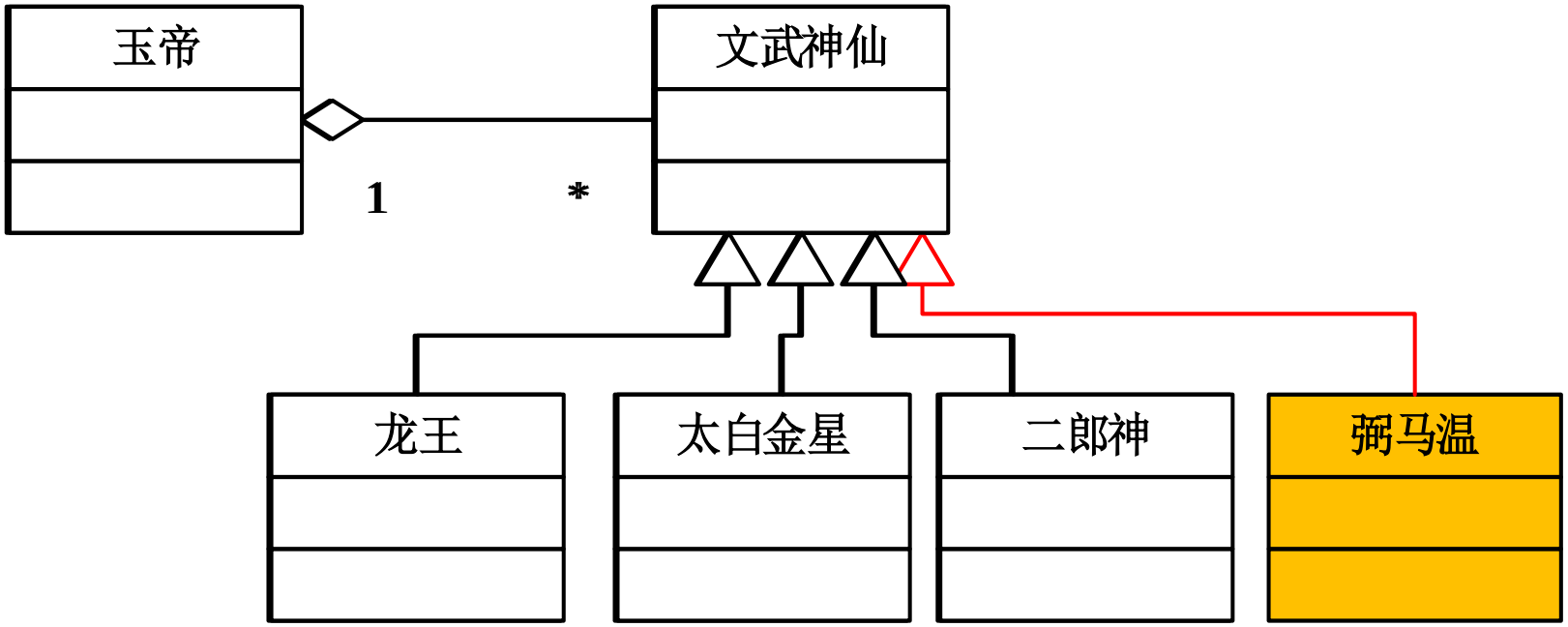
Design Principles and Design Patterns

开闭原则

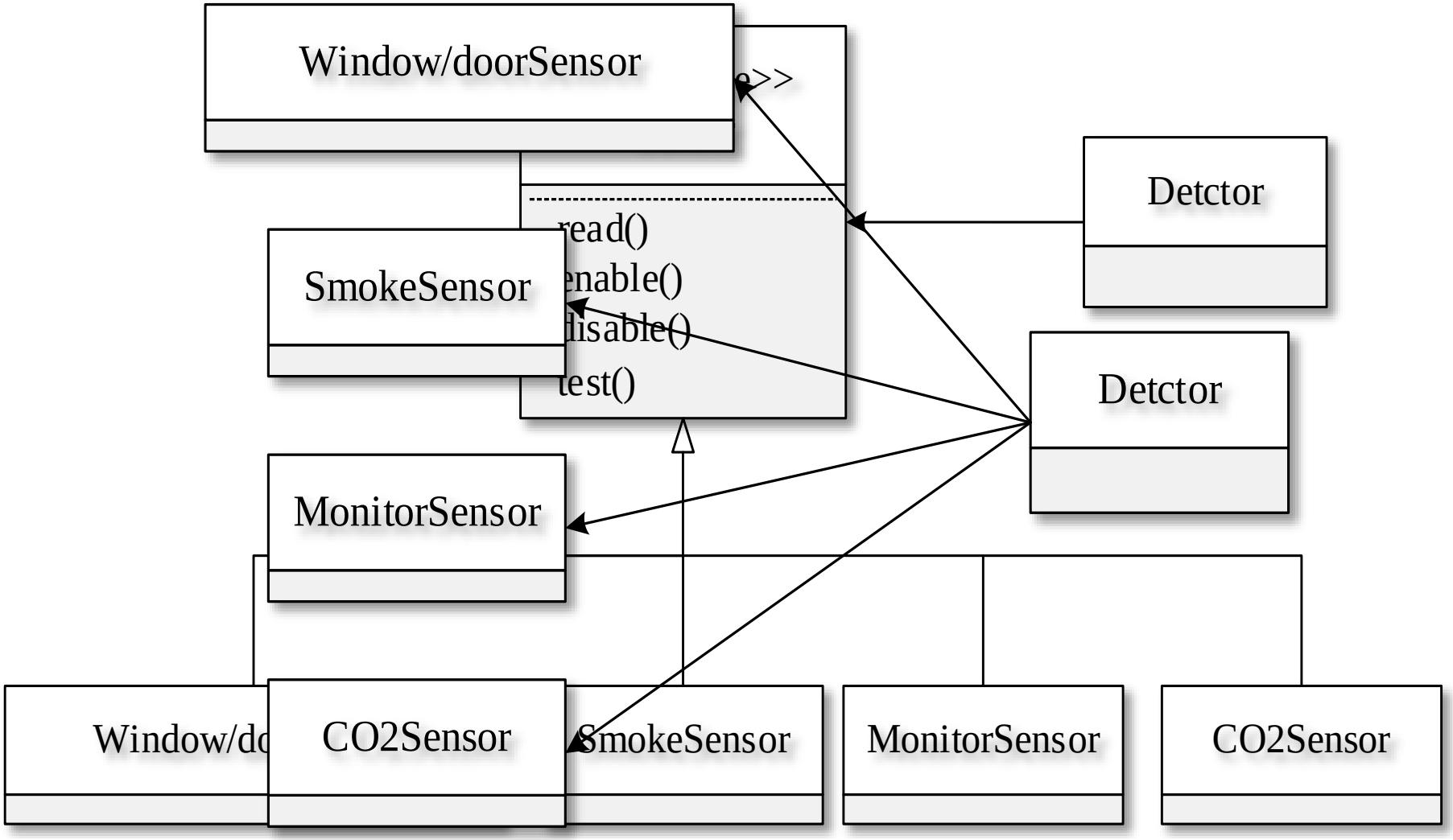
- 在设计一个构件时，应当让这个构件在不被修改（代码层）的前提下被扩展，即在不修改源代码的情况下改变这个模块的行为。
- 开闭原则的关键是抽象

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



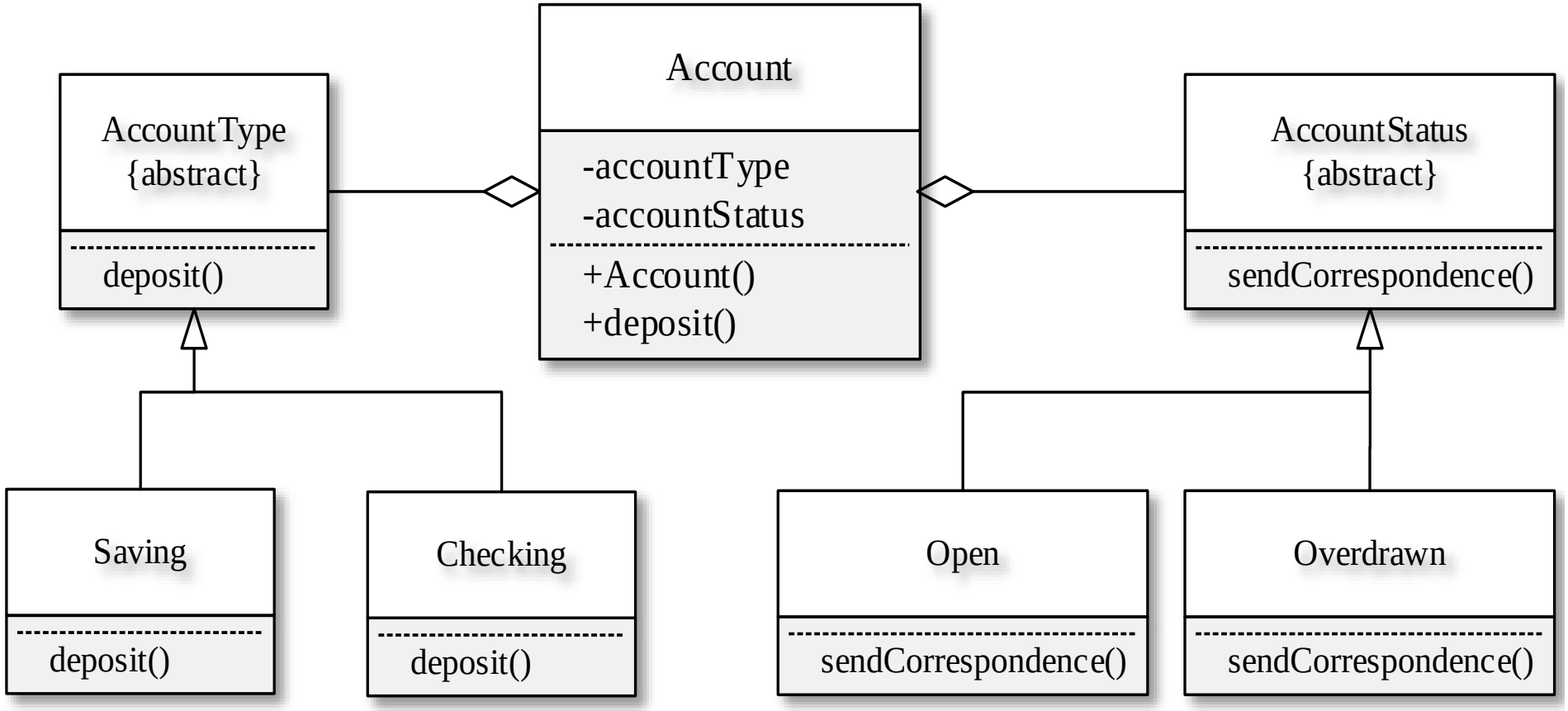
例2


Window/doorSensor
>>
Detctor
read()
SmokeSensor
enable()
disable()
test()
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.

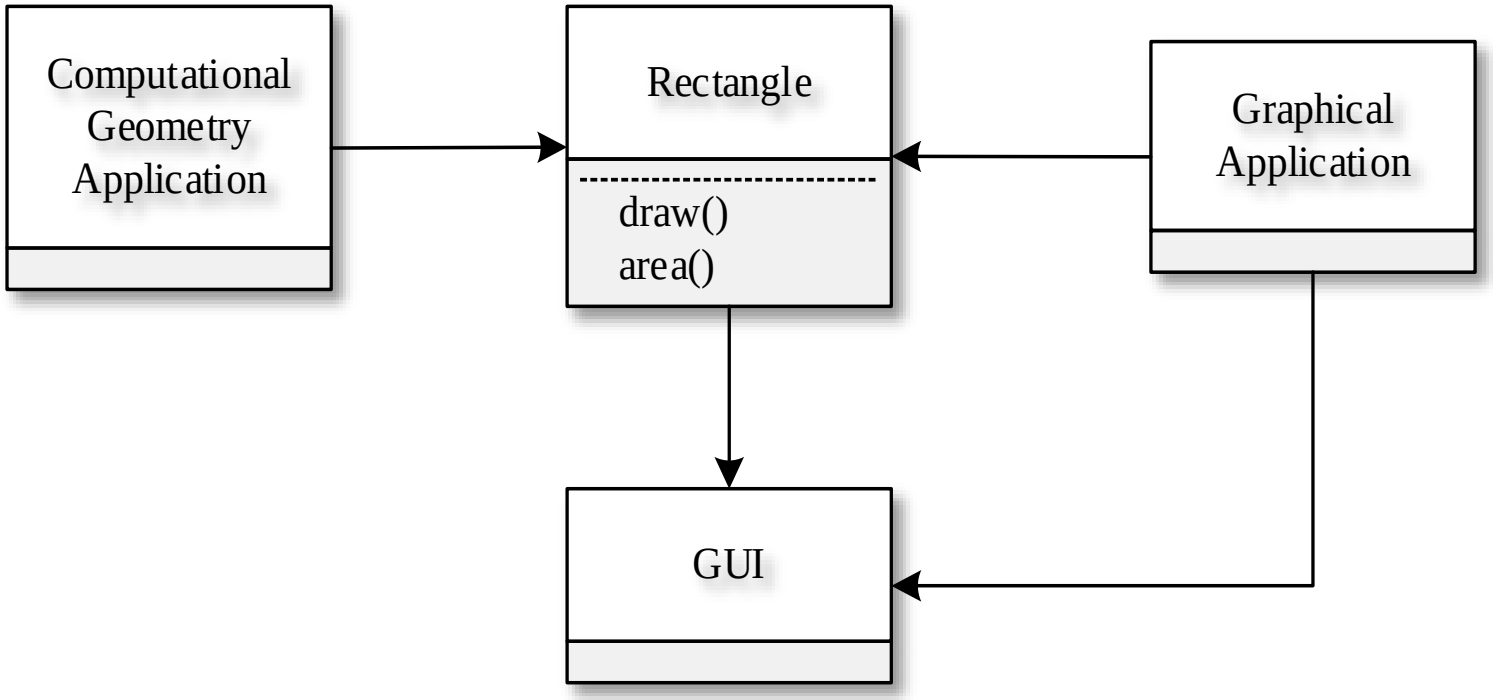
Robert C. Martin

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。

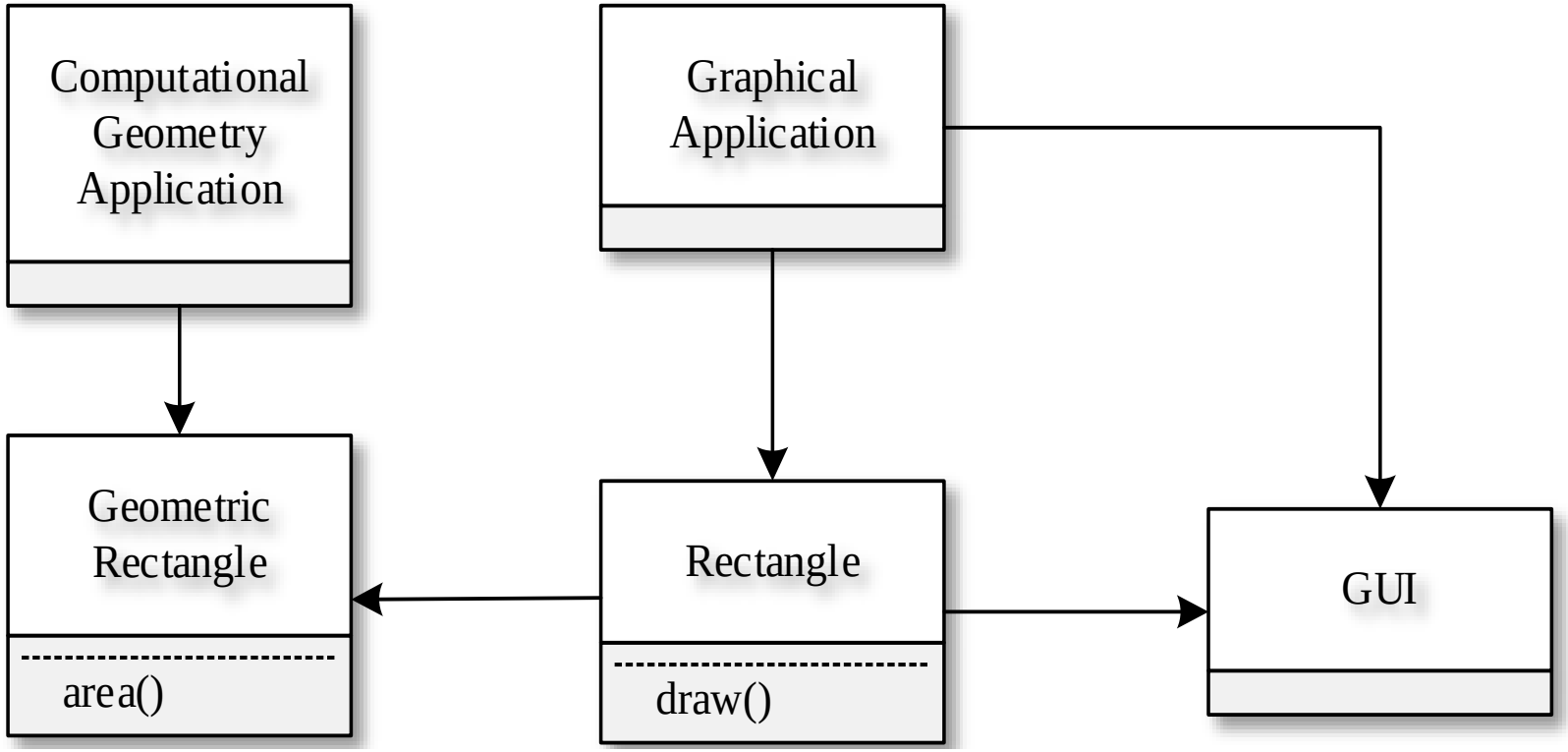
例



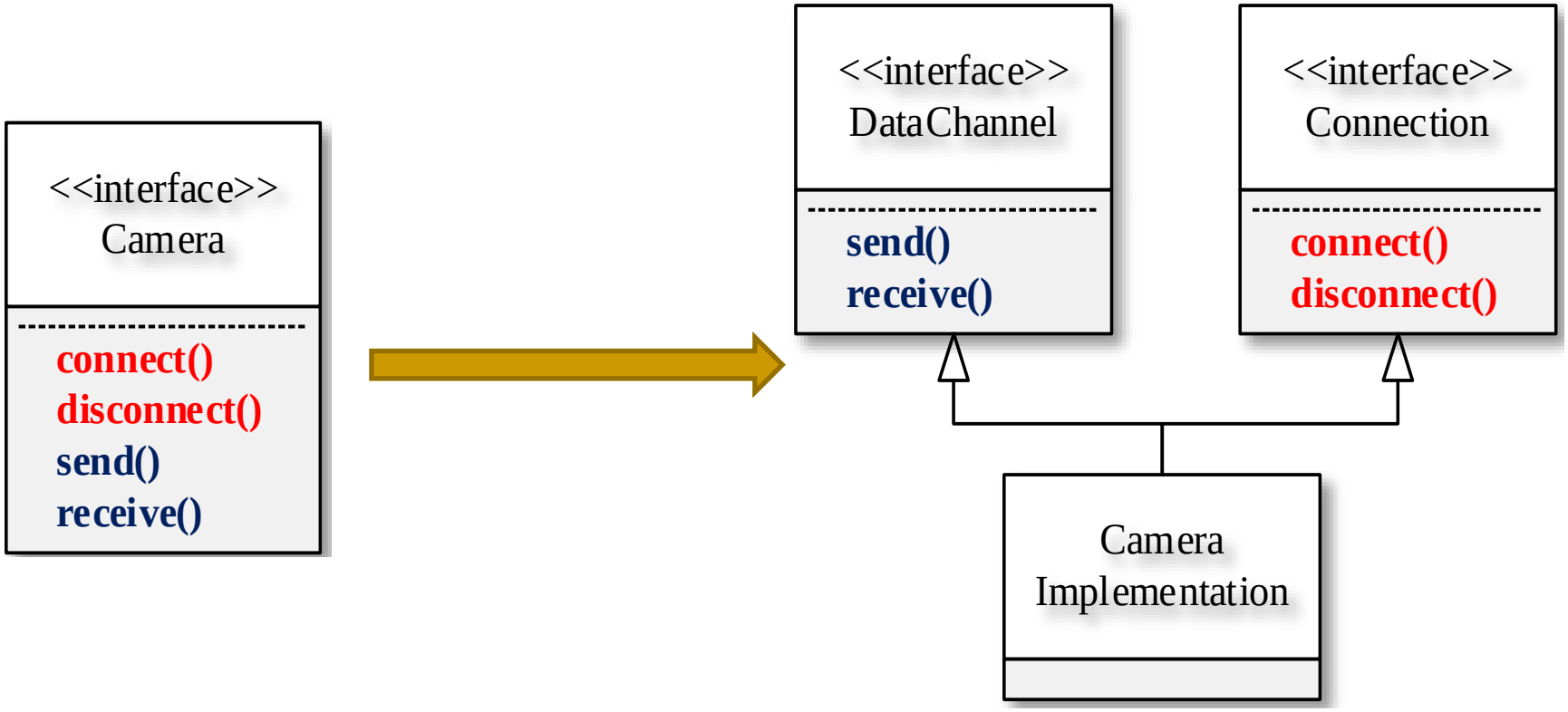
- Rectangle 类包含两个方法：draw方法负责在GUI上绘制矩形，area方法负责计算矩形图形面积。
- 违反了 SRP 原则，带来了如下问题：
  - 必须在计算几何应用中包含对 GUI 库的引用，导致应用程序无谓的消耗了链接时间、编译时间、内存空间和存储空间等。
  - 如果因为某些原因对图形应用的一个更改导致 Rectangle 类也相应做了更改，会影响计算几何应用的功能。

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

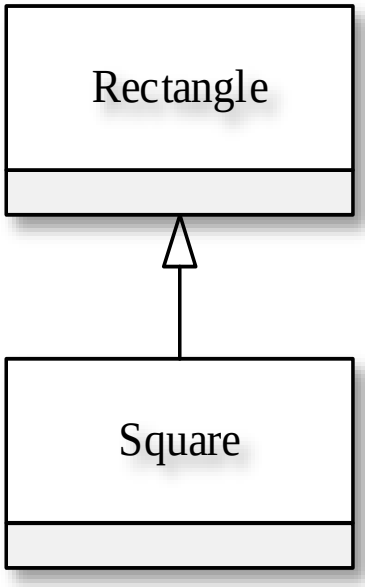
- 假设在一个应用程序中使用了 Rectangle 类
- 某一天用户提出新的需求，要求该应用程序除了能够处理Rectangle之外还要能够处理Square。

```
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

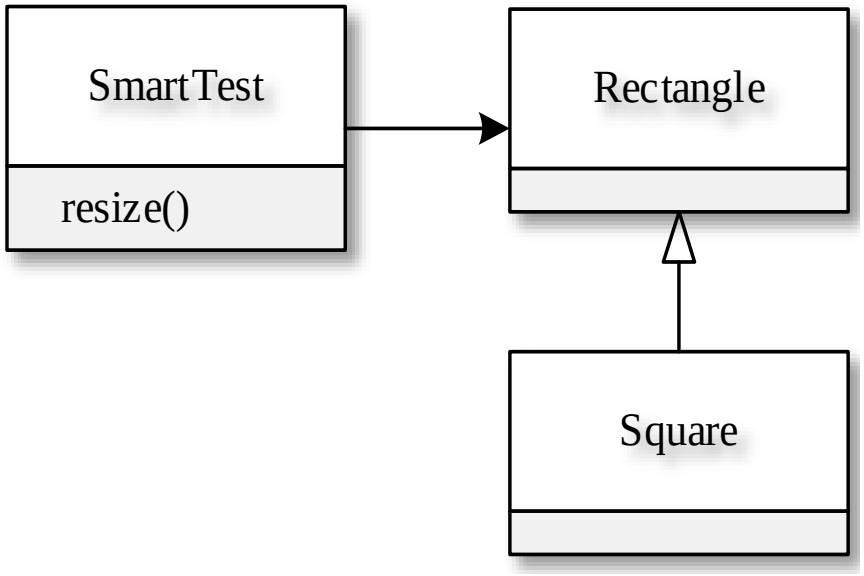
```
void f(Rectangle r)
{
  r.SetWidth(32);
}
```

OK

?

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

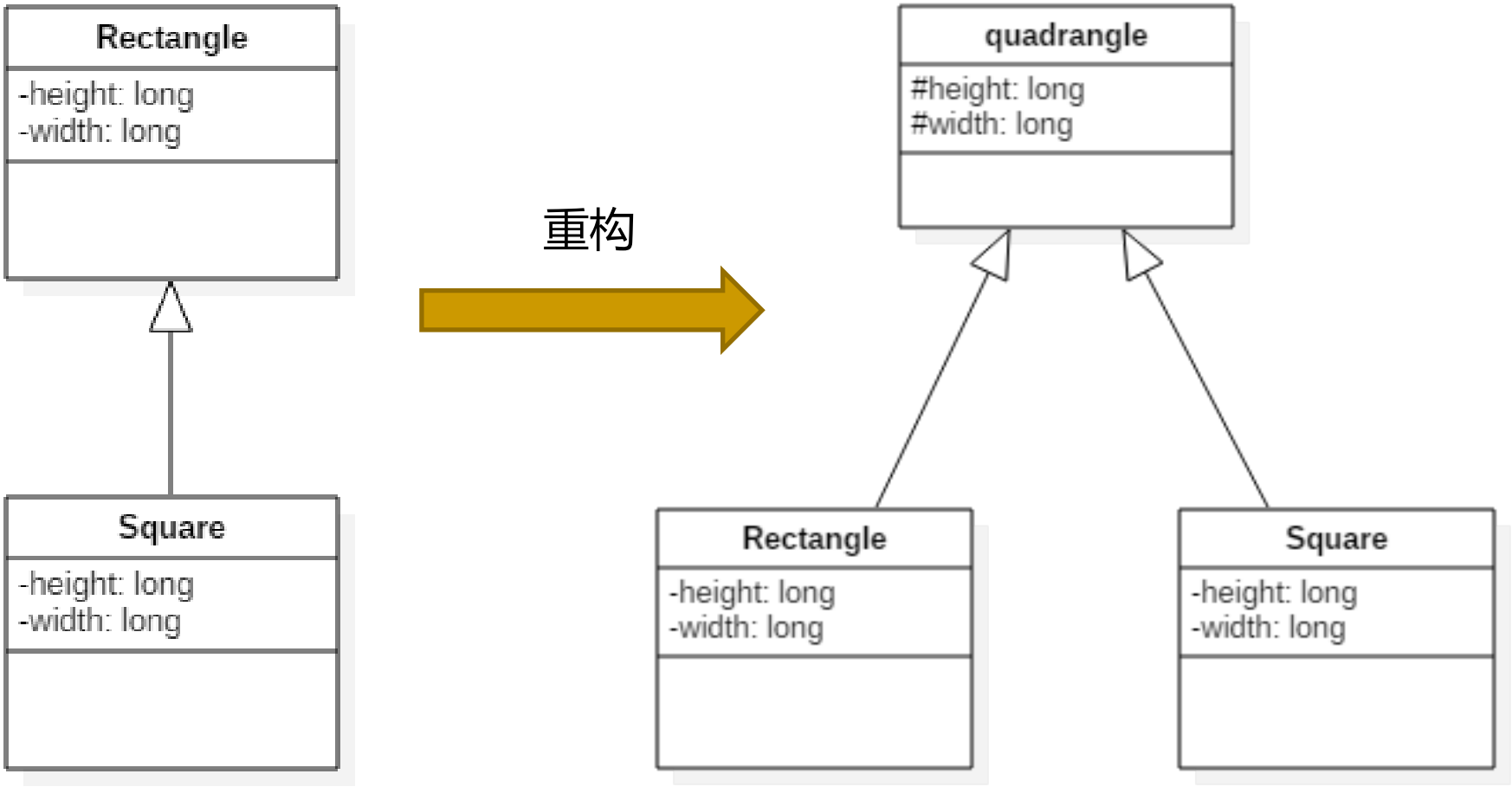
例：长方形和正方形




```
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

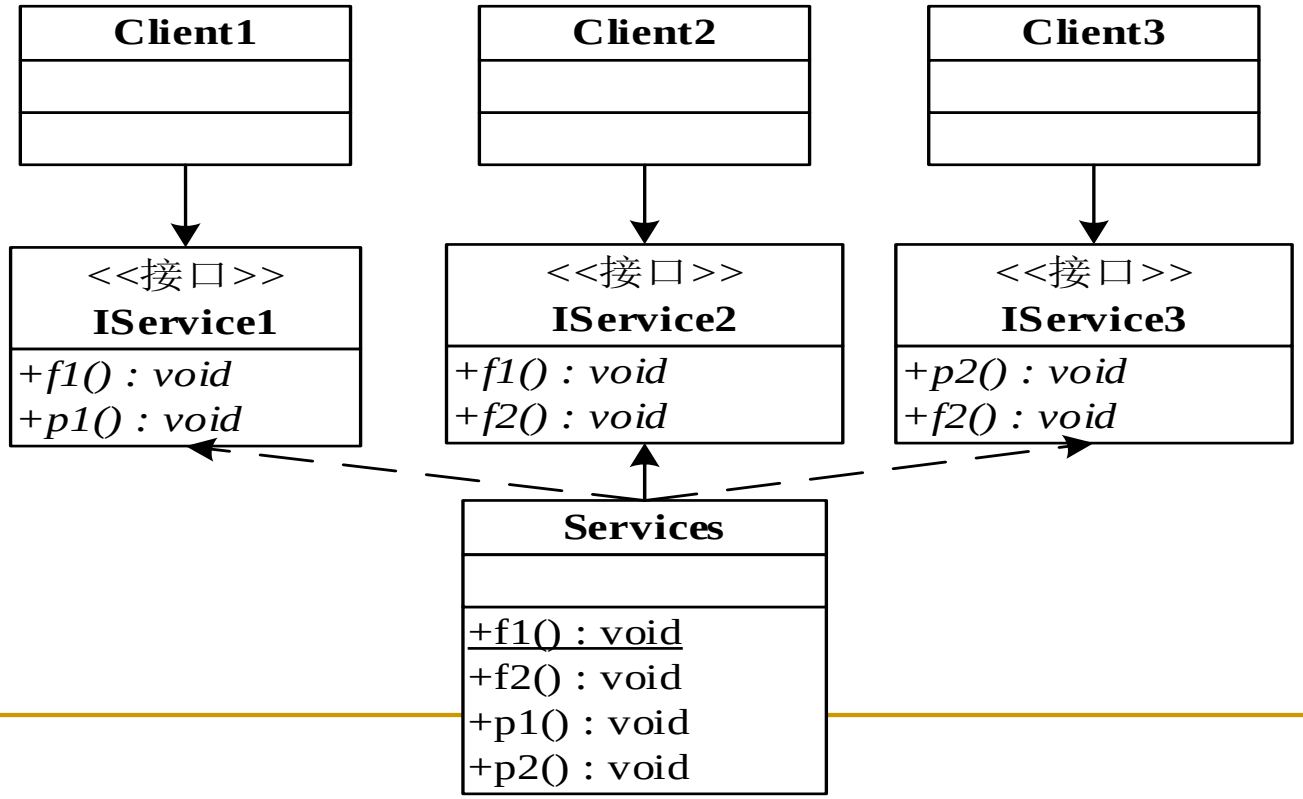
Many client-specific interfaces are better than one general purpose interface.

Robert C. Martin

Design Principles and Design Patterns

接口隔离原则

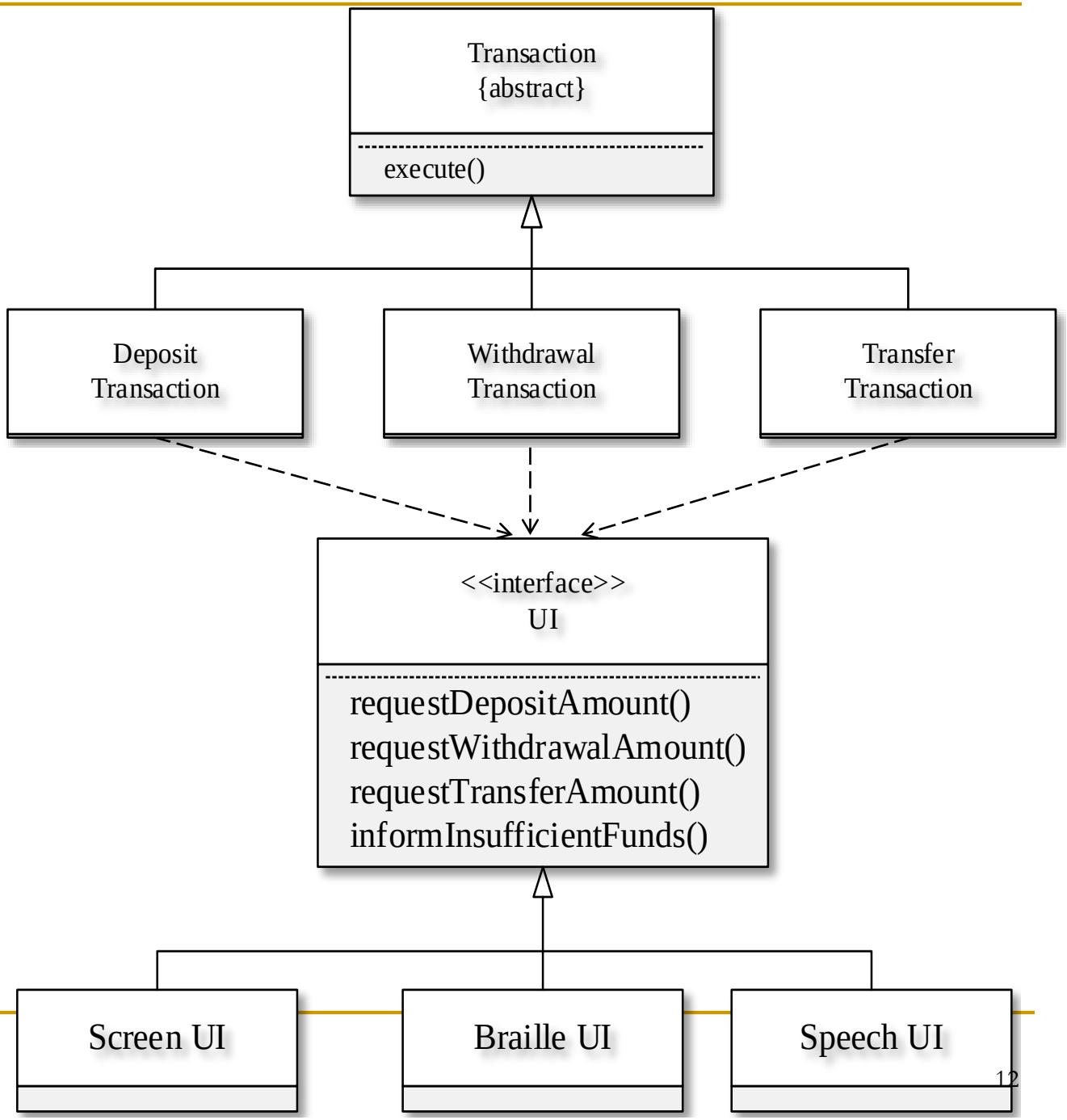
- 如果类的接口定义暴露了过多的行为，则说明这个类的接口定义内聚程度不够好。
- 类的接口可以被分解为多组功能函数的组合，每一组服务于不同的客户类。



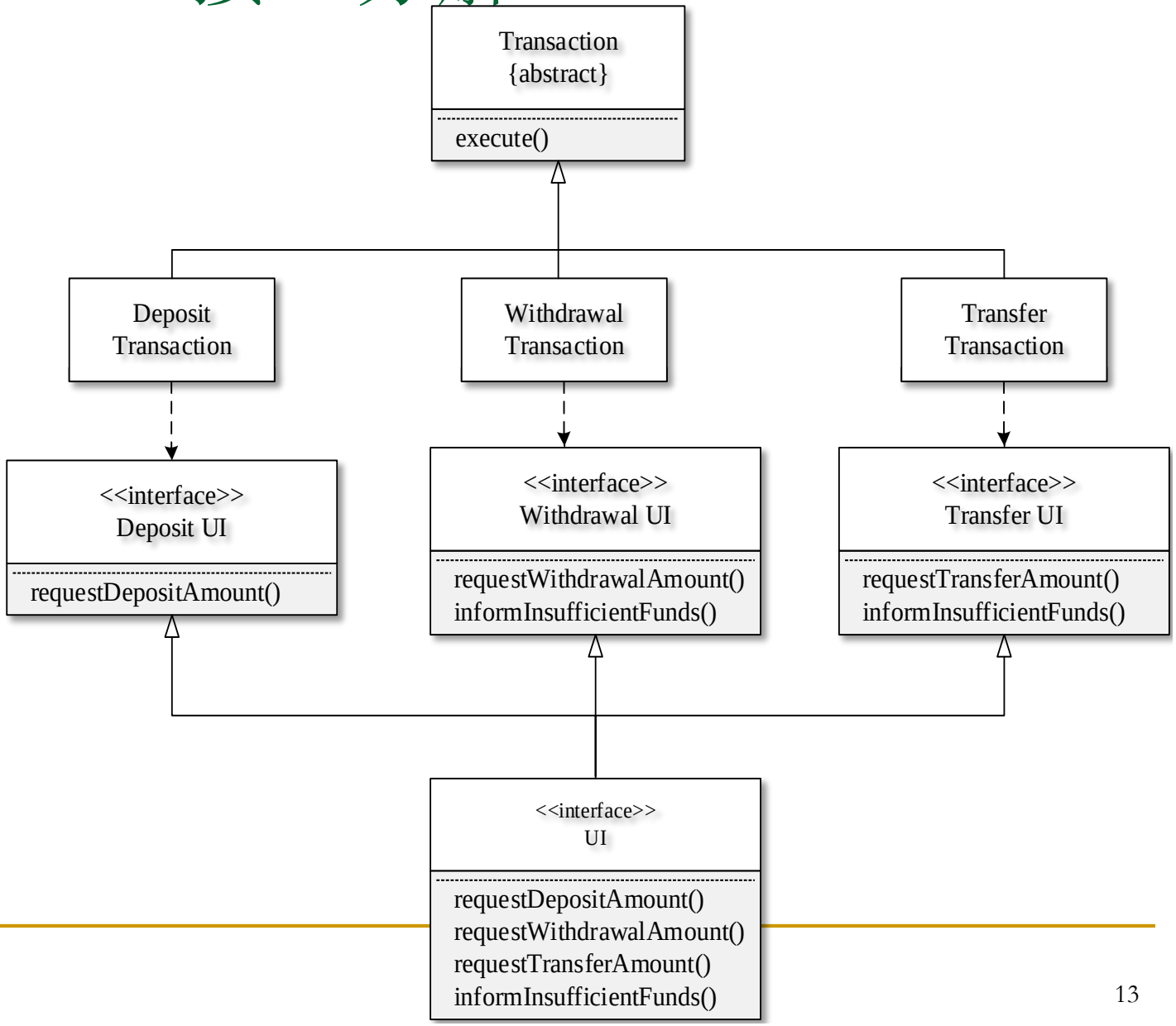
接口污染

- 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。

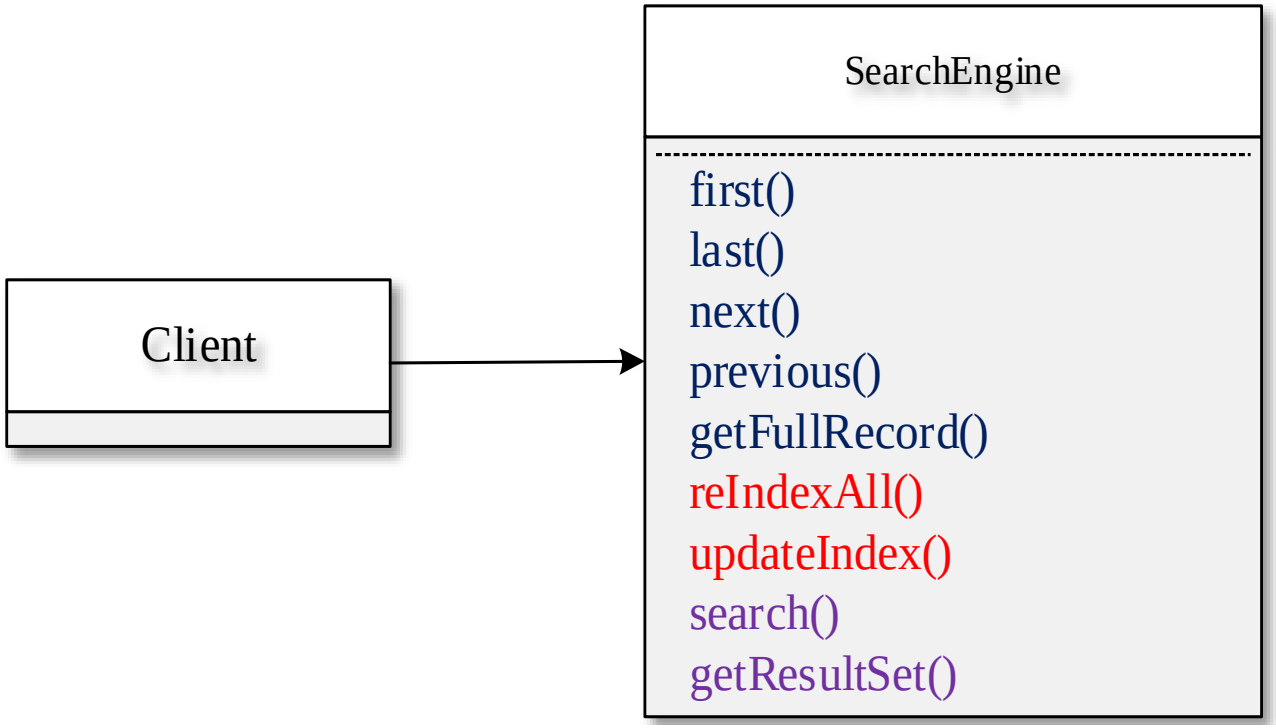
例：ATM


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

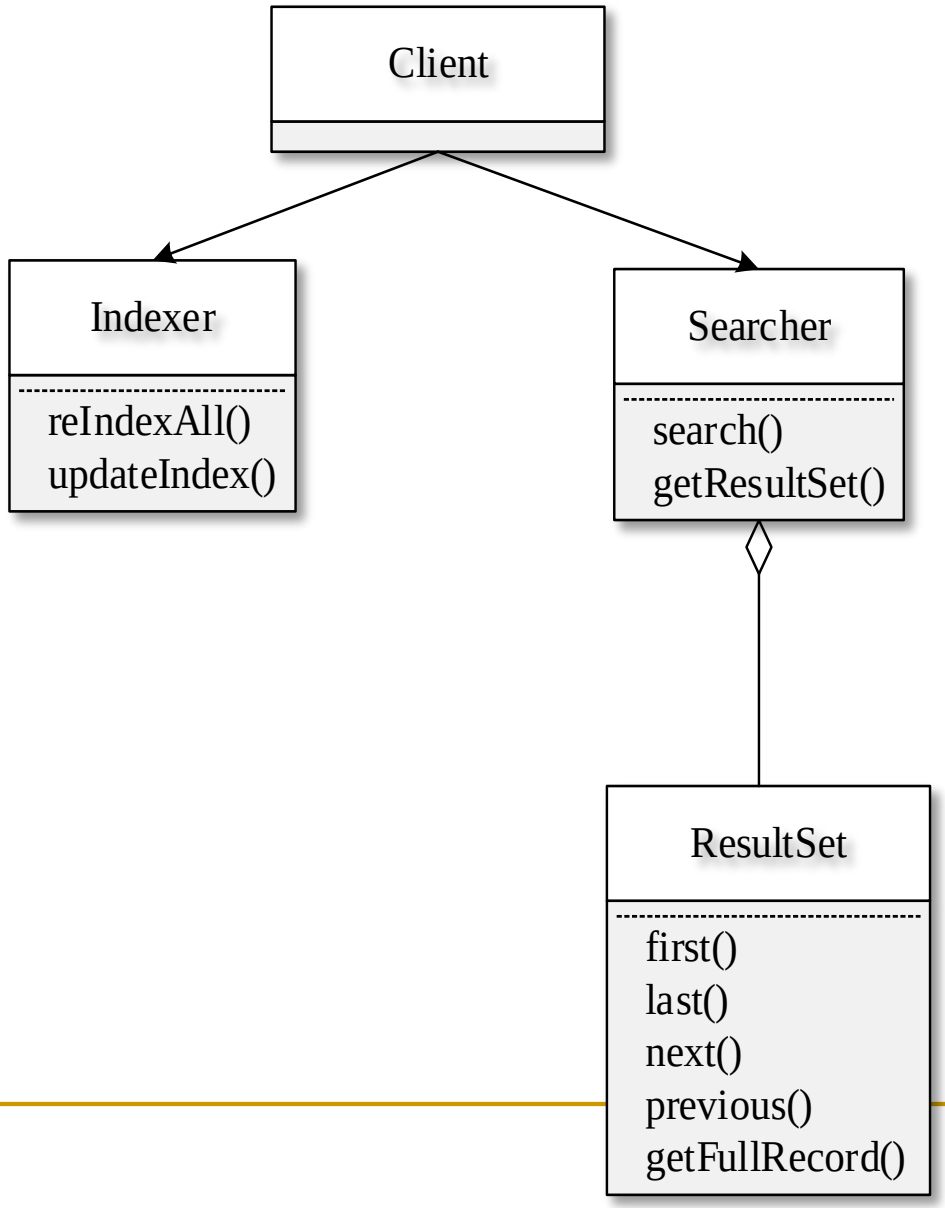

Transaction
{abstract}
execute()
Deposit
Transaction
Withdrawal
Transaction
Transfer
Transaction
<<interface>>
Deposit UI
requestDepositAmount()
<<interface>>
Withdrawal UI
requestWithdrawalAmount()
informInsufficientFunds()
<<interface>>
Transfer UI
requestTransferAmount()
informInsufficientFunds()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

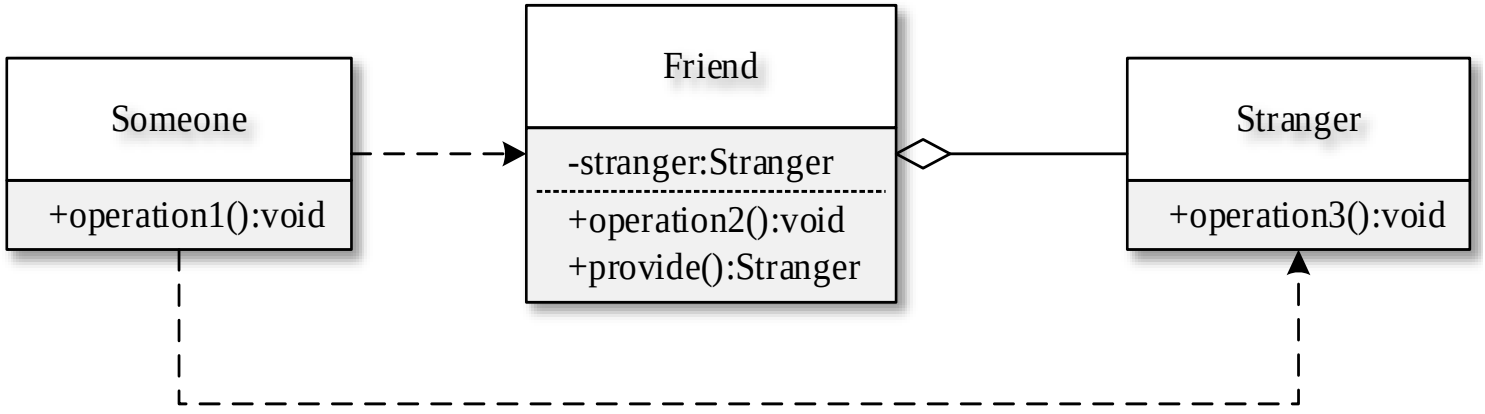
例：全文搜索引擎的系统设计--Good



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() ;
   }
}
```

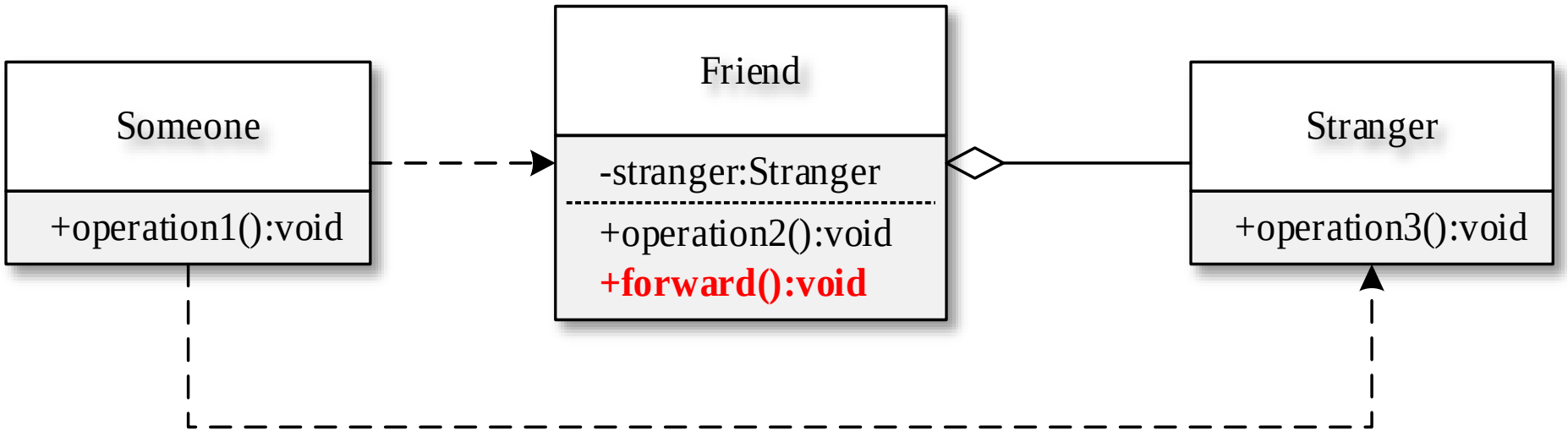
```
public class Friend{
   private Stranger stranger = new Stranger() ;
   public void operation2(){}
   public Stranger provide(){
      return stranger ;
   }
}
```

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）。



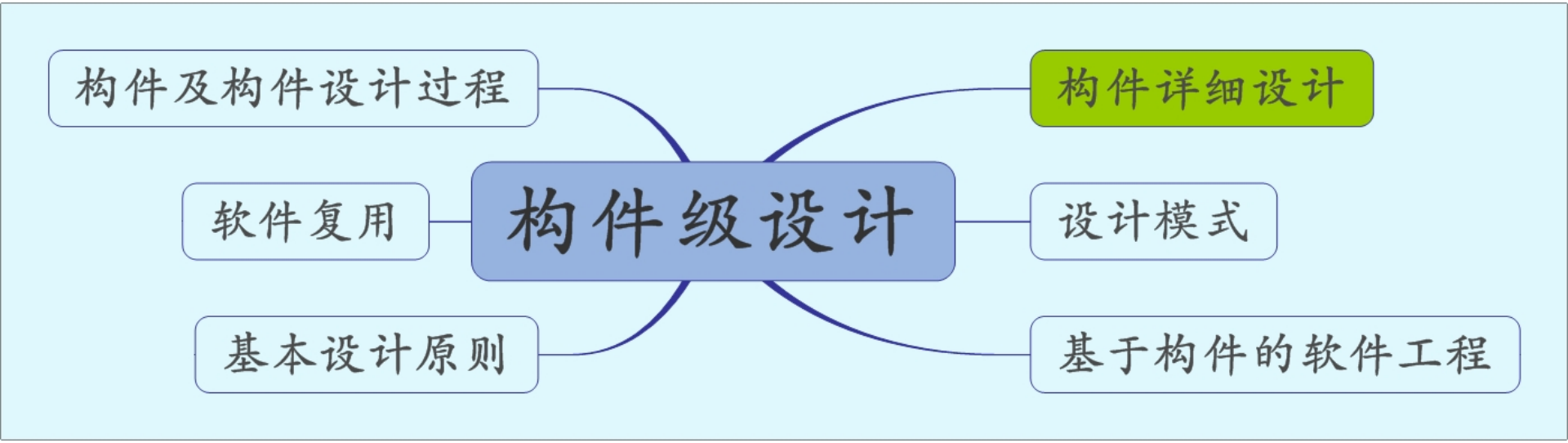
使用迪米特法则进行改造—重构

```
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() ;
    }
 }
```

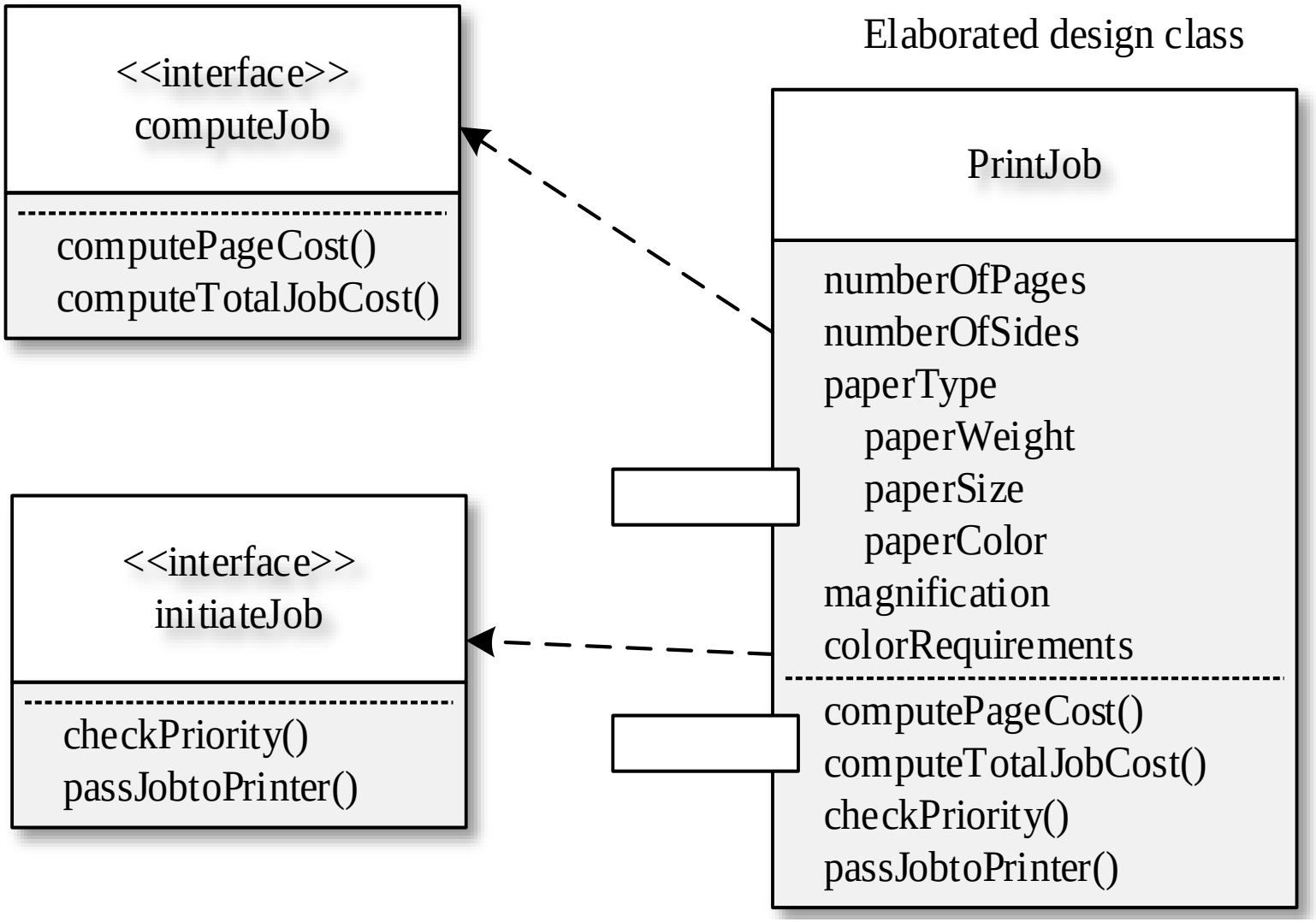
forward()方法是转发方法，调用的细节被隐藏在Friend内部，使Someone与Stranger之间的直接联系被省略掉了，降低了系统内部的耦合度。


构件及构件设计过程
构件详细设计
软件复用
构件级设计
设计模式
基本设计原则
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


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

构件详细设计—属性

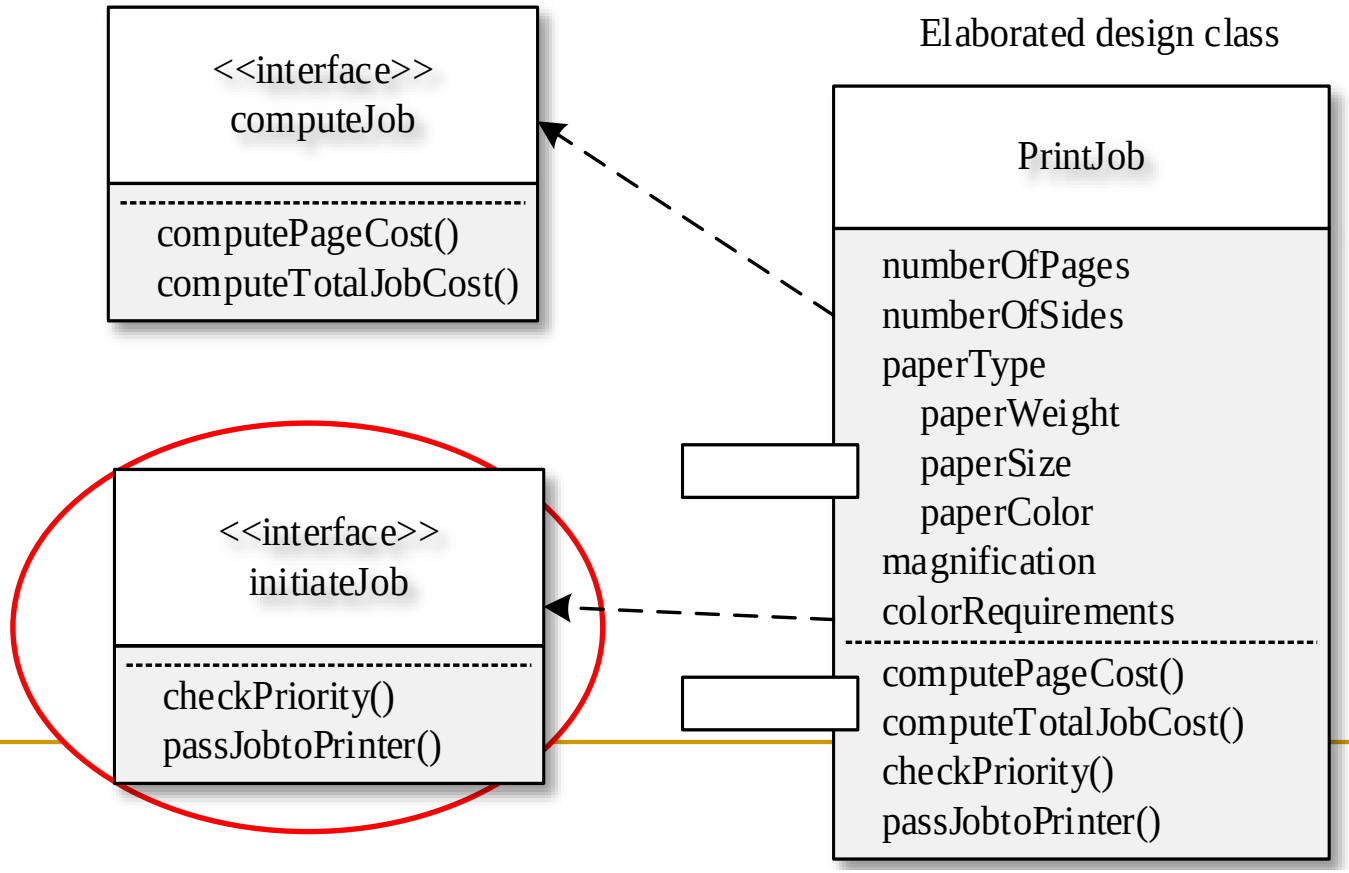
- 定义属性的数据结构和类型
  - 通常情况下，用实现阶段使用的编程语言来定义
  - 现阶段，使用UML属性格式定义属性的数据类型
    - 第一次迭代：属性只有名称
    - 第二次迭代：属性增加数据格式

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

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

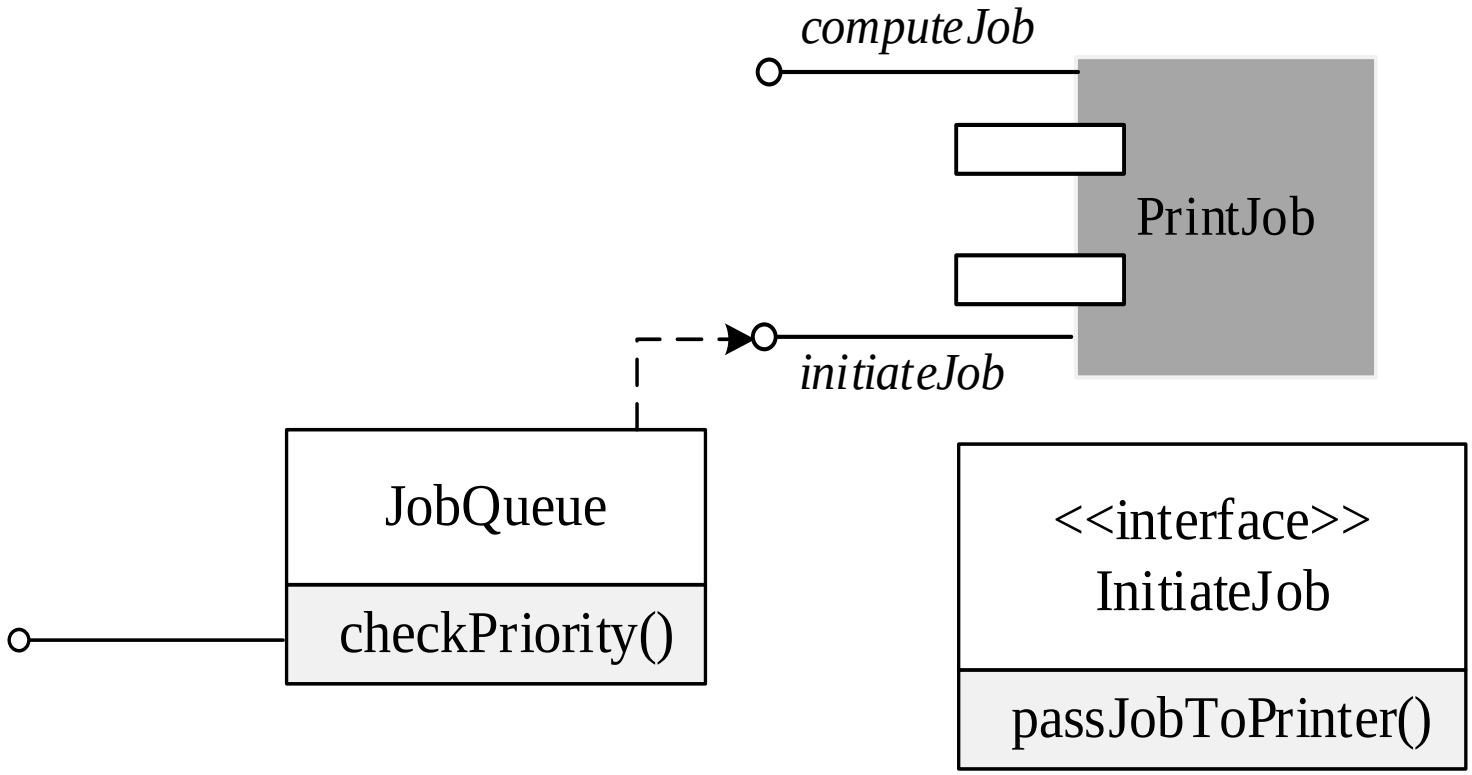
构件详细设计—接口

- 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：

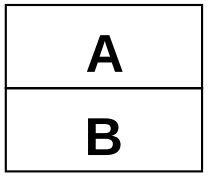
- 图形工具： 将过程细节用图形来表示，在图中，逻辑结构用具体的图形表示。
- 列表工具：利用表来表示过程细节，表列出了各种操作和相应的条件。
- 语言工具：用类语言（伪码）表示过程的细节，很接近编程语言。

程序流程图

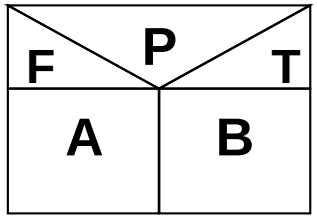
- 程序流程图又称为程序框图，它是历史最悠久、使用最广泛的描述过程设计的方法。
- 它的主要优点是对控制流程的描绘很直观，便于初学者掌握。
- 程序流程图历史悠久，至今仍在广泛使用着。

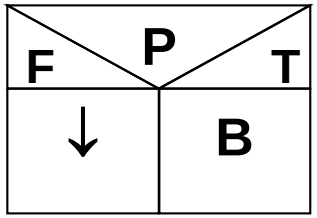
盒图(N-S图)

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。

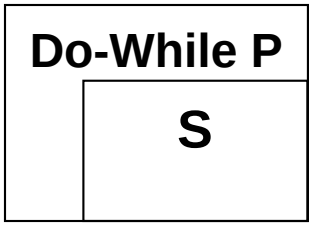


Sequential

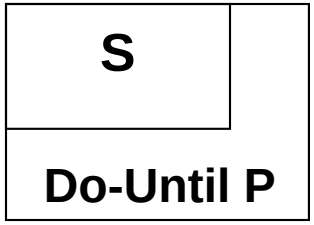




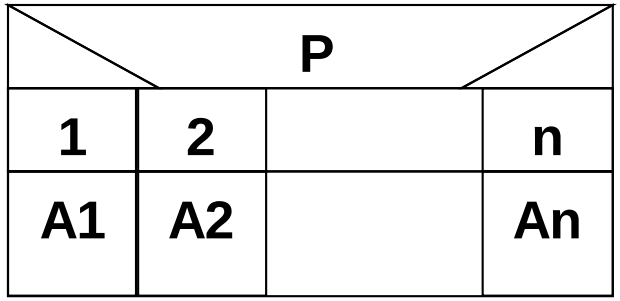
Selective



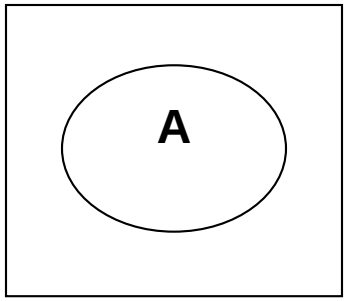
While



Until



Case

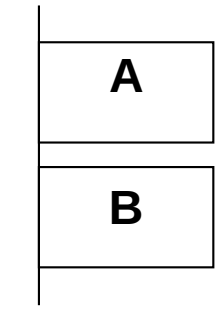


Call subroutine

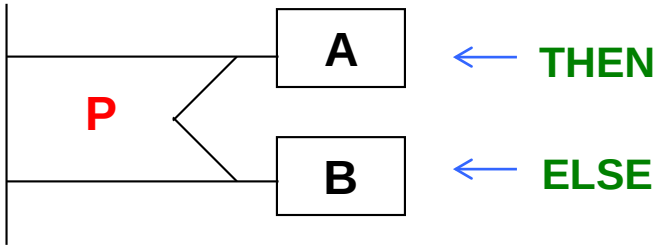
PAD图

- **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。**

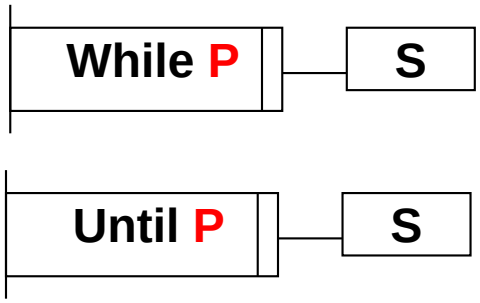
PAD图



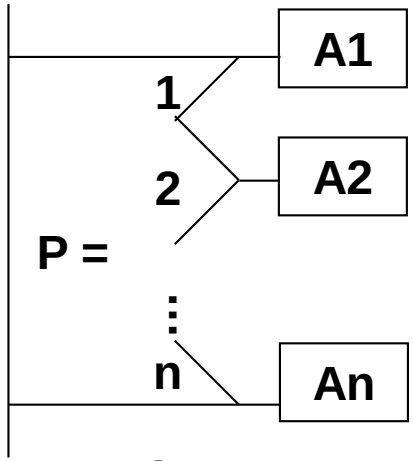
Sequential



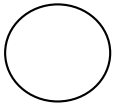
Selective



Loops



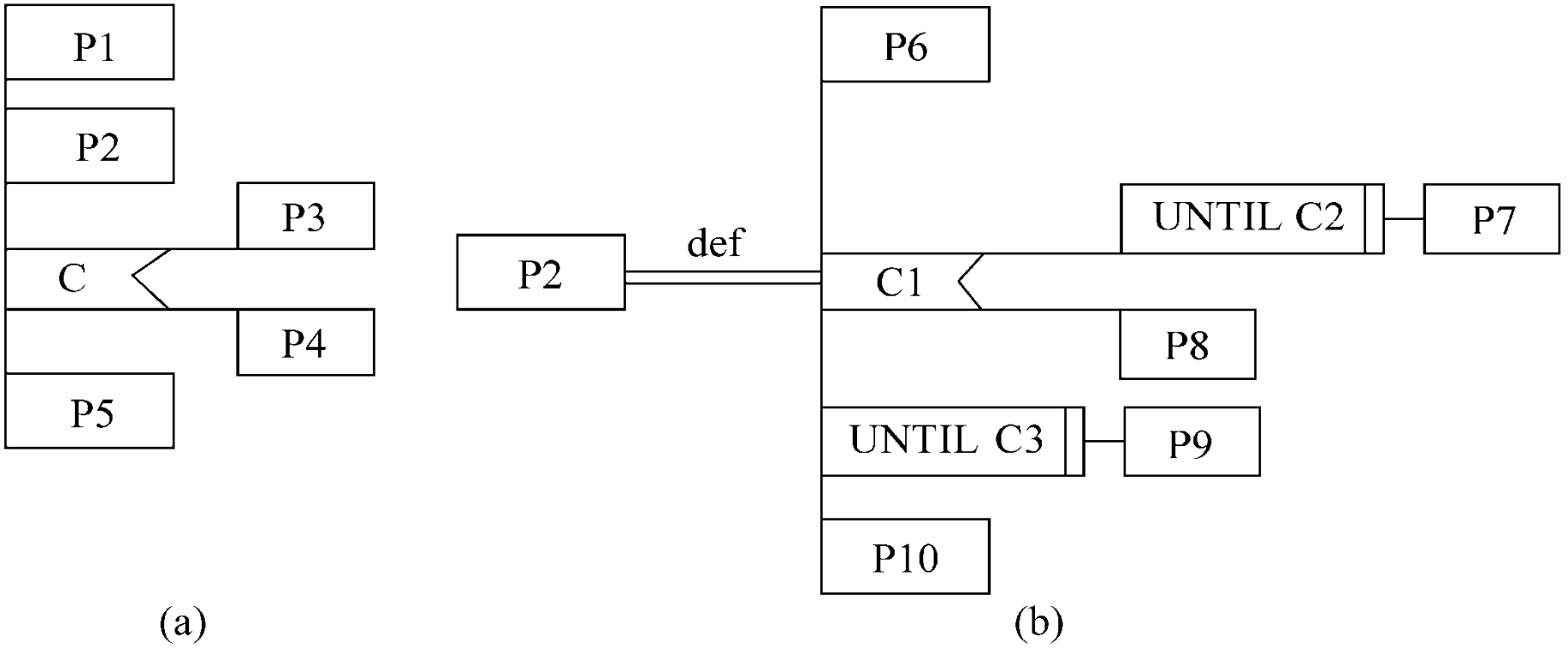
Case



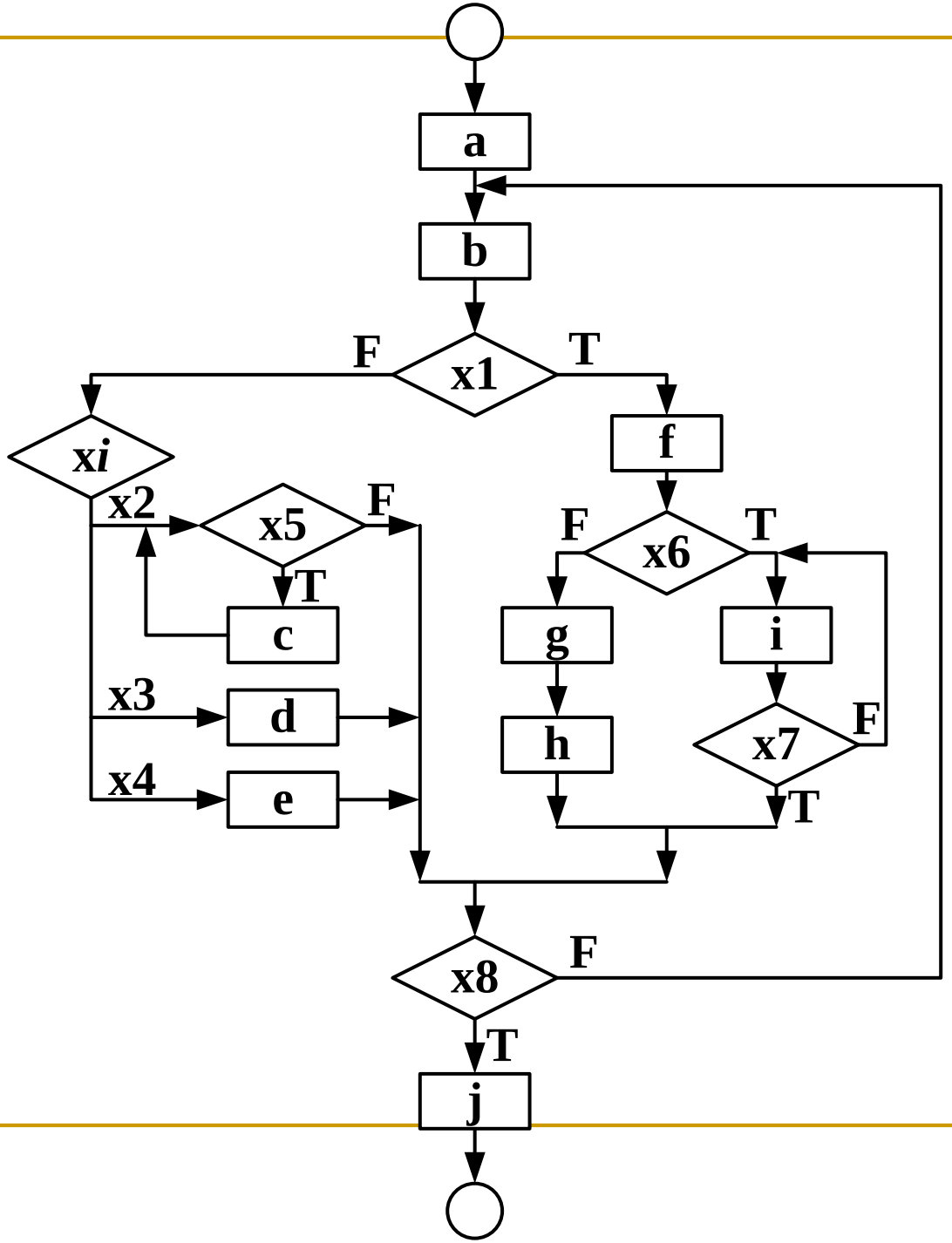
Statement Index

Definition

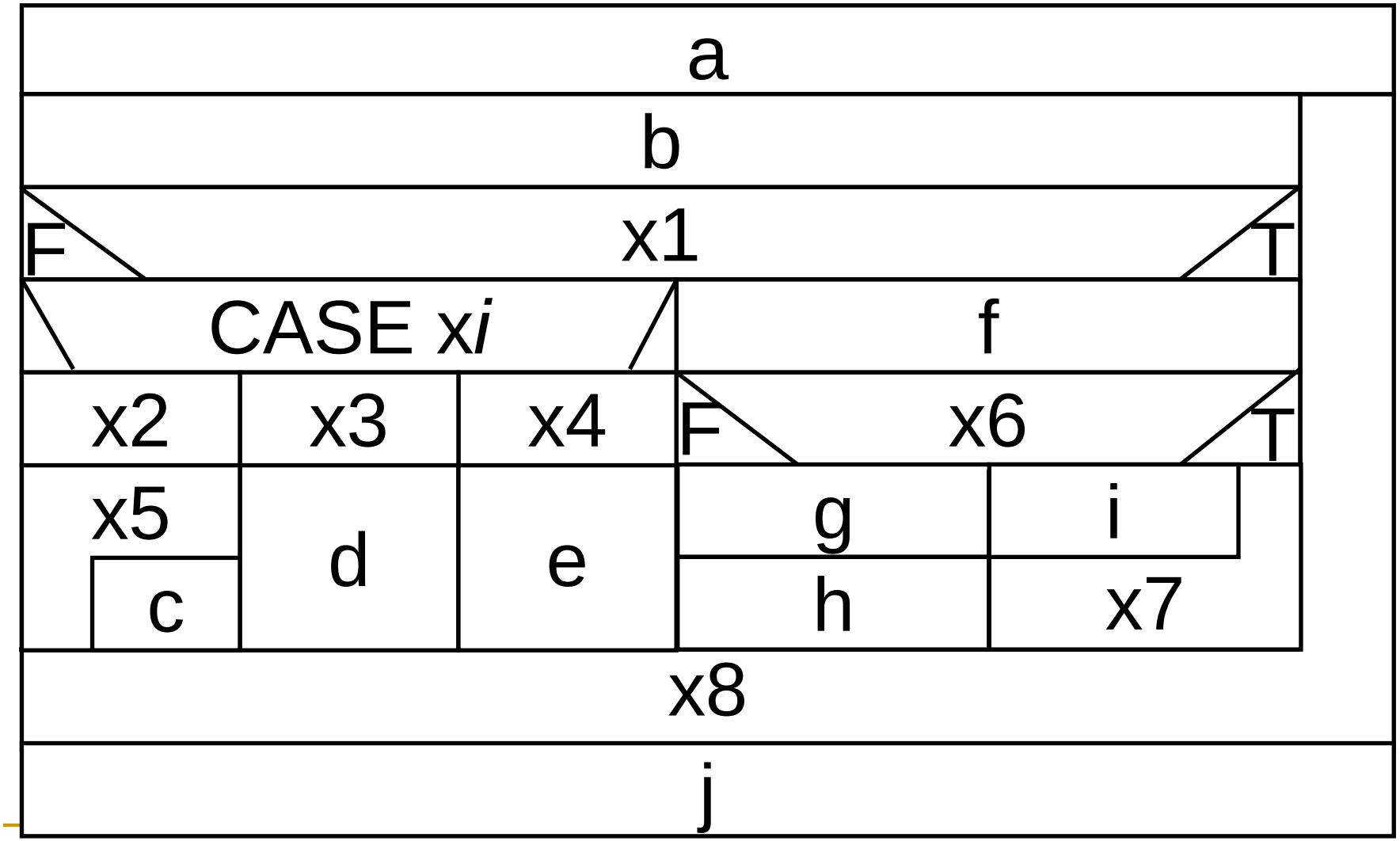
PAD图例



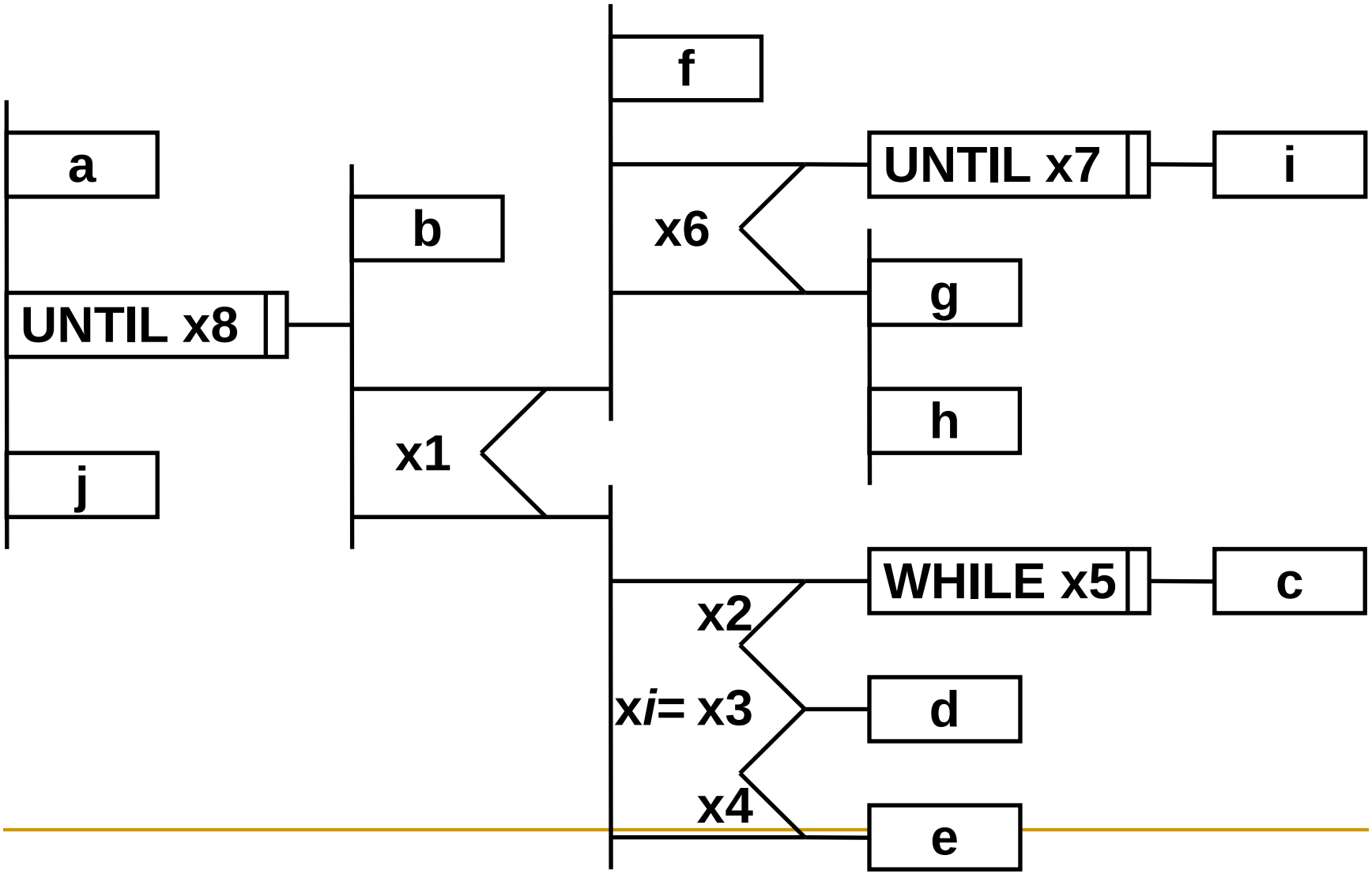
- **例题：**某程序流程图如右图所示，请分别用N-S图和PAD图表示。



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

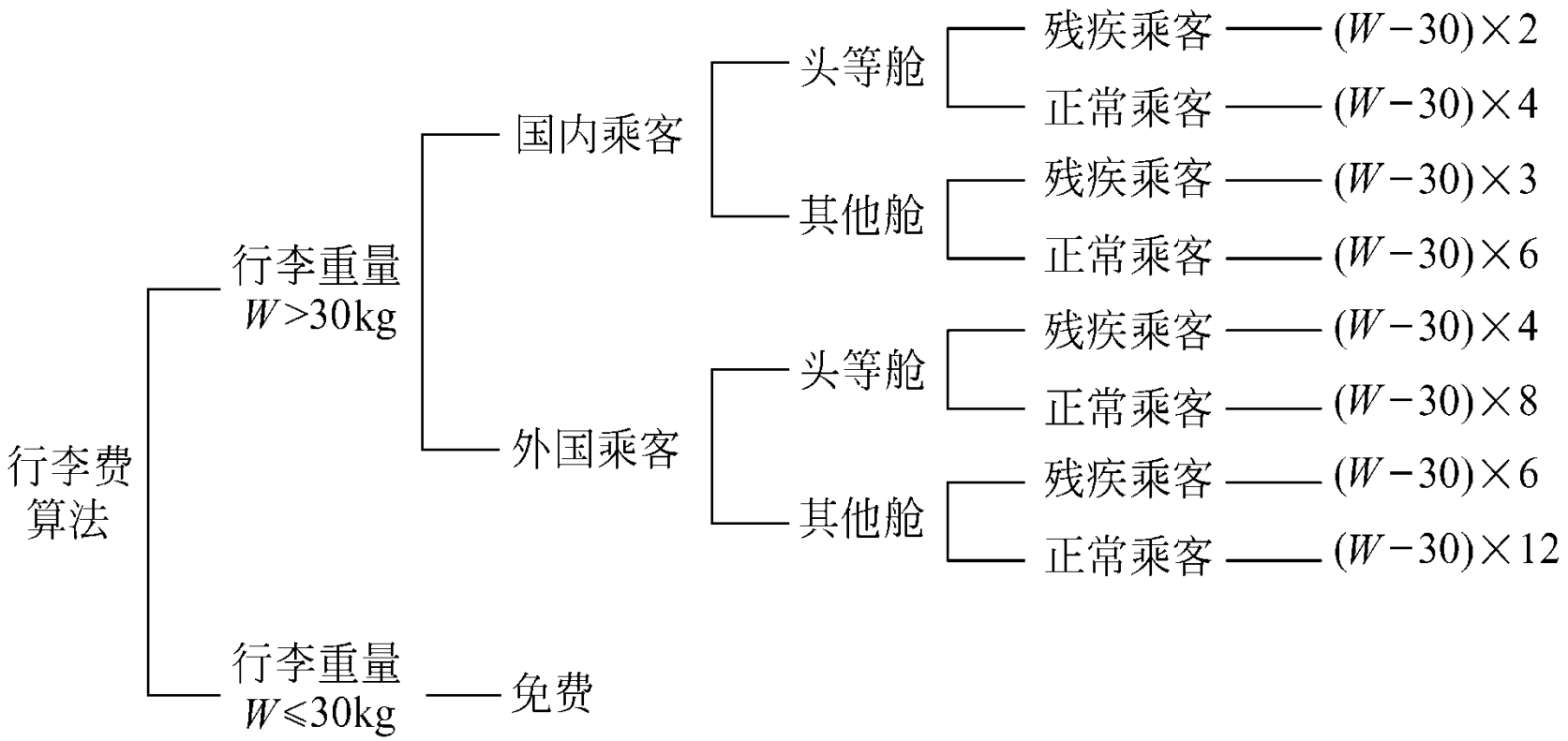
- 假设某航空公司规定，乘客可以免费托运重量不超过30kg的行李。
- 当行李重量超过30kg时，对头等舱的国内乘客超重部分每公斤收费4元，对其他舱的国内乘客超重部分每公斤收费6元。
- 对外国乘客超重部分每公斤收费比国内乘客多一倍，对残疾乘客超重部分每公斤收费比正常乘客少一半。

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

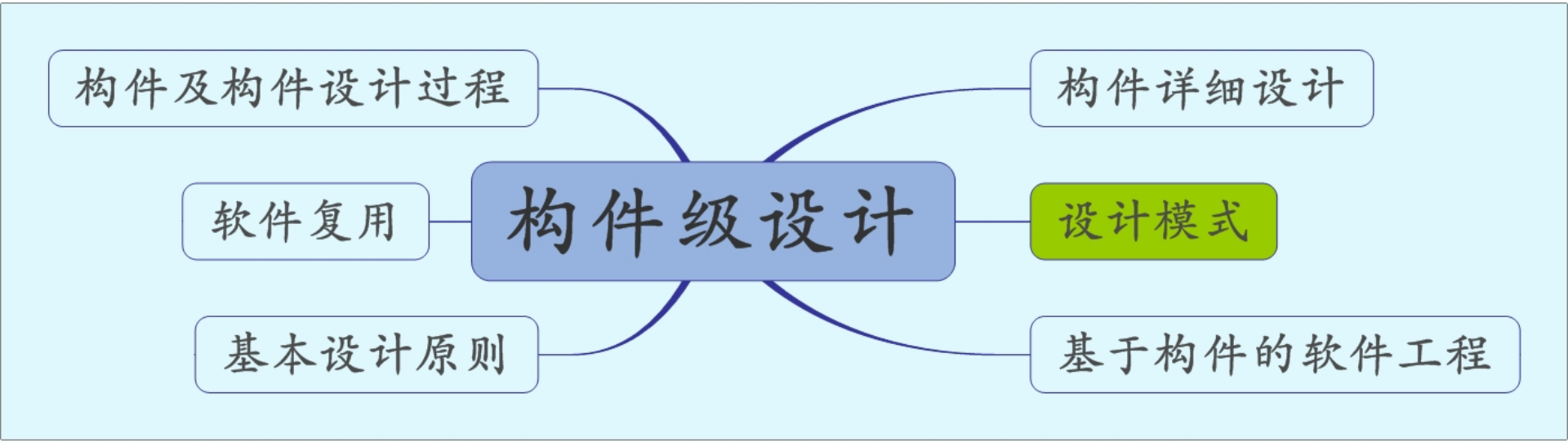
过程设计语言

- 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;
          }
          ……
      }
}
```

outline



概述

- 在设计过程中经常问这样一个问题

  “I wonder if anyone has developed a solution for this?”
- 对于清晰描述的一组问题，基于模式的设计通过查找一组已被证明有效的解决方案来创建新的应用系统。
- 起源于20世纪70年代
  - Alexander
  - 建筑设计模式 ：建筑设计中存在一定的共性模式

设计模式定义

- Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

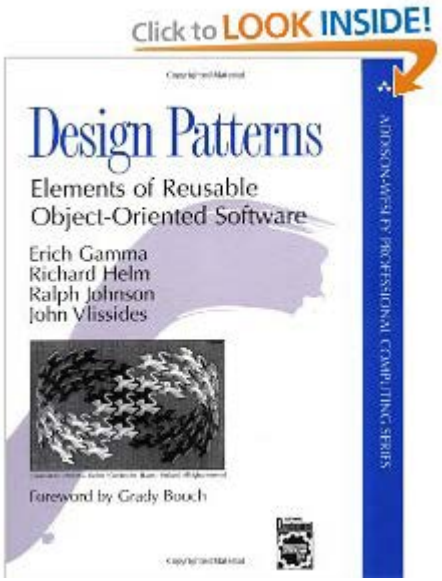
  -----Chirstopher Alexander

- “每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心。这样，你就能一次又一次地使用该方案而不必做重复劳动”

软件设计中的模式

- Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides

  《Design Patterns—Elements of Reusable Software》
- Gang of Four (GOF) 23
- 设计模式已经成为一种谈论设计的词汇表



设计模式的分类

- 创建型模式（Creational Pattern）
  - 有关对象创建的模式：5
- 结构型模式（Structural Pattern）
  - 描述对象构造和组成的方式:7
- 行为型模式（Behavioral Pattern）
  - 描述一组对象交互的方式：11

设计模式的4个基本元素

- **一个名字**，作为对模式的有意义的参照
- **一个问题域的描述**，解释该模式何时适用
- **一个对设计解决方案的描述**，给出设计解决方案的模板，包括方案的各个部分、它们之间的关系、以及它们的职责
- **一个对效果的陈述**，应用该模式的结果以及权衡

观察者（Observer）模式

模式名称：观察者

问题描述：在很多情况下必须将一个对象的状态呈现与对象本身分离开，并为状态信息提供多种呈现方式。在状态发生变化时，所有的呈现都会自动得到通知并进行更新。

这个模式的使用情形是，状态信息需要不止一种呈现方式，并且维护状态信息的对象不需要知道所使用的特定的呈现格式。

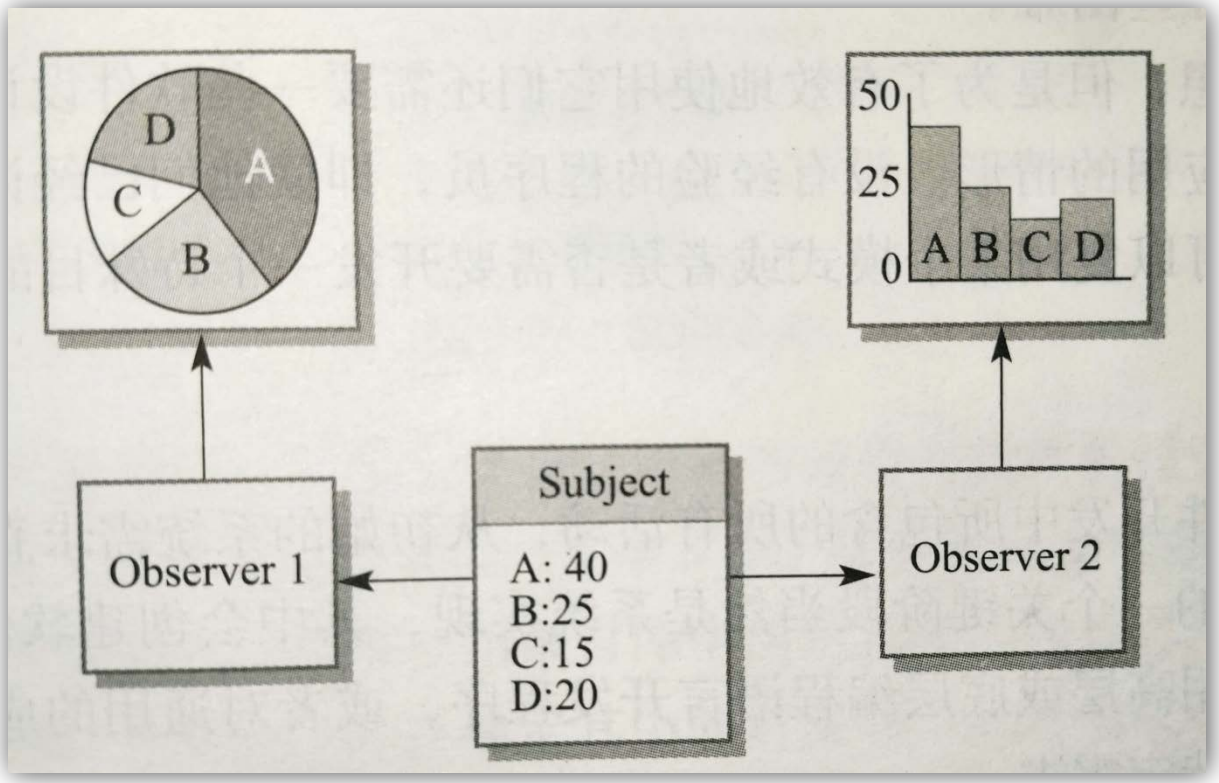
解决方案描述：解决方案包括两个抽象对象Subject(主题，即被观察者)和Observer(观察者)，以及两个继承了相关抽象对象的具体对象ConcreteSubject和ConcreteObserver。抽象对象包括适用于所有情形的通用操作。需要呈现的状态在ConcreteSubject中维护，该对象继承了Subject的操作从而允许该对象增加和移除观察者（每个观察者对应一种呈现方式）以及在状态发生变化时发出通知。

ConcreteObserver维护了ConcreteSubject状态的一份拷贝，并且实现了Observer的Update()接口，该接口允许这些拷贝能够被同步更新。ConcreteObserver自动呈现状态并在任何时候当状态更新时自动反映变化。

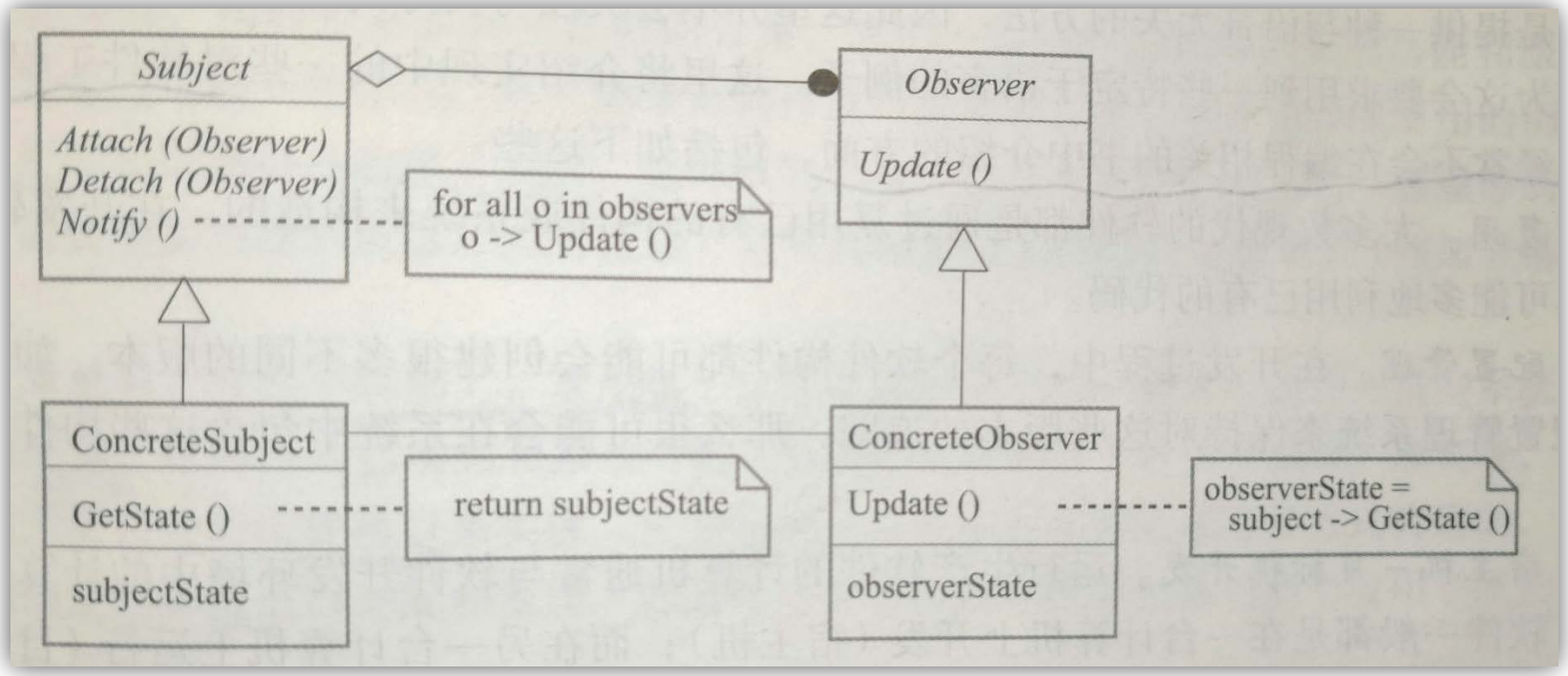
UML模型图

效果：主题对象只知道抽象的观察者对象，而不知道具体类的细节，因此，这些对象之间的耦合被最小化了。

多种呈现方式

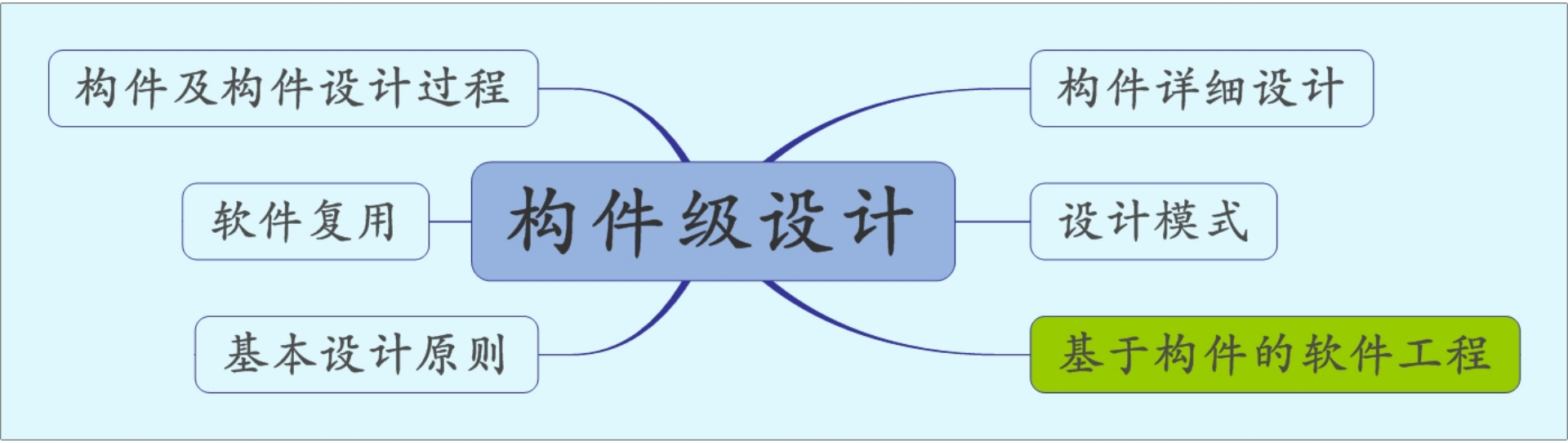

D
A
C
B
50
25
0
A B C D
Subject
A: 40
B:25
C:15
D:20
Observer 1
Observer 2

观察者模式的UML模型



几点建议

- 为了在设计中使用模式，需要认识到你所面对的任何设计问题都可能会有一个与之相关联、可以应用的模式
  - 向多个对象告知一些其他对象的状态发生了变化 **观察者模式**
  - 为一个集合中的元素提供一种标准的访问方式，不用考虑集合是如何实现的 **迭代器模式**
  - 允许在运行时扩展一个已有的类的功能 **装饰者模式**
- 设计模式非常有用，但是为了更有效地使用还需要一些软件设计经验


构件及构件设计过程
软件复用
基本设计原则
构件级设计
构件详细设计
设计模式
基于构件的软件工程

Component-based software engineering

- CBSE作为一种基于复用软件构件的软件系统开发方法，是在20世纪90年代末期出现的
- CBSE现在是主流的软件工程方法
- CBSE中构件接口与构件实现应该明确的分离开来
- 构件标准使构件集成变得更容易，用不同语言编写的构件可集成在同一个系统中使用
- 为了使独立的、分布的构件一起工作，需要有处理构件之间通信的**中间件**

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
    - 构件组合（composition）将合格的、适合的构件组合到体系结构中

构件标准

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

- **问题**
  - 参与提出标准的公司不能就构件的单一标准达成一致，从而限制了这种方法对软件复用的影响
  - 针对不同平台开发的构件不能互操作
  - 已有的这些标准和协议非常复杂，很难理解

面向服务的软件工程

- 针对这些问题，发展了“**构件即服务**”的概念，并且提出标准来支持**面向服务的软件工程**
- Web服务是一个松耦合、可复用的软件构件，封装了离散的功能，该功能是分布式的并且可以被程序访问。Web服务是通过标准互联网和基于XML的协议被访问的服务
- Web服务没有“请求”接口
- Web服务接口是定义服务功能和参数的简单的“提供”接口
- Web服务由“重量级”向“轻量级”发展