第九章 构件级设计建模

王美红

主要内容

- 什么是构件
- 设计基于类的构件
- 内聚性
- 耦合性
- 实施构件级设计
- 对象约束语言
- 设计传统构件

概述

- 程序体系结构设计类似于设计一栋房子的全貌，有多少件屋子，每间屋子之间的连接等等。
- 构件级设计类似于设计房子的每个较大的组成部件的内部结构，比如卧室该怎样布置，卫生间该如何安排等等
- 构件级设计比体系结构设计更细，但还细不到编码的程度，比如，设计卧室里边用拼木地板，但不会强调该地板是横着，还是竖着安，这些是具体编码时要考虑的事情。

概述（续）

- 目的：避免高层次的抽象模型向低层次的程序之间转换时容易引入错误的问题。
- 构件级设计建模时机：在体系结构设计第一次迭代后完成

概述（续）

- 构件级设计形式：
  - 用编程语言表示
  - 用能够容易转化为代码的中间表示(如图形的、表格的或基于文本的)

9.1 什么是构件

- 通俗地讲，构件是一段程序，该程序能完成一个相对独立的功能，并有一定的通用性。
- 正式定义：系统中某一定型化的、可配置的和可替换的部件，该部件封装并暴露一系列接口。

9.1 什么是构件（续）

- 针对不同的系统设计体系，构件所指的对象不一样。

9.1.1 面向对象观点

- 在面向对象的设计中，构件指一个协作类的集合。
- 一般来讲，构件的规模比类大，但有时一个构件也可以对应一个类。
- 在构件级设计时，应设计出类的所有属性以及和其它类之间的相关操作，通信接口必须明确定义。

印刷任务构件设计

细化：

- 属性（数据结构）
- 操作（算法细节）
- 接口（接口中可能隐含与其它的协作）

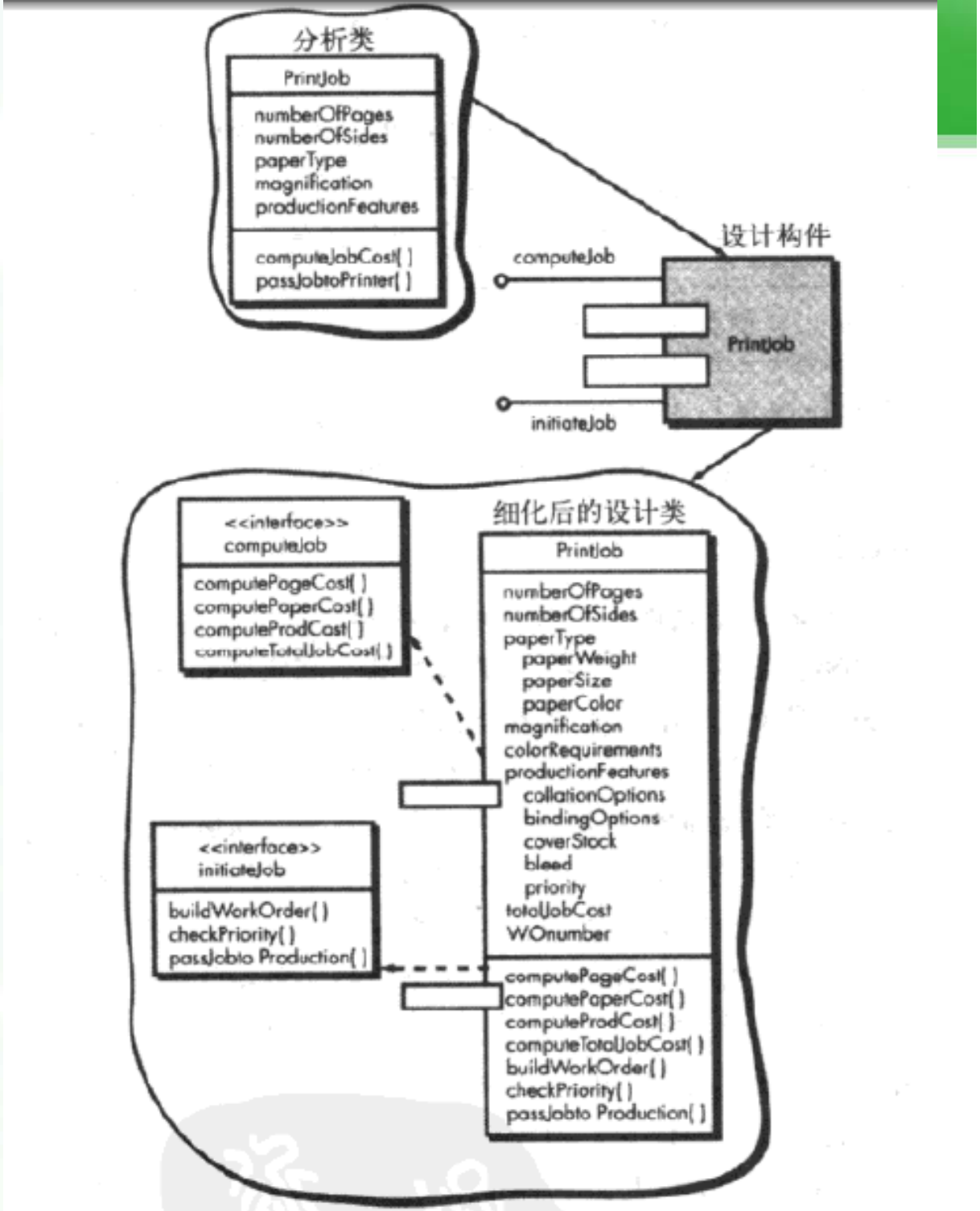


图　　设计构件的细化

9.1.2 传统观点

- 传统构件也称为模块
  - 程序的一个功能要素，程序由处理逻辑及实现处理逻辑所需的内部数据结构以及能够保证构件被调用和实现数据传递的接口构成。

9.1.2 传统观点(续)

- 构件（模块）的分类：
  - 控制构件-----协调不同模块之间的调用
  - 问题域构件------完成部分或全部用户的需求
  - 基础设施构件-----负责完成问题域中的相关处理的功能

模块的导出是以数据流图的方式进行的
控制构件位于顶层，问题域构件位于低层

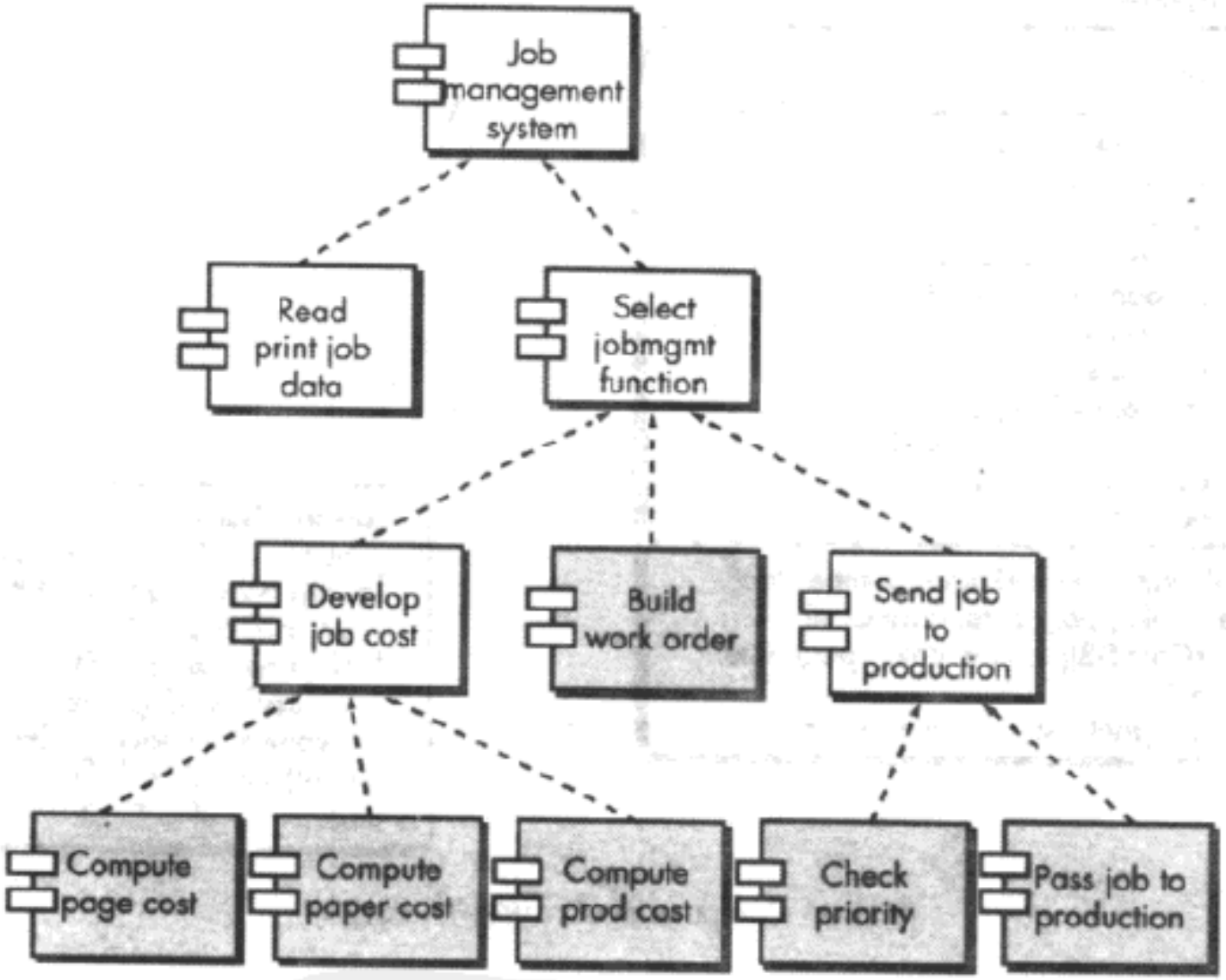


图　一个传统系统的结构图

每个模块都要被细化

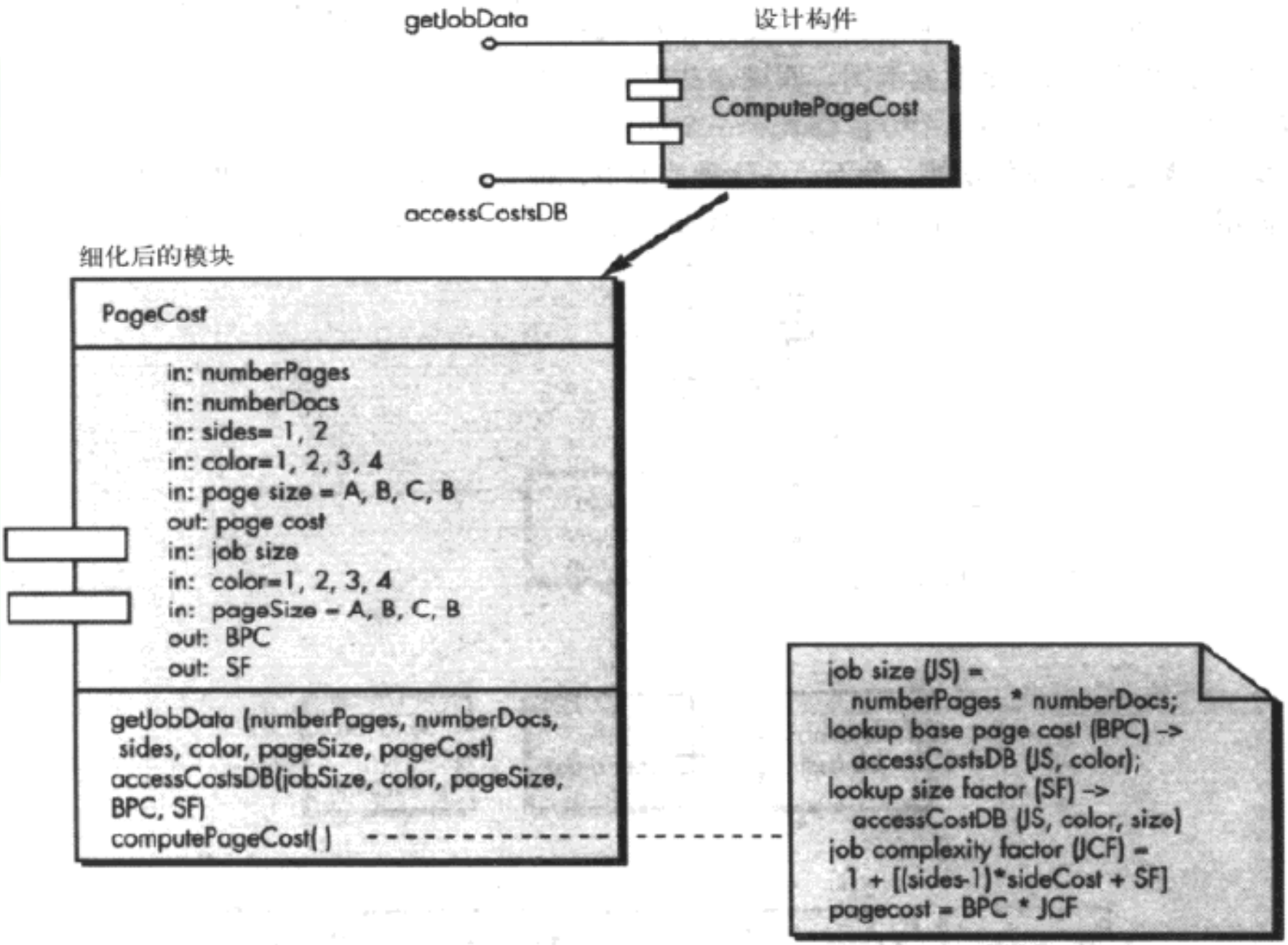


图　　ComputePageCost的构件级设计

9.1.3 过程相关的观点

- 前面两种都是假设从头开始设计构件
- 在实际工作中，往往借鉴他人或自己前期的工作成果，直接引用或做一定的改进后引用已有的一些构件，基本原理是所从事的项目之间有内在的联系或相关性。

9.2 设计基于类的构件

- 当选择了面向对象软件工程方法后，构件级设计主要关注分析类的细化和基础类的定义和精化

9.2.1 基本设计原则

- 开关原则（The Open-Closed Principle ，OCP）
  - 模块应该对外延有开放性，对修改具有封闭性。
  - 即设计者应该采用一种无需对构件自身内部（代码或者内部逻辑）做修改就可以进行扩展（在构件所确定的功能域内）的方式来说明构件。

- SafeHome针对不同的传感器，接口都向Dector构件呈现一致的视图。

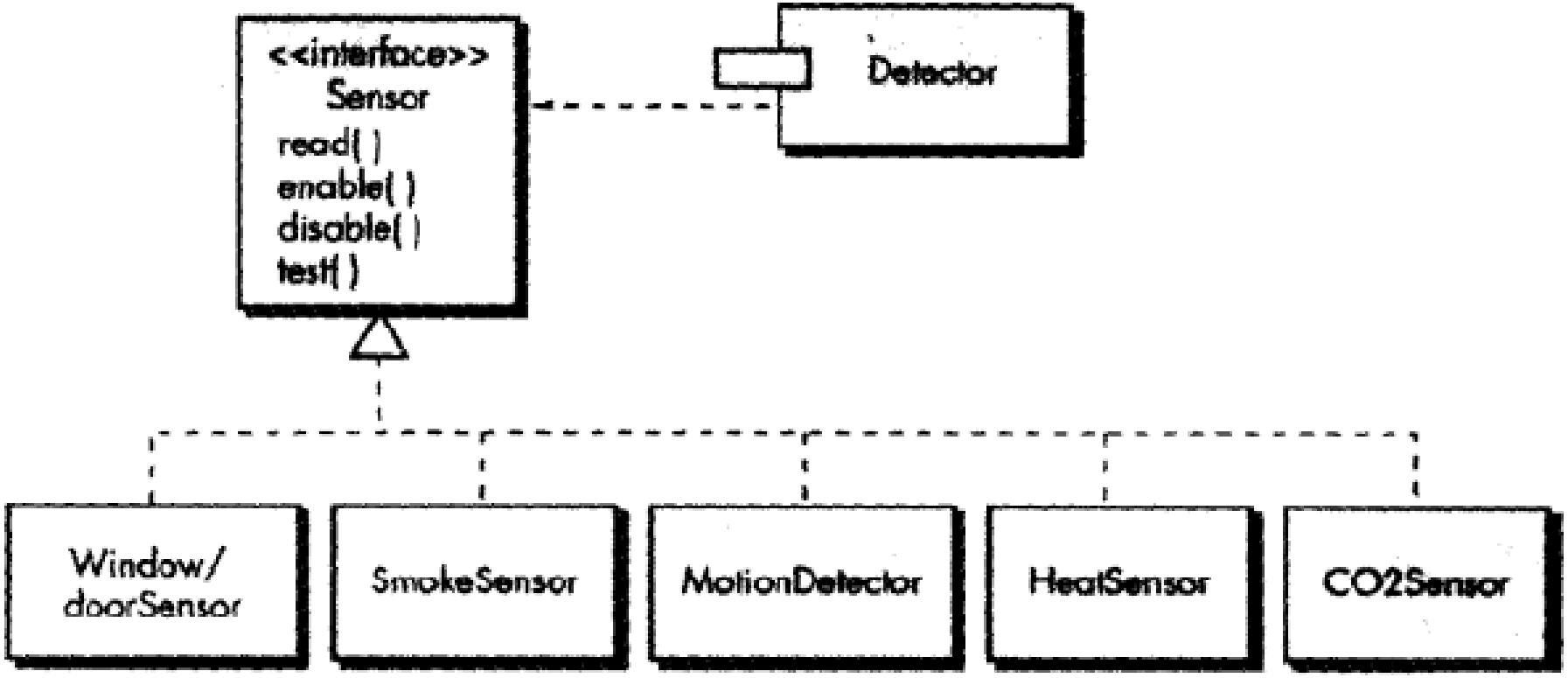


图　　遵循OCP原则

9.2.1 基本设计原则（续）

- Liskov替换原则（Liskov Subsitution Principle, LSP）
  - 子类可以替换它们的基类
  - 源自基类的任何子类必须遵守基类与使用该基类的构件之间的隐含约定（前置条件、后置条件）。

9.2.1 基本设计原则（续）

- 依赖倒置原则（Dependency inversion principle, DIP）
  - 依赖于抽象，而非具体实现
  - 构件依赖的其它构件（不是依赖抽象类，如接口）愈多，扩展起来就愈困难
  - 抽象可以比较容易地对设计进行扩展

9.2.1 基本设计原则（续）

- 接口分离原则(Interface Segregation principle, ISP)
  - 多个用户专用接口比一个通用接口要好
  - 设计者应该为每一个主要的客户类型都设计一个特定的接口。
  - 如SafeHome中FloorPlan类用于安全和监督功能，两处操作有些不同，监督功能多关于摄像头的操作，定义两个接口。

9.2.1 基本设计原则（续）

- 将多个构件组织起来的原则：
  - 发布复用等价性原则---对类打包管理，同时升级。
  - 共同封装原则----一同变更的类应该和在一起。
  - 共同复用原则----可能一起被复用的类才能打包到一块。

9.2.2 构件级设计指导方针

- 一些实用的设计指导方针可以应用于构件、构件的接口，以及对于最终设计有着重要影响的依赖和集成特征等方面。

9.2.2 构件级设计指导方针（续）

- 构件
  - 构件的名称来源于问题域，应能体现该构件的主要功能，并且为所有共利益者理解。
  - 使用构造型帮助识别构件的特性
    - 《infrastructure》，《table》，《database》

9.2.2 构件级设计指导方针（续）

- 接口
  - 表示必须的接口（棒棒糖标记），放在构件框的左边。
- 依赖与继承
  - 为提高可读性，依赖自左向右，继承自下（导出类）而上（基类）；构件的依赖关系通过接口表示，而非“构件到构件”。

9.2.3 内聚性

- 功能内聚—一个模块完成一种且只一种运算并返回结果时发生这个级别上的内聚
- 分层内聚—由包、构件和类来实现，高层能访问低层，但底层不能访问高层
- 通信内聚—访问相同数据的所有操作被定义在一个类中
- 顺序内聚—将构件或者操作按照前者为后者提供输入的方式组合

9.2.3 内聚性（续）

- 过程内聚—构件或者操作的组合方式是，允许在调用前面的构件或操作之前，马上调用后面的构件或操作，及时没有数据传递
- 暂时内聚—操作的执行是为了反映某一特定的行为或状态。
- 实时内聚—在一类中，但是在其他方面不相关的构件、类或操作被分为一组。

9.2.4 耦合性

- 内容耦合—当一个构件“暗中修改其他构件内部数据”时
- 共用耦合—大量构件使用同一全局变量时
- 控制耦合—操作A调用操作B，且传递控制标记
- 印记耦合—类B被声明为类A某一操作中的一个参数中的一个参数类型时

9.2.4 耦合性（续）

- 数据耦合—当操作需要传递较长的数据参数时
- 例程调用耦合—当一个操作调用另一个操作时
- 类型使用耦合—构件A使用在构件B中定义的一个数据类型时

9.2.4 耦合性（续）

- 包含或导入耦合—当构件A引入或者包含一个构件B的包或者内容时
- 外部耦合—当一个构件通信和协作时发生

构件的UML表示

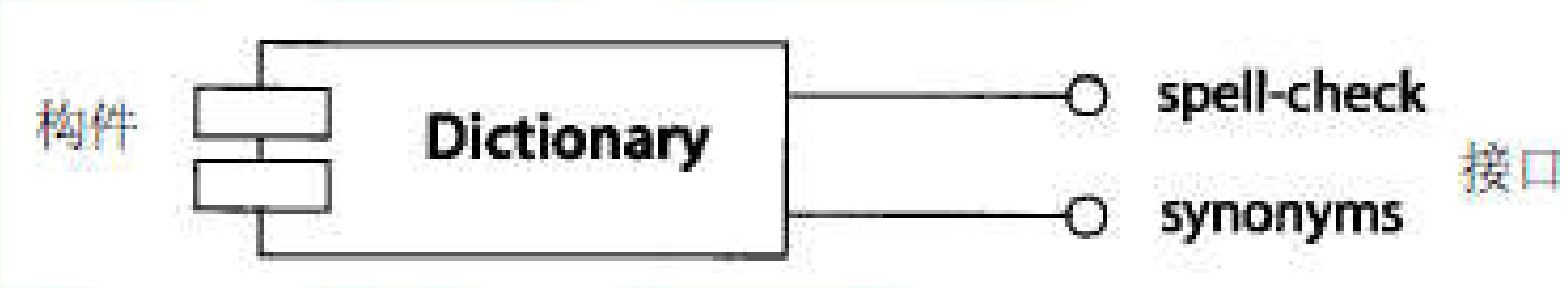


图 带接口的构件

构件具有它们支持的接口和需要从其他构件得到的接口

构件图表示了构件之间的依赖关系。
每个构件实现（支持）一些接口，并使用另一些接口。如果构件间的依赖关系与接口有关，那么构件可以被具有同样接口的其他构件替代。

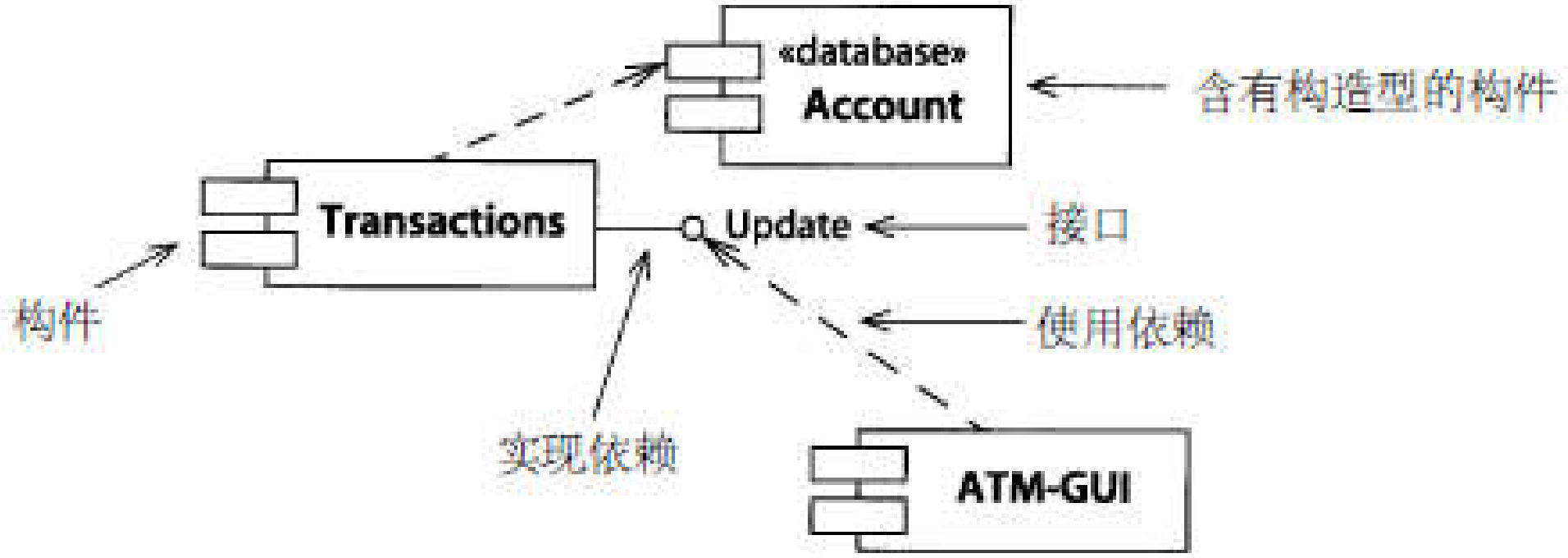
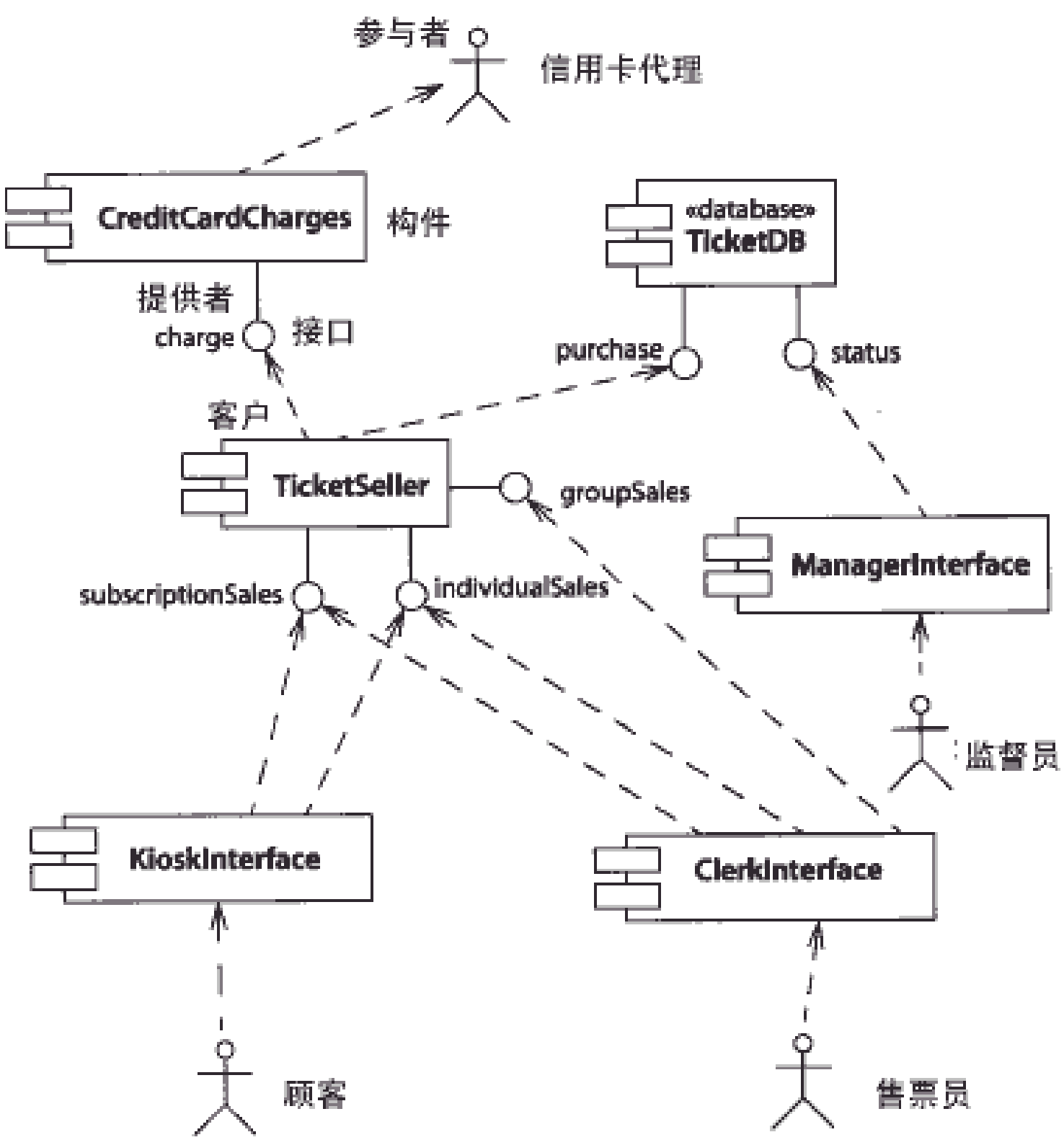


图 构件图

构件图举例


参与者
信用卡代理
CreditCardCharges
构件
«database»
TicketDB
提供者
charge
接口
purchase
status
客户
TicketSeller
groupSales
ManagerInterface
subscriptionSales
individualSales
监督员
KioskInterface
ClerkInterface
顾客
售票员

9.3 实施构件级设计

一个典型的构件级设计步骤：

1. 标识出所有与问题域相对应的类
2. 确定所有与基础设施域相对应的类

- GUI构件、操作系统构件、对象和数据管理构件

3. 细化所有不能作为复用构件的类

（1）说明消息的细节流

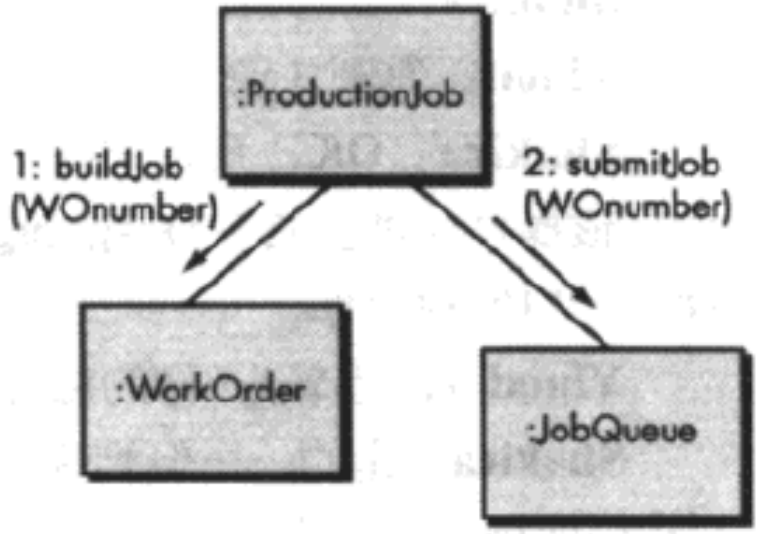
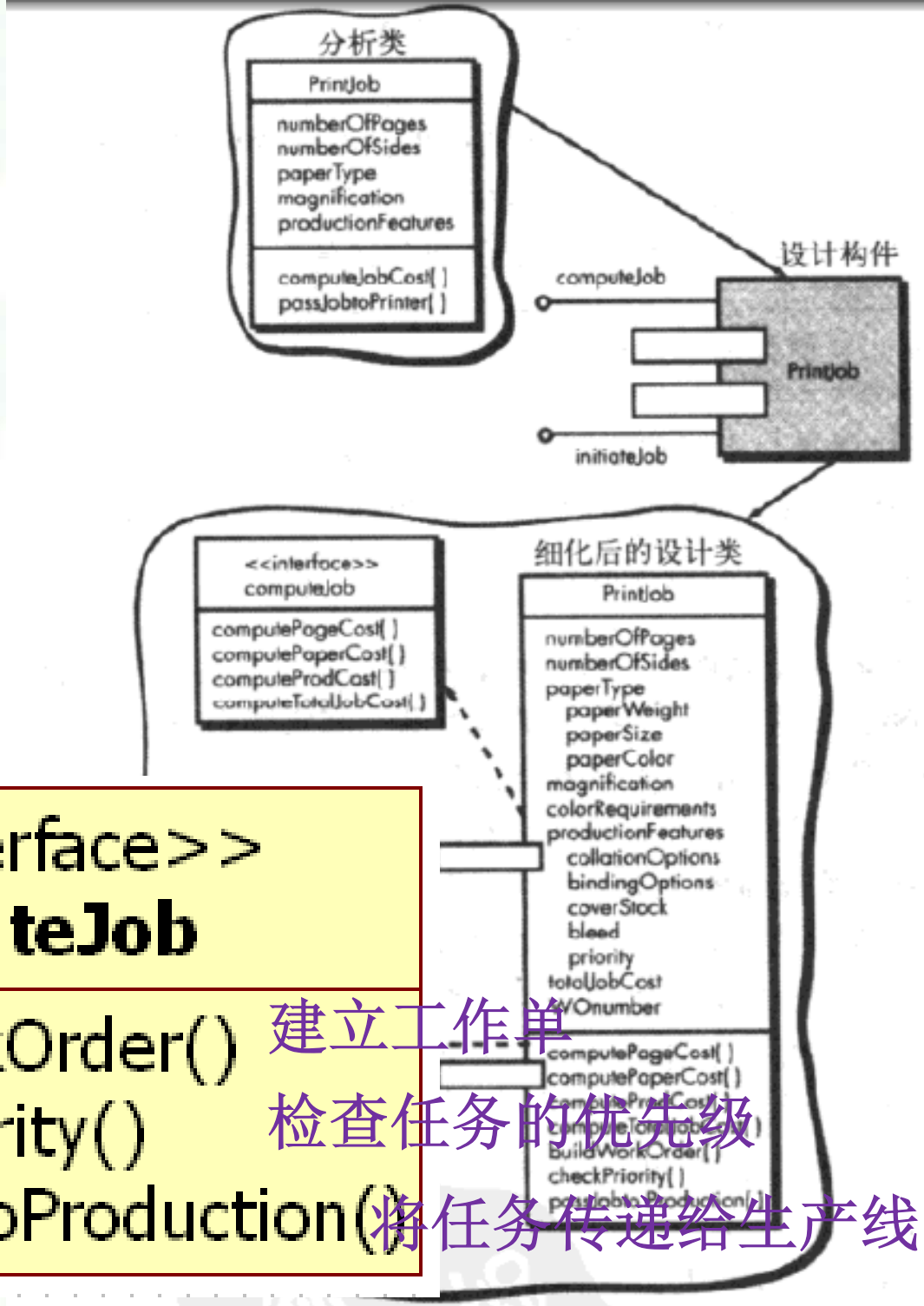


图　　带消息的协作图

[guard condition] sequence expression (return value) :=
message name (argument list)

- （2） 为每个构件确定适当的接口
  - UML接口是“一组外部课件的（即公共的）操作，接口不包含内部结构、没有属性，没有关联……”
  - 为设计类定义的接口可以归结为一个或者更多的抽象类
  - 抽象类中的每个操作接口应该是内聚的

内聚性差！



<<interface>>
initateJob

+buildWorkOrder()　建立工作单

+checkPriority()　检查任务的优先级

+passJobToProduction()　将任务传递给生产线

图　设计构件的细化

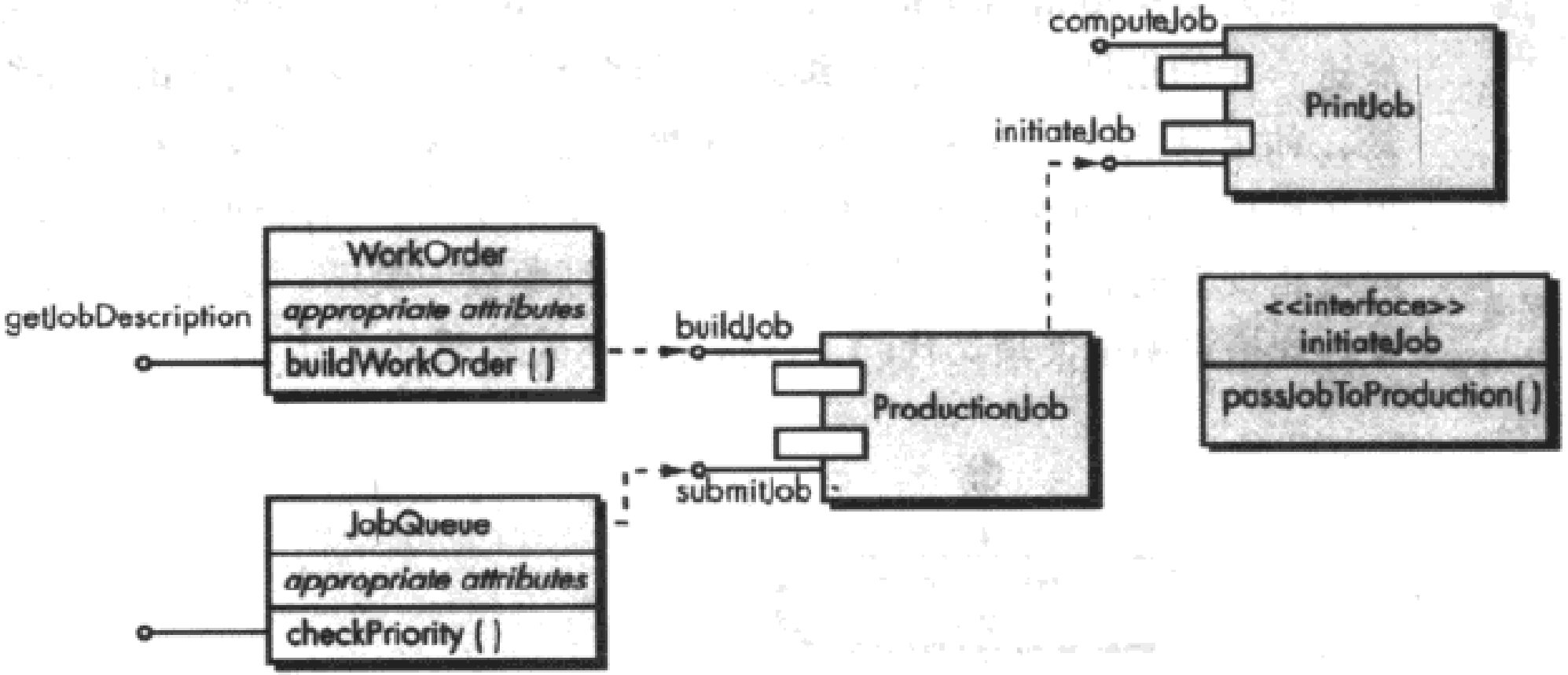


图　　为PrintJob重构接口和类定义

小圆圈代表接口
从构件到接口的实线表明该构件提供的列在接口旁的服务
从构件到接口的虚线箭头说明这个构件要求接口提供的服务

9.3 实施构件级设计（续）

（3）细化属性并定义数据类型和结构

- UML用下面的语法来定义属性的数据类型

```
name : type-expression = initial-value {property string}
```

举例：

```
paperType-weight: string = "A" { contains 1 of 4 values - A, B, C, or D}
```

9.3 实施构件级设计（续）

- （4）描述每个操作中的处理
  - 采用如下方式进行扩展：

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

- 若算法比较复杂或者难于理解，需要设计细化

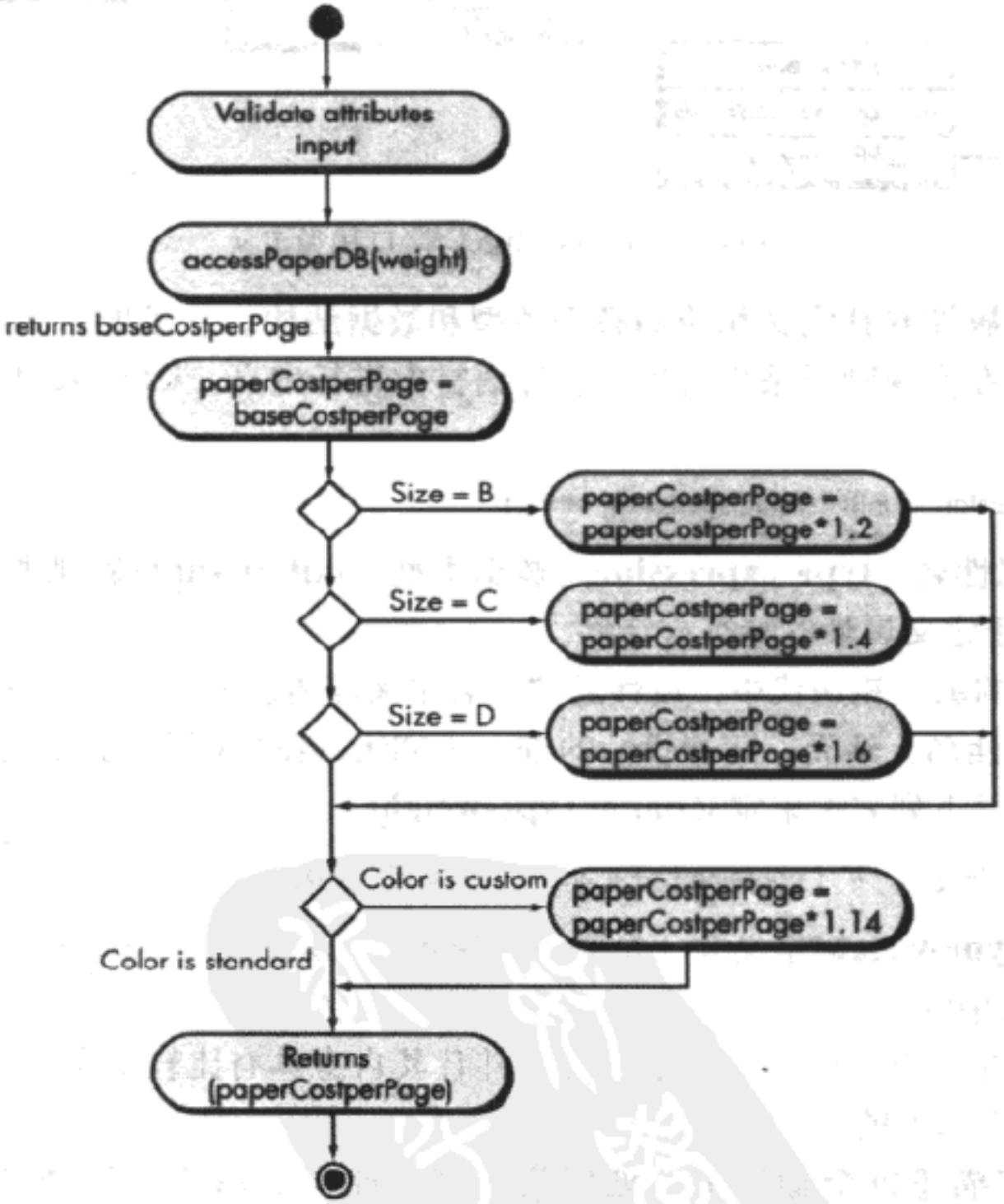


图　　computePaperCost()操作的UML活动图

9.3 实施构件级设计（续）

4. 说明持久数据源（数据库或文件）等相关类

5. 开发并细化类的行为表示

event-name (parameter-list) [guard-condition] / action expression

表示状态图中的事件

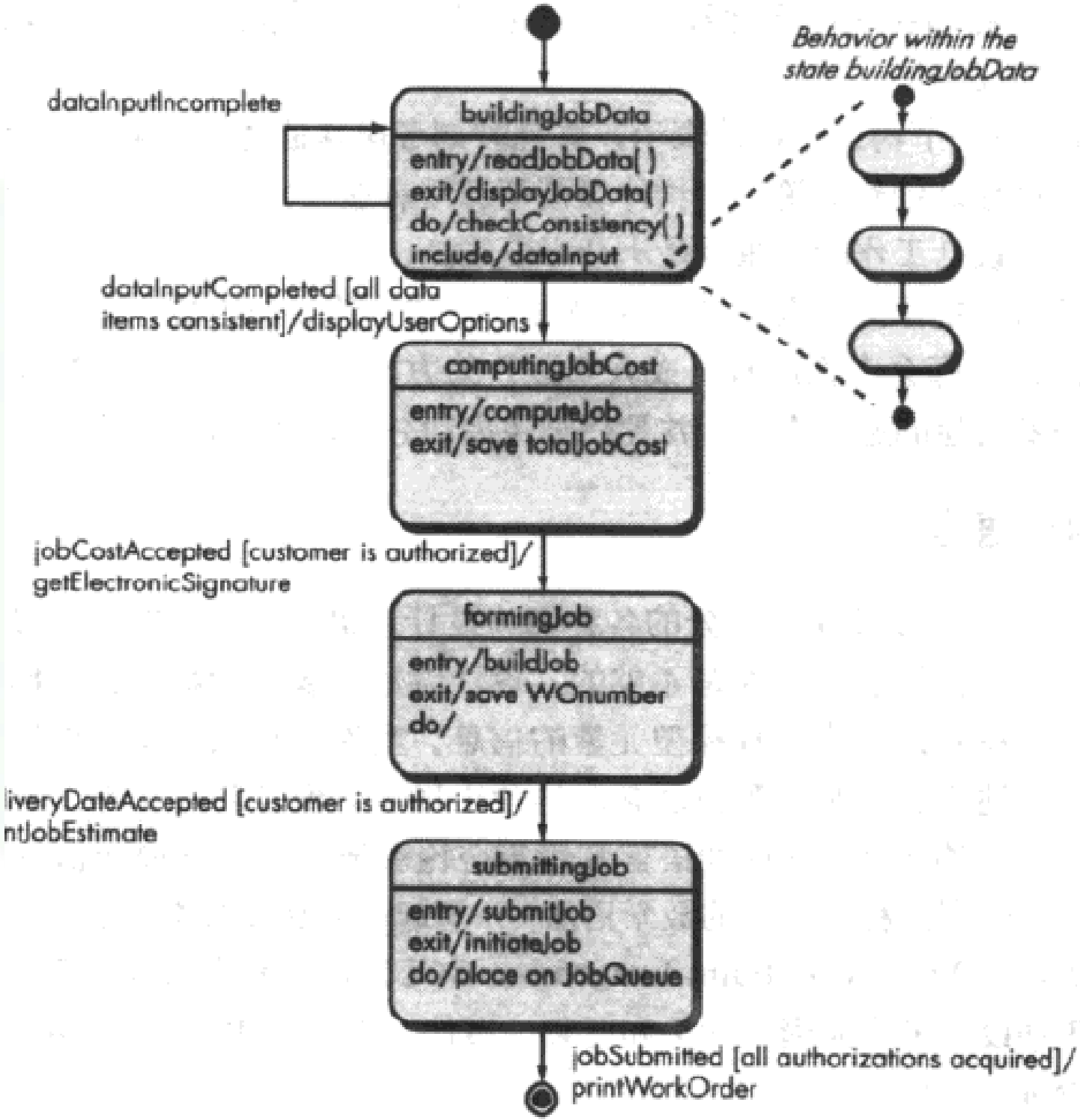
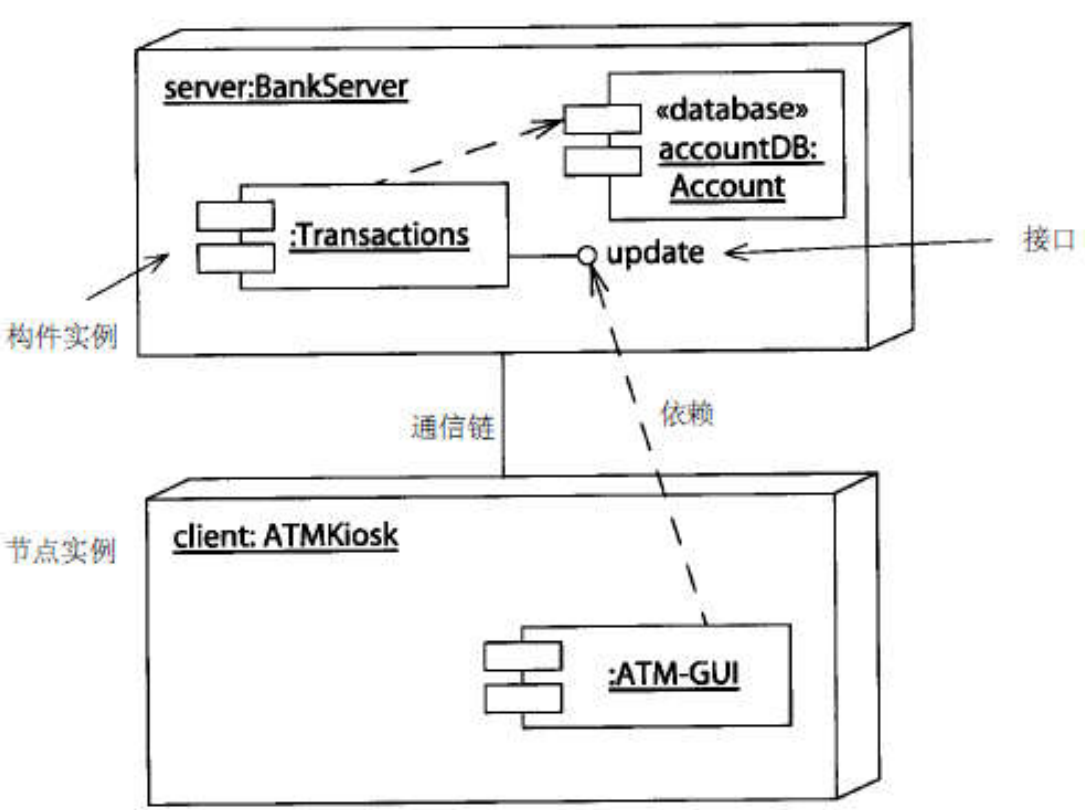


图　　PrintJob类的状态图

9.3 实施构件级设计（续）

6. 细化部署图

- 表示主要构件包的位置
- 某些情况下，部署图在这个时候被细化为实例形式



7. 反省和检查现有的设计

9.4 对象约束语言

- 对象约束语言（Object Constraint Language, OCL），一种形式化语言
- 四个组成部分：
  - 语境—定义了哪些情况语句是正确的
  - 特征—描述语境的一些特征
  - 操作—用来操纵和限制一个特性
  - 关键字—用于说明条件表达式

    （if, then, else, and, or, not, implies）

9.4 对象约束语言(续)

- OCL的语法和结构:
  - UML和其他建模语言建模描述系统特征是通过OCL表达式来实现，因此表达式是OCL的核心，它携带了关于它所约束对象的相关信息
  - 作为OCL的核心，表达式可以被用于多种不同的场合，以不同的形式规约被描述的对象。

9.4 对象约束语言（续）

- 举例：

Customer

self.authorizationAuthority = ‘yes’

用于表示PrintJob状态图中jobCostAccept事件的警戒条件

用OCL说明前置条件和后置条件

```
context PrintJob::validate(upperCostBound : Integer, custDeliveryReq :
  Integer)
   pre: upperCostBound > 0
        and custDeliveryReq > 0
        and self.jobAuthorization = 'no'
   post: if self.totalJobCost <= upperCostBound
            and self.deliveryDate <= custDeliveryReq
        then
            self.jobAuthorization = 'yes'
        endif
```

9.5 设计传统构件

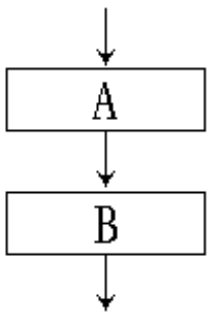
- 由各种逻辑结构组成：
  - 顺序型
  - 条件型
  - 重复型

常用方法

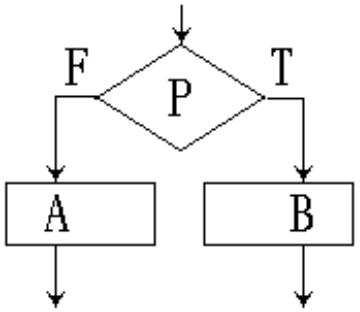
1. 图形工具
   - 程序流程图
   - 盒图(N-S图)
   - 问题分析图(PAD)
2. 表格工具
   - 判定表
3. 语言工具
   - 过程设计语言(PDL)(伪码)

9.5.1 图形化设计表示

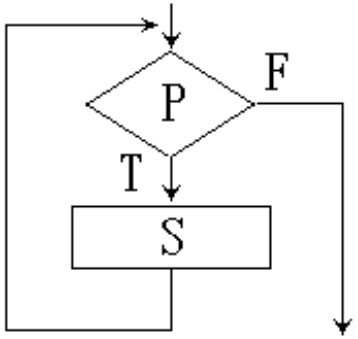
一、流程图



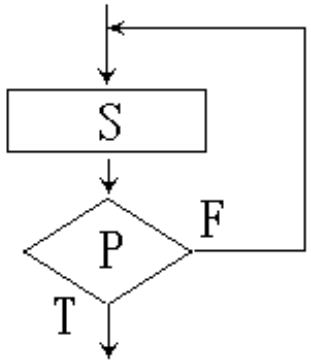
① 顺序型



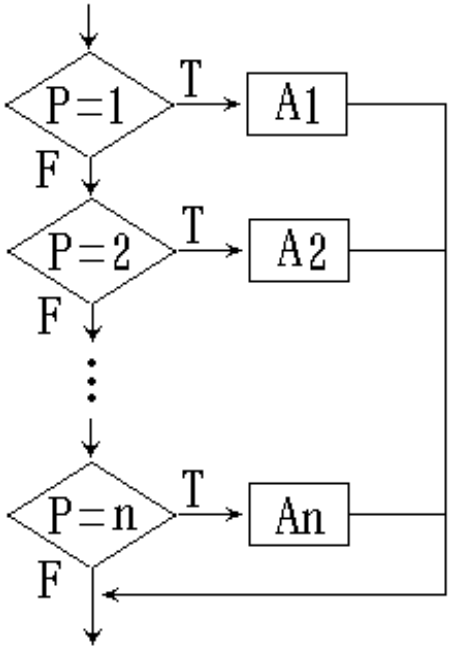
② 选择型



③ 先判定型循环
(DO-WHILE)

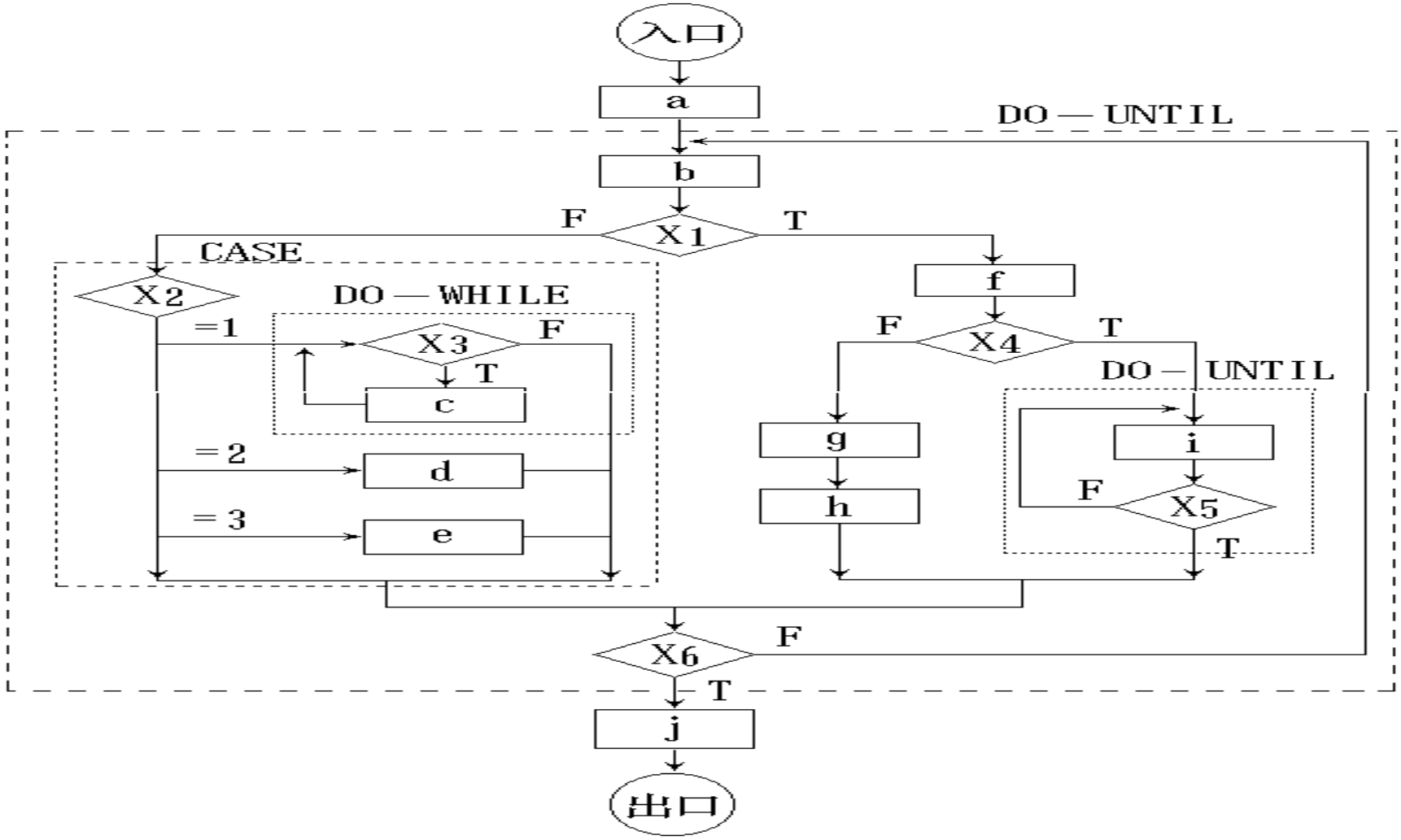


④ 后判定型循环
(DO-UNTIL)



⑤ 多情况选择型
(CASE型)

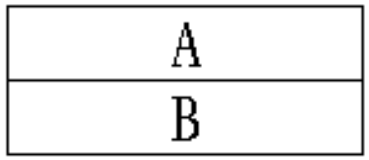
流程图举例


入口
a
DO — UNTIL
b
F
X1
T
CASE
X2
DO — WHILE
=1
X3
F
T
c
=2
d
=3
e
f
F
X4
T
DO — UNTIL
g
h
i
F
X5
T
X6
F
T
j
出口

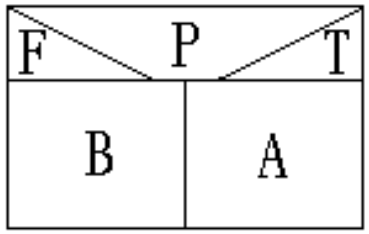
程序流程图

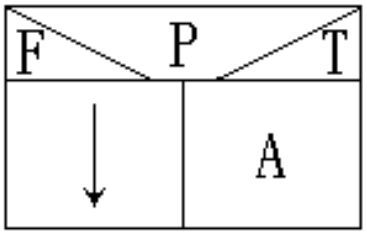
- 任何复杂的程序流程图都应由以上五种基本结构组合而成。
- 优点：
  - 容易掌握，且历史“悠久”，使用广泛。
- 缺点：
  - 本质上不具备逐步求精的特点
  - 对于提高大型系统的可理解性作用甚微
  - 不易表示数据结构
  - 转移控制不太方便

二、盒图，也叫N-S图

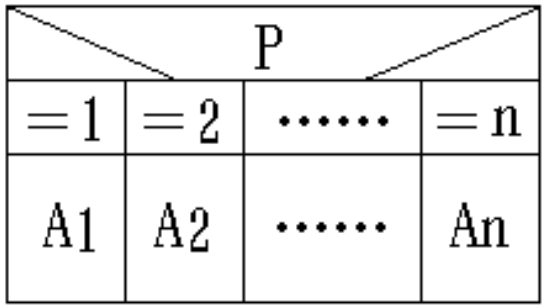


① 顺序型

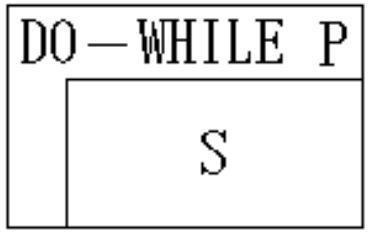




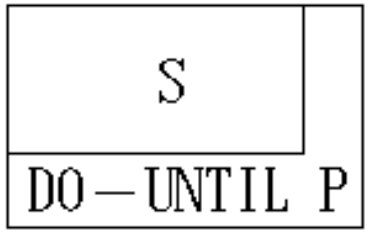
② 选择型



⑤ 多分支选择型
（CASE型）

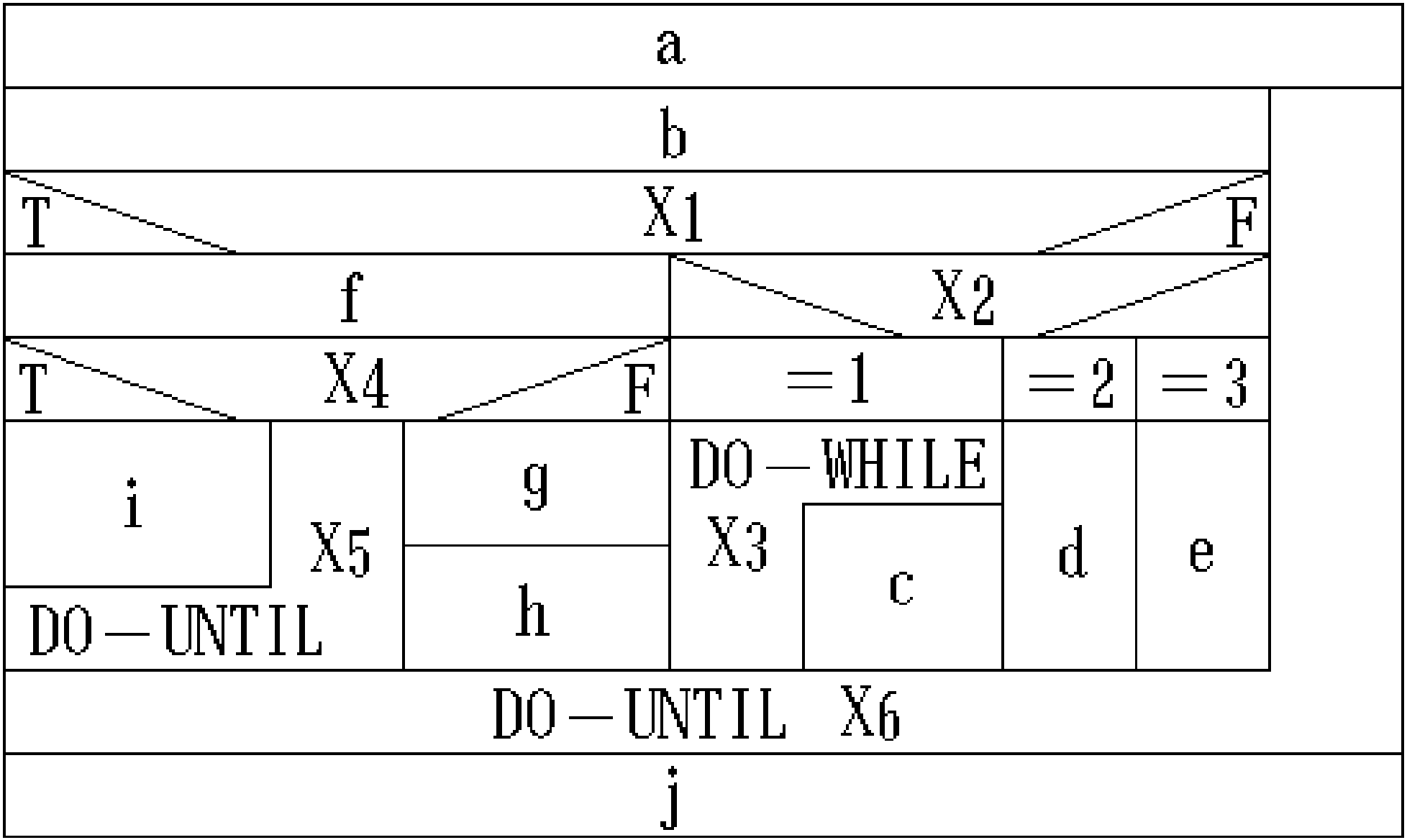


③ WHILE重复型

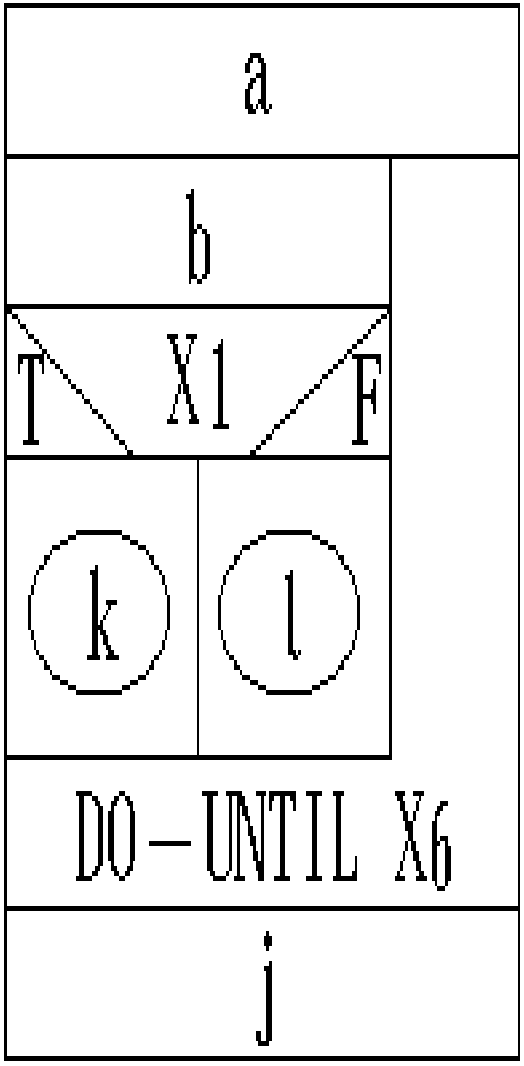


④ UNTIL重复型

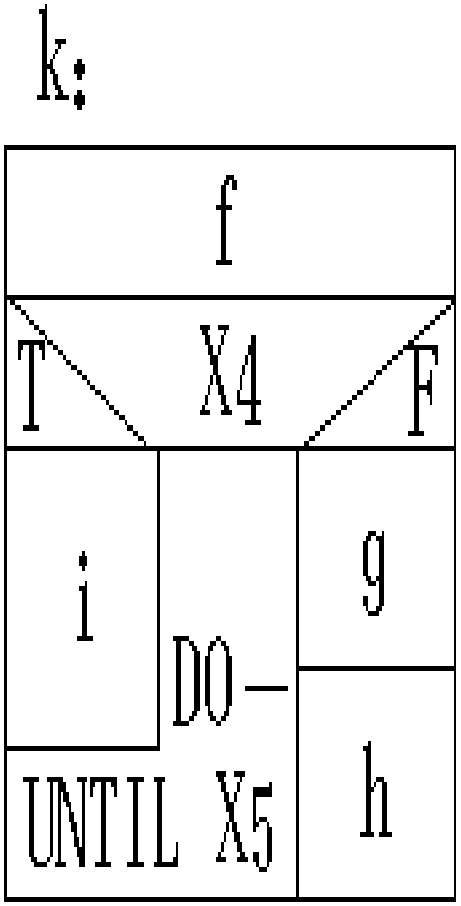
盒图举例


a
b
T
X1
F
f
X2
T
X4
F
=1
=2
=3
i
X5
g
h
DO—WHILE
X3
c
d
e
DO—UNTIL
DO—UNTIL
X6
j

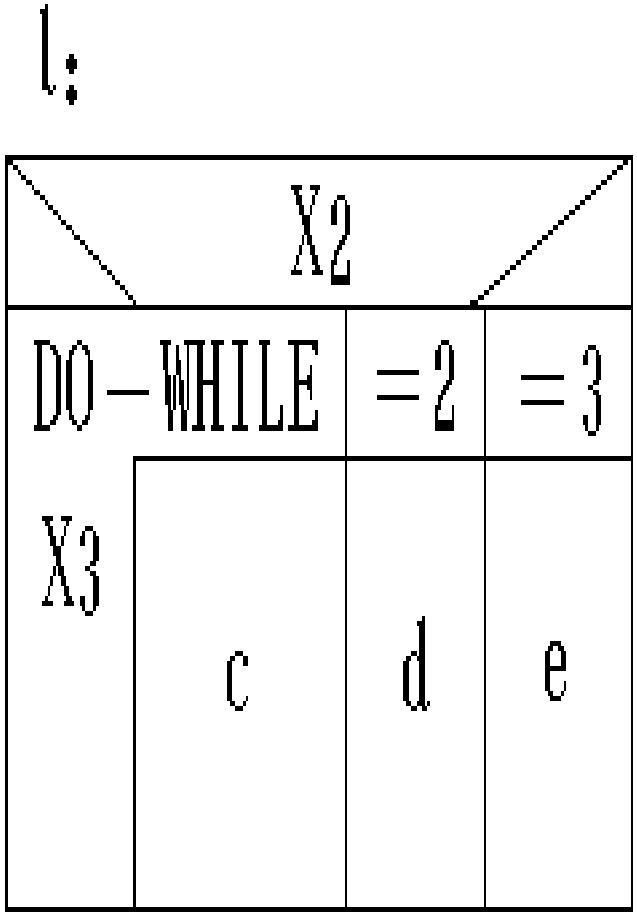
盒图的嵌套定义形式



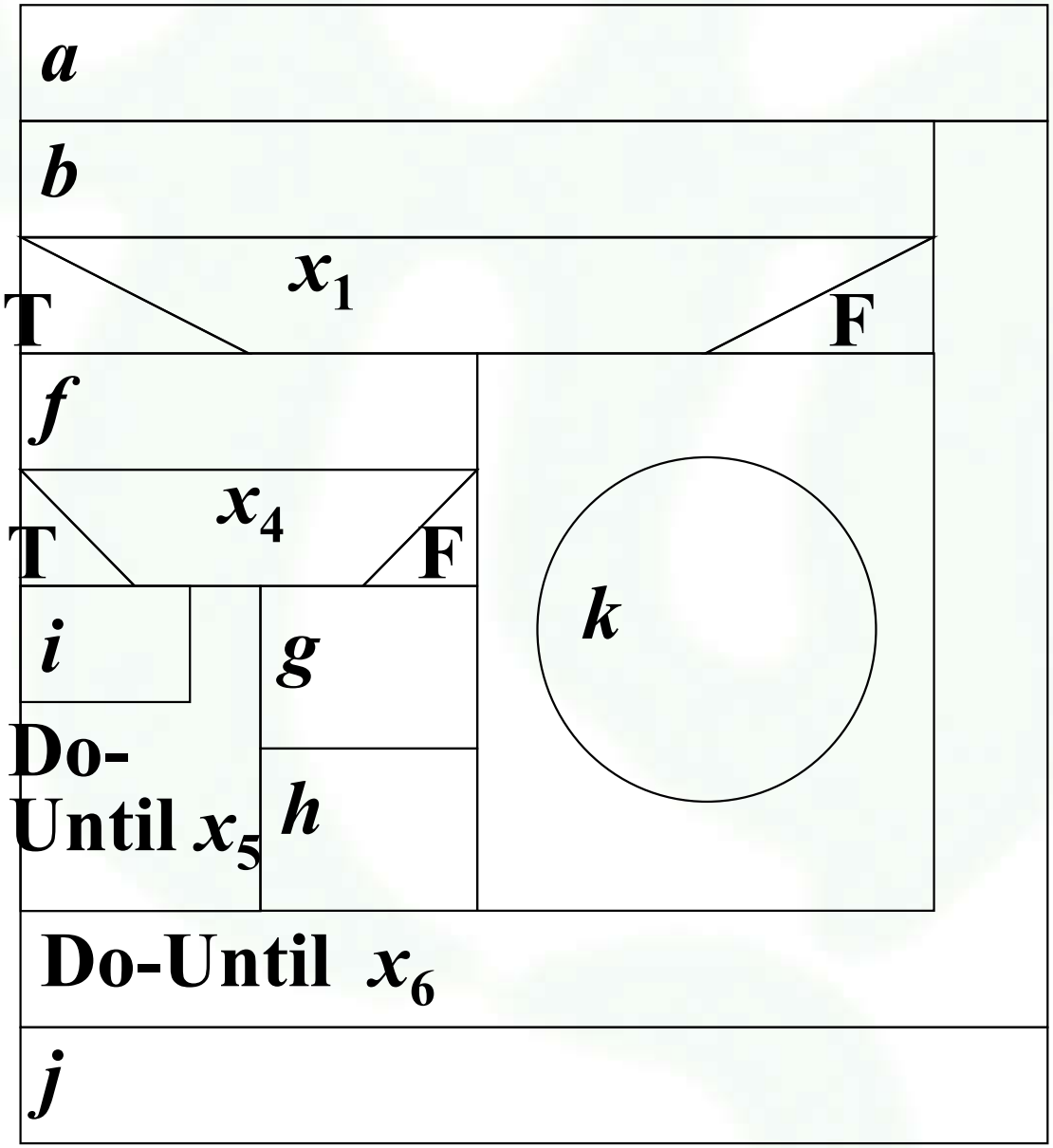
(a)



(b)



(c)



k :

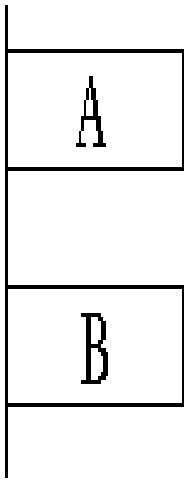
x2		
1	2	3
Do-While x3	d	e
c		

盒图

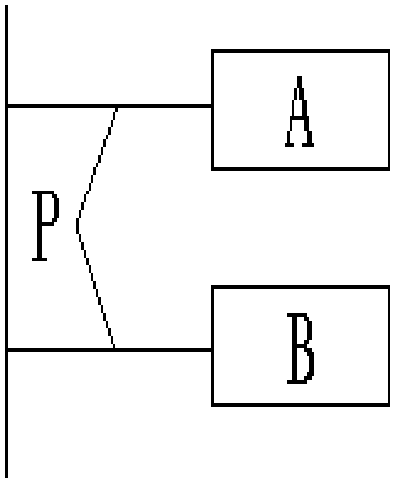
- 特点：
  - 没有箭头，不允许随意转移控制；
  - 每个矩形框(Case中条件取值例外)都是一个功能域(即一个特定结构的作用域)，结构表示明确；
  - 局部及全程数据的作用域易见；
  - 易表现嵌套关系(embedded structure)以及模块的层次结构。

三．问题分析图(PAD)

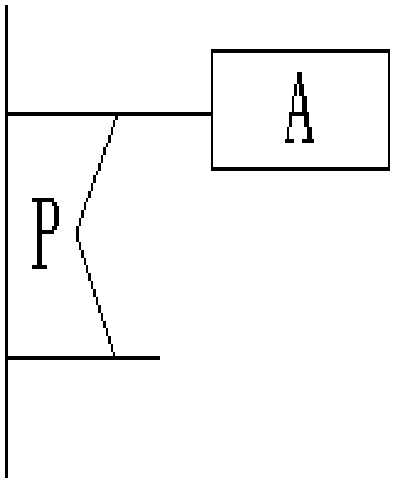
PAD(Problem Analysis Diagram):日立公司，1973

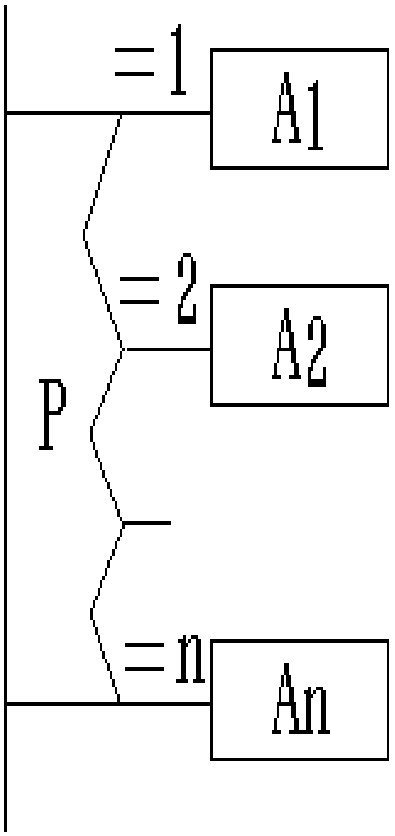


① 顺序型

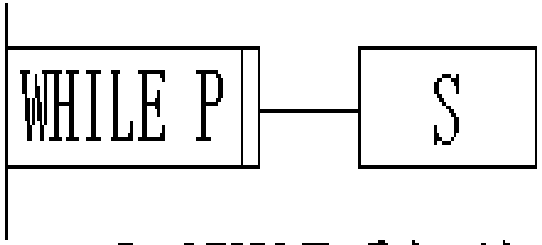


② 选择型

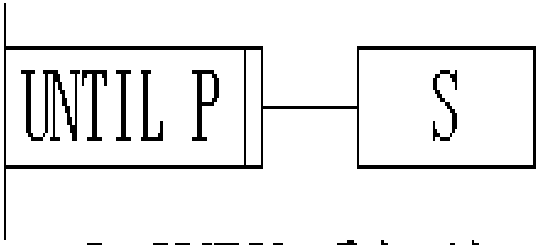




⑤ 多分支选择型（CASE型）

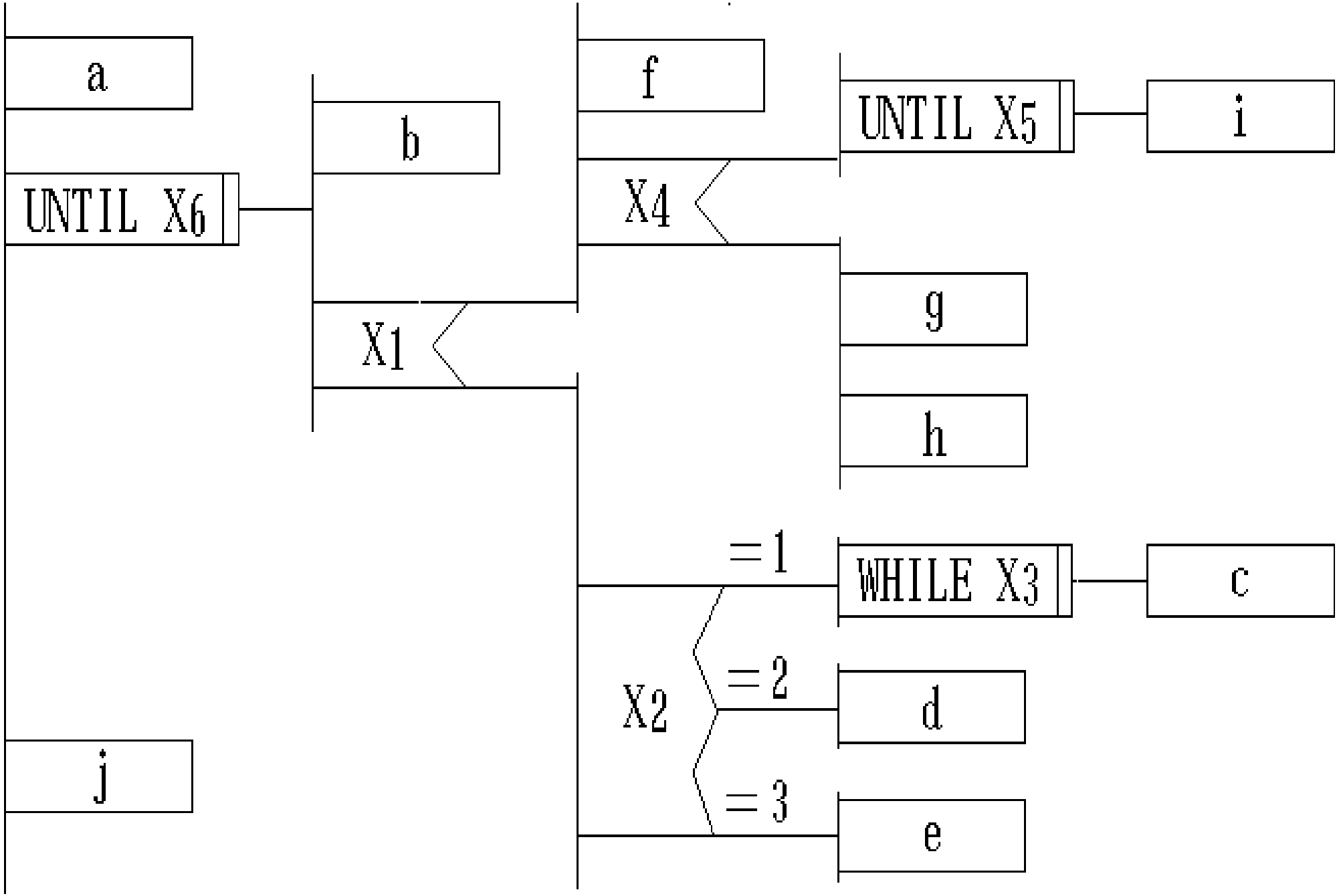


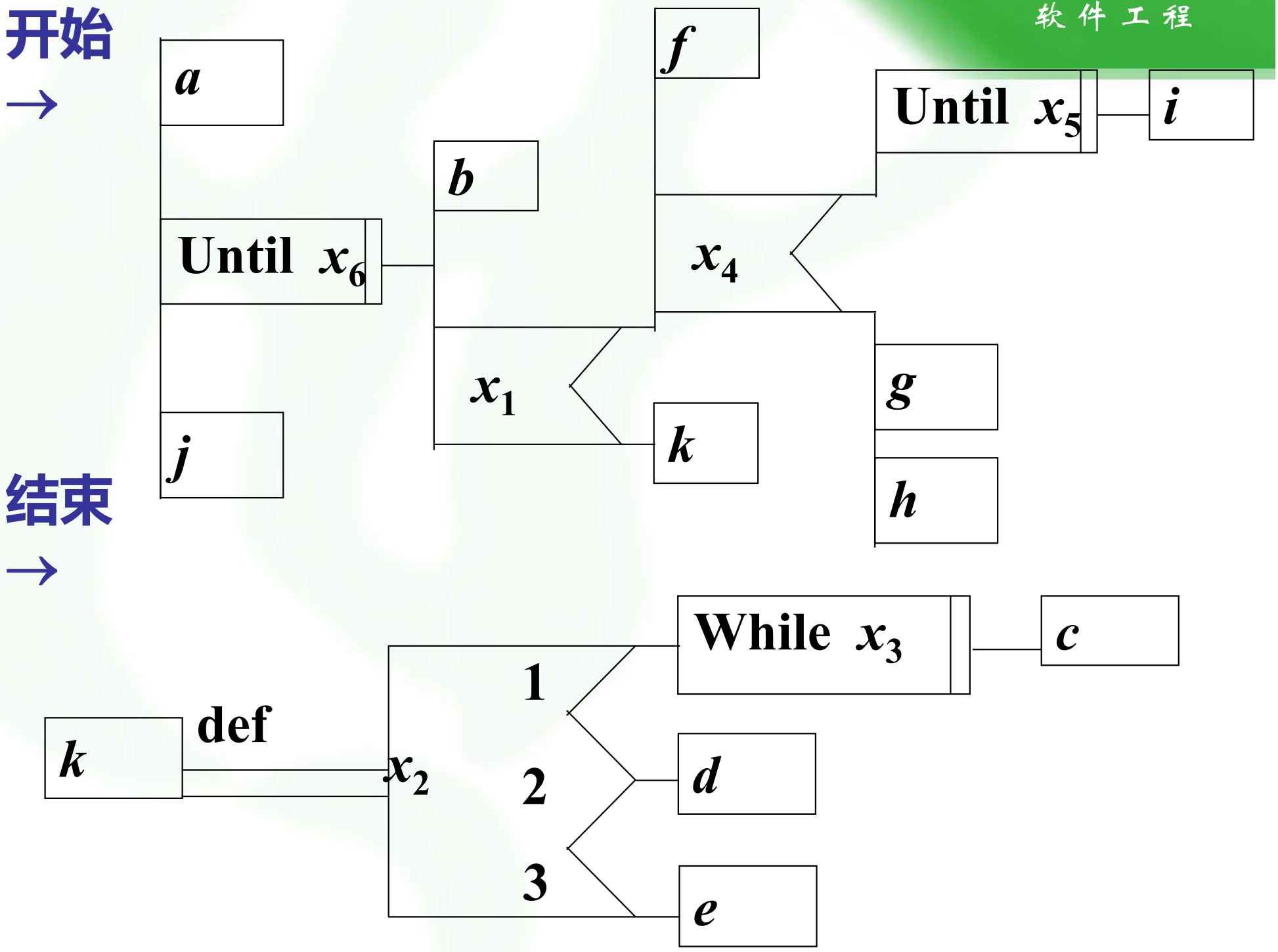
③ WHILE重复型



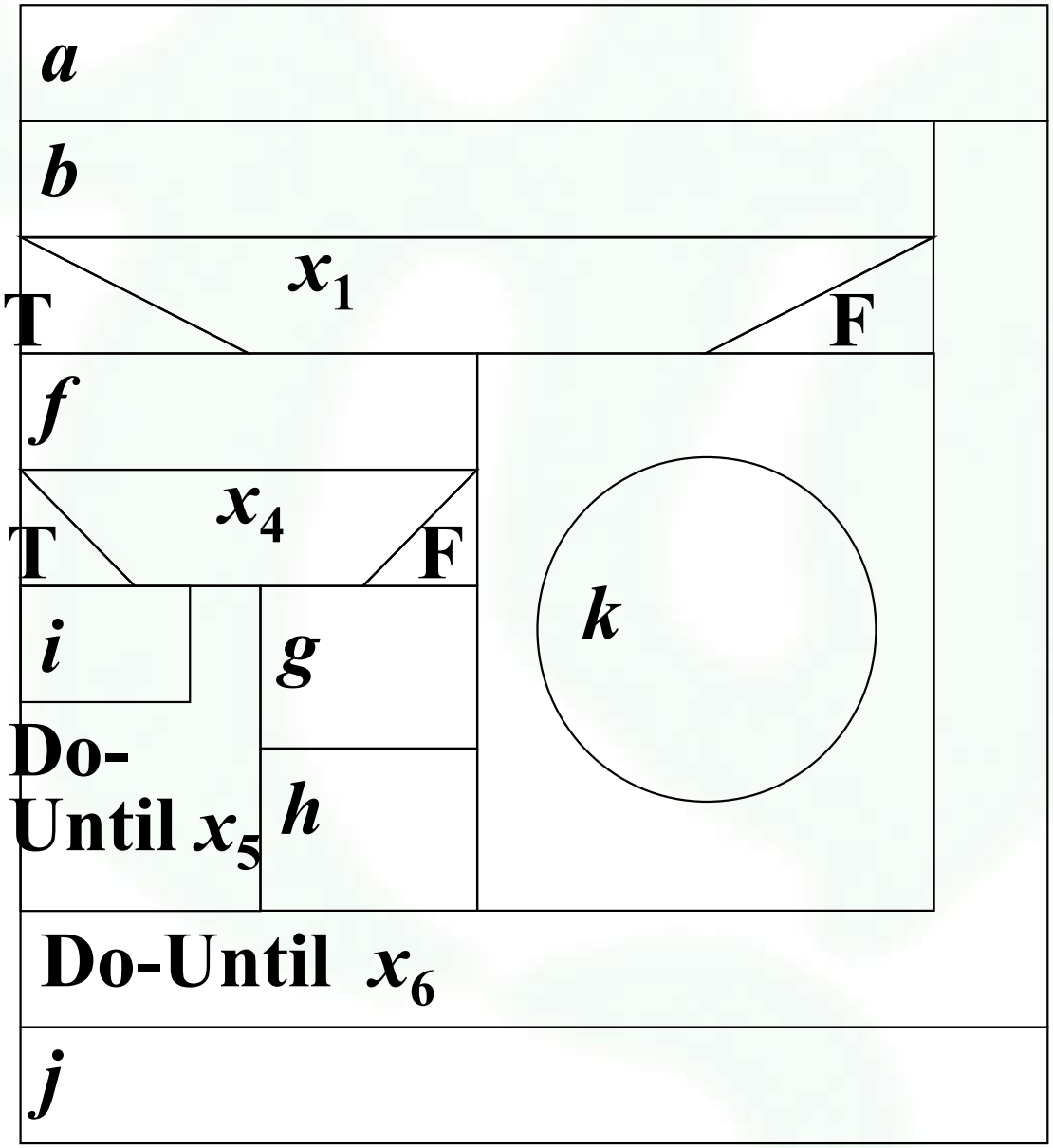
④ UNTIL重复型

PAD描述的示例




开始
a
f
Until x_5
i
b
Until x_6
x_4
x_1
k
g
h
j
结束
While x_3
c
1
def
k
x_2
2
d
3
e

例：N-S图与PAD的转换



k：

x2		
1	2	3
Do-While x3	d	e

PAD

- PAD特点：
  - 结构清晰，层次分明，易读；
  - 支持逐步求精的设计思想；
  - 容易将PAD自动转换为高级语言源程序。

9.5.2 表格式设计方式：

- 决策表---将事件、事件的发生条件、发生的具体模式等信息填写在一张表里。

	规则					
条件	1	2	3	4	5	6
普通会员	T	T				
白银会员			T	T		
黄金会员					T	T
特别折扣	F	T	F	T	F	T
动作						
无折扣	✓					
实行8%折扣			✓	✓		
实行15%折扣					✓	✓
实行特殊的x%折扣		✓		✓		✓

图　　决策表

9.5.3 程序设计语言

也成结构化英语或伪代码

```
/* 计算运费*/
count ( );
{ 输入x；输入y;
if 条件1 { 公式1；call sub；}
else if 条件2 { 公式2；call sub；}
else if 条件3 { 公式3；call sub；}
else { 公式4；call sub；}
}
sub ( );
{ for(I=1，3) do { 记帐；输出；}
}
```

PDL的特点

- 优点：易于实现由PDL到源代码的自动转换
- 缺点：不够直观。

9.6 基于构件的开发

- 基于构件的软件工程(Component-Based Software Engineering)是一种强调使用可服用的软件构件来设计与构造计算机系统的过程。

- 问题：
  - 仅仅将多组可复用的软件构件组合起来，就能构造出一个复杂的系统吗？
  - 这种工作能够以一种高效和节省成本的方式完成吗？
  - 能否建立恰当的激励机制来鼓励软件工程师复用而不是重新开发？
  - 管理团队是否也愿意为构造可复用软件构件过程中的额外开销买单？
  - 能否以使用者易于访问的方式构造复用所必须的构件库？
  - 已有的构件可以被需要的人找到并使用吗？

9.6.1 领域工程

- 领域工程的目的是识别、构造、分类和传播一组软件构件，这些构件在某一特定的应用领域中可以适用于现在或未来的软件。
- 三个主要活动：分析、构建和传播。

9.6.2构件的合理性检验、适应性修改与组合

- 可复用构件的存在并不能保证这些构件可以很容易地很有效地被集成到为应用所选择的体系结构中

构件合格性检验

- 保证某候选构件能够执行需要的功能，完全适合系统的体系结构，并具有该应用所需的质量特性（例如，性能、可靠性、可用性）
- 契约式设计：
  - 定义明确的和可核查的构件接口规格说明，从而使构件的潜在用户快速了解其意图。
  - 将前置条件、后置条件和不变式的表述加入到构件规格说明中。

- 构件合格性检验的一些重要因素：
  - 应用编程接口、构件所需的开发工具和集成工具、运行时需求、服务需求、安全特征、嵌入式设计假设、异常处理

构件适应性修改

- 集成时可能存在冲突，常使用构件包装的适应性修改技术
  - 白盒包装技术（有内部设计和代码的完全控制权）
  - 灰盒包装技术（构件扩展语言或API）
  - 黑盒包装技术（接口中预处理和后处理）

构件组合

- 建立一个基础设施域（通常是专门的构件库）以将构件绑定到一个运行系统中。

9.6.2 体系结构不匹配

- 可复用的构件设计者常常对耦合构件的有关环境进行隐式假设，如果假设不正确，就产生了体系结构不匹配的情况。
- 设计概念，如抽象、隐藏、功能独立、细化……都有助于防止体系结构不匹配。

9.6.3 复用的分析与设计

- 可复用设计需要软件工程师采用良好的软件设计概念和规则。
- 建立标准数据、接口和程序模板，就有了进行设计所依托的框架。

9.6.4 构件的分类与检索

- 如何以无歧义的、可分类的术语来描述软件构件？
  - 如Tracz提出的3C模型：概念、内容和环境。
- 复用环境应具备特点：
  - 存、检索构件的构件数据库；访问数据库的管理系统；检索系统；CBSE工具，支持将复用的构件集成到新的设计或实现中。

作业

- 逐步求精和重构是一回事吗？如果不是，他们有什么区别？
- 简述实施构件级设计的步骤。