软件工程

第13章 构件级设计

徐本柱 软件学院

2018-10

主要内容

- 什么是构件
- 设计基于类的构件
- 实施构件级设计
- 设计传统构件
- 基于构件的开发

概念

- 完整的软件构件是在体系结构设计过程中定义的
  - 但没有在接近代码的抽象级上表示
    - 内部数据结构和
    - 每个构件的处理细节。
- 构件级设计定义了
  - 数据结构、
  - 算法、
  - 接口特征和
  - 分配给每个软件构件的通信机制。

重要性和步骤

- 在建造软件之前就确定该软件是否可以工作
  - 为了保证设计的正确性，
  - 与早期设计表示（数据、体系结构和接口设计）的一致性，
  - 构件级设计需要以一种可以评审设计细节的方式来表示软件。
  - 提供了一种评估数据结构、接口和算法是否能够工作的方法。
- 数据、体系结构和接口构成了构件级设计的基础
  - 每个构件的类定义或者处理叙述都转化为一种详细设计，
  - 采用图形或文本的形式详述内部的
    - 数据结构、
    - 局部接口细节和
    - 处理逻辑。
  - 设计符号包括UML图和一些辅助表示。
  - 采用现有的可复用软件构件，而不是开发新的构件。

工作产品和质量保证措施

- 每个构件的设计都以图形、表格或文本方式表示
- 采用设计评审机制：
  - 对设计执行检查以确定
    - 数据结构、
    - 接口、
    - 处理顺序和
    - 逻辑条件等是否都正确，
  - 给出早期设计中与构件相关的数据或控制变换。

引言

- 体系结构设计第一次迭代完成后，开始构件级设计
- 该阶段，全部的数据和软件的程序结构都已经建立
- 目的是把设计模型转化为运行软件。
- 现有设计模型的抽象层次相对较高，
- 而可运行程序的抽象层次相对较低。
- 这种转化具有挑战性，
  - 因为可能会在软件过程后期阶段引入难于发现和改正的微小错误。
- 设计模型转化为源代码时，必须遵循一系列设计原则
- 保证不在开始时就引入错误
- 程序创建以体系结构设计模型为指南→中间表示（构件级设计）
- 主要讨论设计指导准则和设计方法

13.1 什么是构件

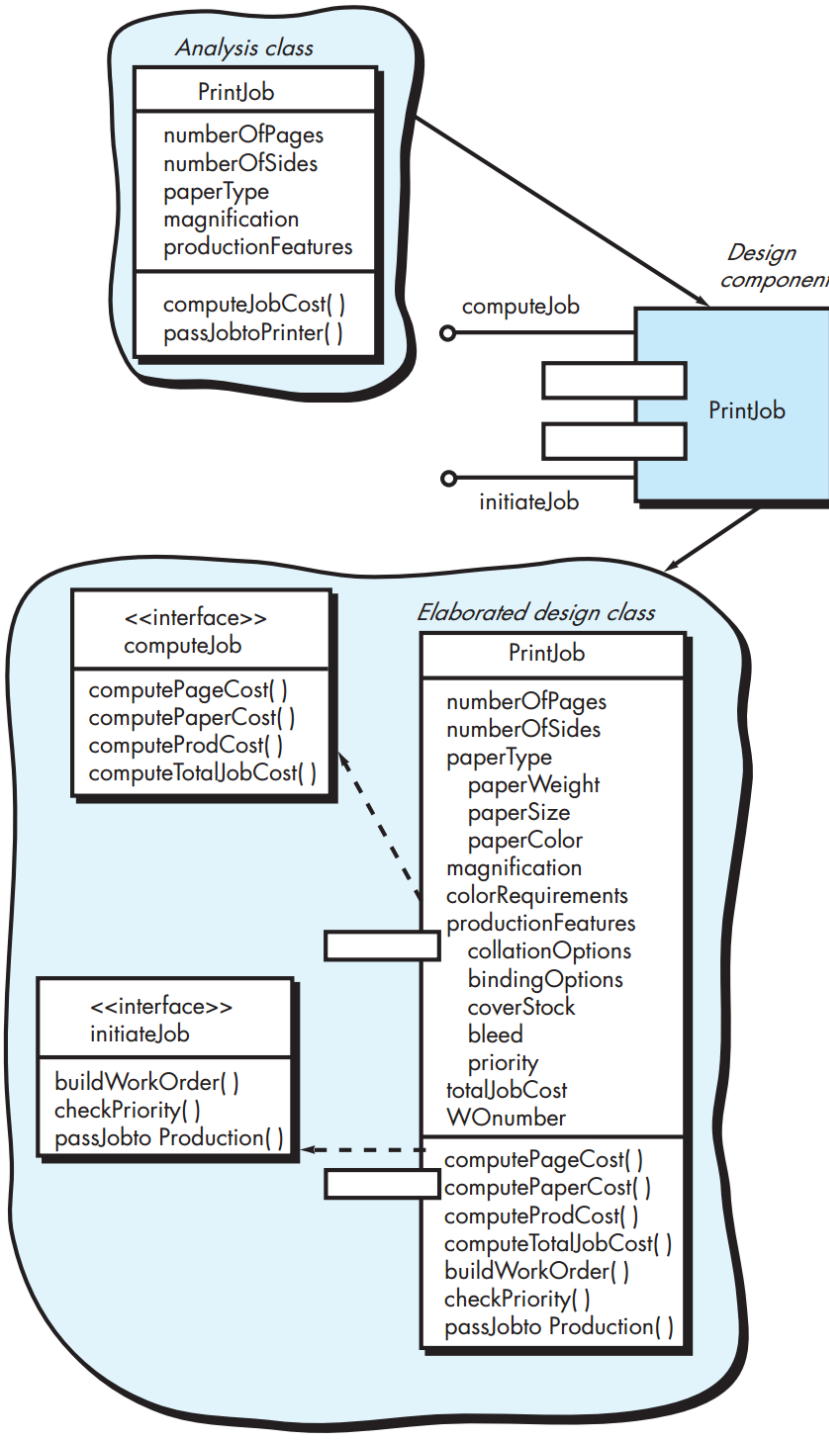
- 构件是
  - 计算机软件中的一个模块化的构造块。
  - **“系统中模块化的、可部署的和可替换的部件，该部件封装了实现并暴露一系列接口”**
- 构件存在于软件体系结构中，
  - 在完成所建系统的需求和目标中起重要作用
  - 驻留在其内部，必须与
    - 其它构件和
    - 存在于软件边界以外的实体
  - 进行通信和合作。

13.1.1 面向对象的观点

- OO软件工程环境中，**构件一个协作类的集合**
  - 每一个类都被详细阐述，包括
    - 所有的属性和
    - 与其实现相关的操作
    - 与其他设计类相互通信协作的接口
  - 从分析模型开始，详细描述
    - 分析类（该类与问题域相关）和
    - 基础类（该类为问题域提供了支持性服务）
- 例：考虑为一个高级印刷车间构建软件
  - 目的是为了收集前台的客户需求，
  - 对印刷业务进行定价，
  - 然后把印刷任务交给自动生产设备。
  - 在需求工程中得到一个PrintJob的分析类，有两个接口：
    - computeJob具有对任务进行定价的功能，
    - initiateJob能够把任务传给生产设备。

印刷车间实例

- 需求工程中得到分析类PrintJob
  - 分析过程中定义其属性和操作
- 体系结构设计中定义为一个构件
  - computeJob接口
  - initiateJob接口
- 开始构件级设计，细化构件的细节
  - 不断补充作为构件PrintJob的类的全部
    - 属性和操作，
    - 逐步细化最初的分析类。
- 对每个构件都要实施细化
  - 对每一个属性、
  - 每一个操作和
  - 每一个接口
- 还要说明实现与操作相关
  - 处理逻辑的算法细节
  - 实现接口所需机制的设计
  - 实现系统内部对象间消息通信机制



13.1.2 传统观点

- 一个构件就是程序的一个**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

- 也称为模块，承担如下三个重要角色之一：
  - (1)控制构件，协调问题域中所有其他构件的调用;
  - (2)领域构件，完成部分或全部用户的需求;
  - (3)基础设施构件，负责领域所需相关处理的功能。

印刷车间实例（传统观点）

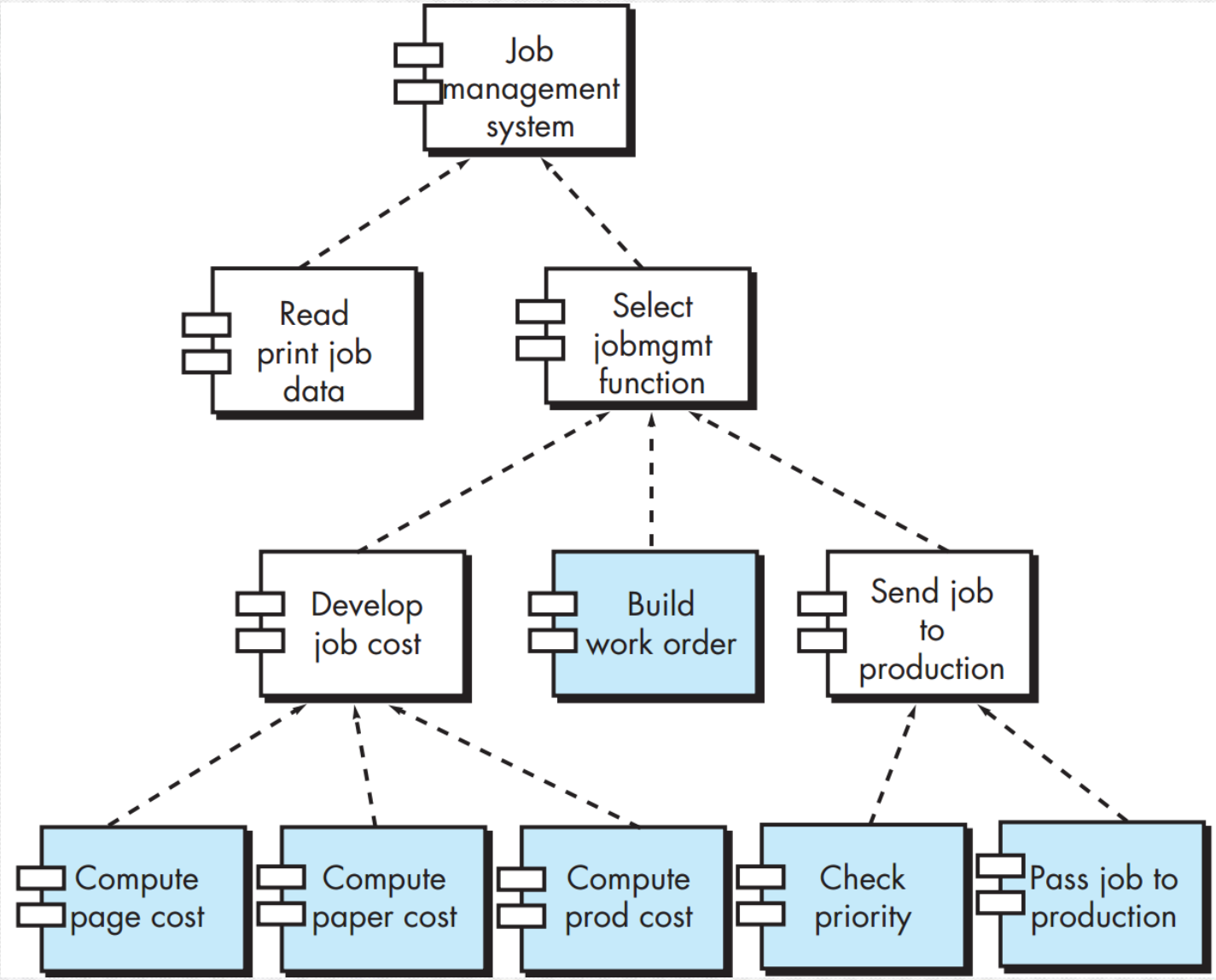


图13-2 一个传统系统的结构图

印刷车间实例（传统观点）

- 在构件级设计中每个模块都要被细化
  - 需要明确定义模块的接口
  - 即每个经过接口的数据或控制对象都需要明确加以说明
  - 定义模块内部使用的数据结构，
  - 采用逐步求精方法设计完成模块中相关功能的算法。
- 考虑ComputePageCost模块
  - 目的在于根据用户提供的规格说明计算每页的印刷成本。
  - 需要以下数据：
    - 文档的页数，文档的印刷份数，
    - 单面或者双面印刷，颜色，纸张大小。
  - 通过该模块的接口传递给ComputePageCost
    - 根据任务量和复杂度，使用这些数据来决定一页的费用
    - 通过接口将所有数据传递给模块

印刷车间实例（传统观点）

- 使用UML建模符号描述的构件级设计。
  - ComputePageCost模块通过调用
    - getJobData模块和
    - 数据库接口accessCostDB来访问数据。
  - ComputePageCost进一步细化→算法、接口细节描述
    - 算法的细节可以由伪代码或者UML活动图来表示
    - 接口被表示为一组输入和输出的数据对象或数据项的集合
  - 设计细化的过程需要一直进行下去，直到能够提供指导构件构造的足够细节为止

传统观点（构件级设计）

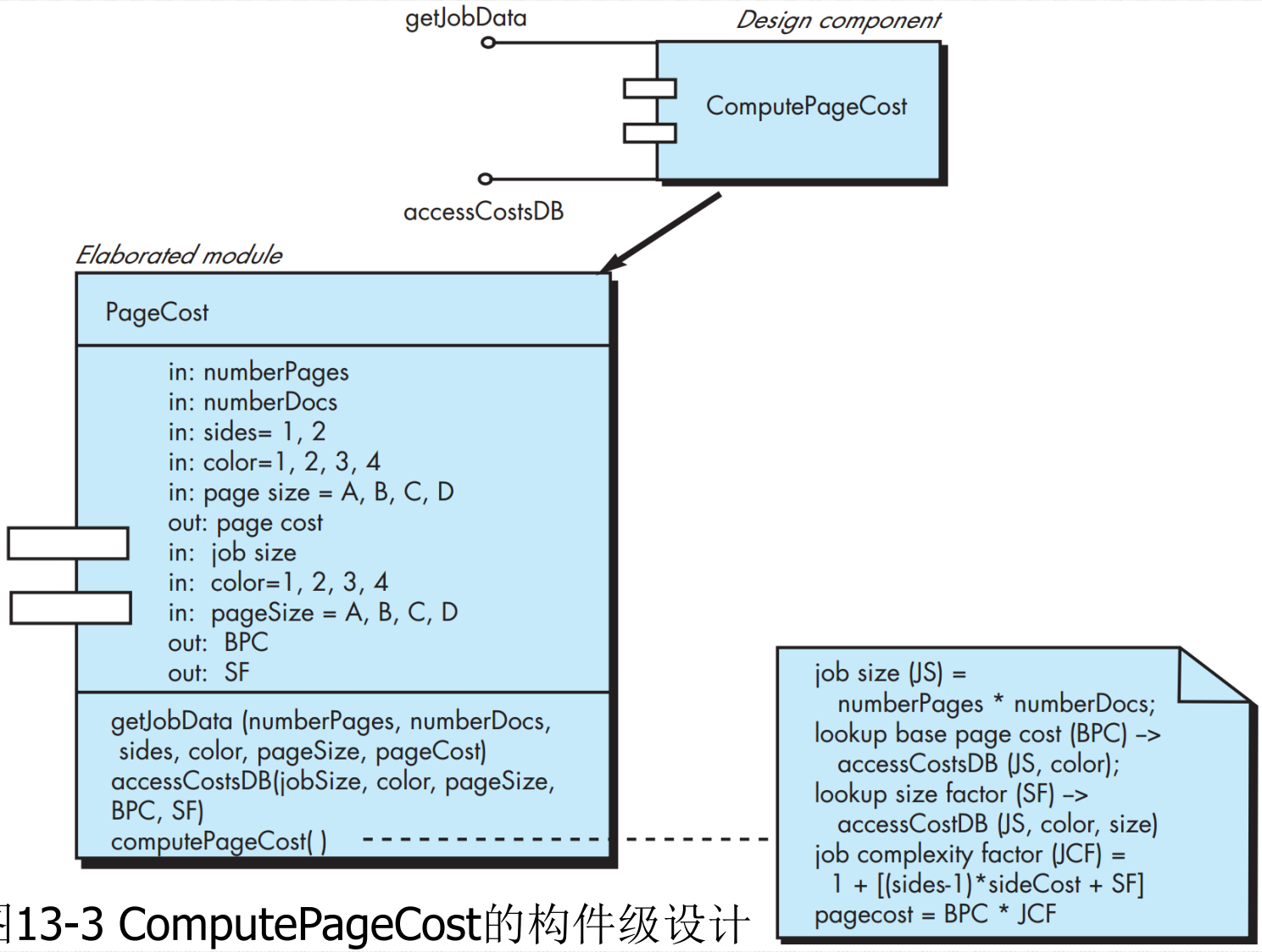


图13-3 ComputePageCost的构件级设计

13.2 设计基于类的构件

- 构件级设计利用了
  - 需求模型开发的信息和
  - 体系结构模型表示的信息。
- OO软件工程中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。
- 构建活动之前所需的设计细节——
  - 类的属性、操作和接口的详细描述

13.2.1 基本设计原则

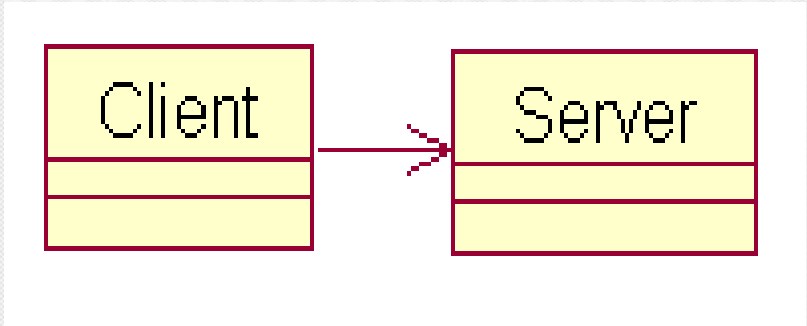
- 有4种构件级设计的基本设计原则被广泛采用
  - 根本动机——产生的设计在发生变更时能够
    - 适应变更并且
    - 减少副作用的传播。
- 开关原则(OCP)、
- Liskov替换原则(LSP)、
- 依赖倒置原则(DIP) 、
- 接口分离原则(ISP)

开闭原则（OCP）

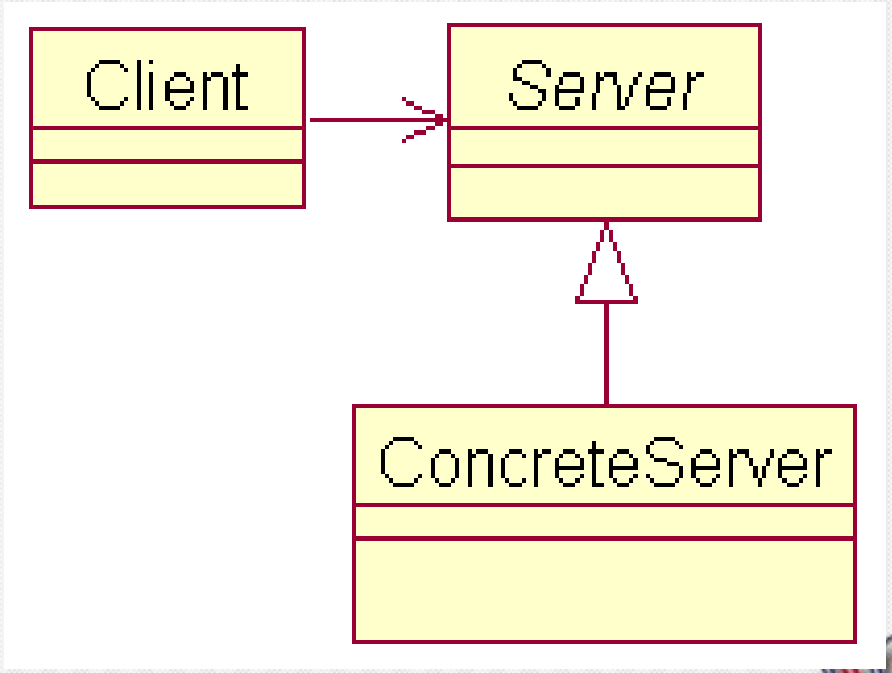
- 对扩展开放，对修改封闭
  - 无需对构件内部做修改（对修改封闭）
  - 以扩展的方式增加功能（对扩展开放）
- 实现的关键
  - 抽象

关键——抽象

- 不符合OCP设计的例子



- 改进的设计



SAFEHOME实例

例如，假设 SafeHome 的安全功能使用了对各种类型安全传感器进行状态检查的 Detector 类。随着时间的推移，安全传感器的类型和数量将会不断增长。如果内部处理逻辑采用一系列 if-then-else 的结构来实现，其中每个这样的结构都负责一个不同的传感器类型，那么对于新增加的传感器类型，就需要增加额外的内部处理逻辑（依然是另外的 if-then-else 结构），而这显然违背 OCP 原则。

图 13-4 中给出了一种遵循 OCP 原则实现 Detector 类的方法。对于各种不同的传感器，Sensor 接口都向 Detector 构件呈现一致的视图。如果要添加新类型的传感器，那么对 Detector 类（构件）无需进行任何改变。这个设计遵守了 OCP 原则。

SAFEHOME实例[51]

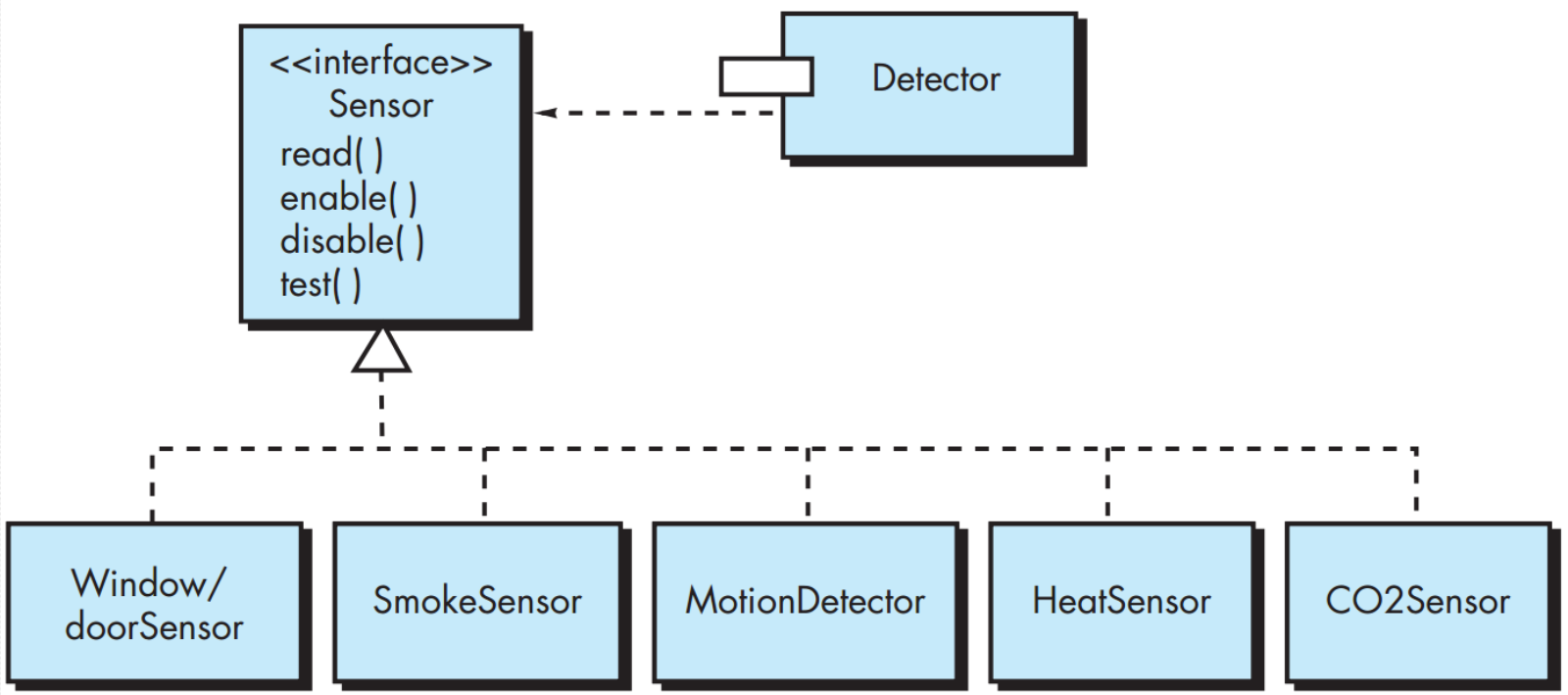


图13-4 遵循OCP原则

Liskov 替换原则LSP

- 滥用继承是一个相当普遍的现象，以为会用继承，会用多态，便很OO
- 在滥用继承是亵渎OO的精髓，知道ISA关系，但还需知道ISA真正的含义
- 是什么设计规则在支配着这种特殊的继承用法？最佳的继承层次的特征？怎样的情况又让我们容易掉进不符合OCP原则的陷阱中

- 定义：
  - Subtypes Must Be Substitutable For Their Base Types
- Barbara Liskov[1988]首先写到
  - 需要如下的替换性质：对于每一个类型*S*的对象*o1*，都存在一个类型*T*的对象*o2*，使得在所有针对*T*编写的程序*P*中，用*o1*替换*o2*后，程序*P*的行为功能不变，则*S*是*T*的子类型
- 违背该原则的后果
  - 如果某个函数使用了指向基类的指针或引用，却违背LSP原则，那么这个函数必须了解该基类的所有派生类，显然违背开闭原则OCP
- 通俗理解
  - 仅当子类能在语义上完全替换基类时，
    - 才能放心地重用那些使用基类的函数和修改派生类型
    - 继承是针对行为而言的

违背LSP原则的例子：

正方形是一个矩形

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

```
void g(Rectangle& r)
{
    r.SetWidth(5);
    r.SetHeight(4);
    assert(r.GetWidth() * r.GetHeight()) == 20);
}
```

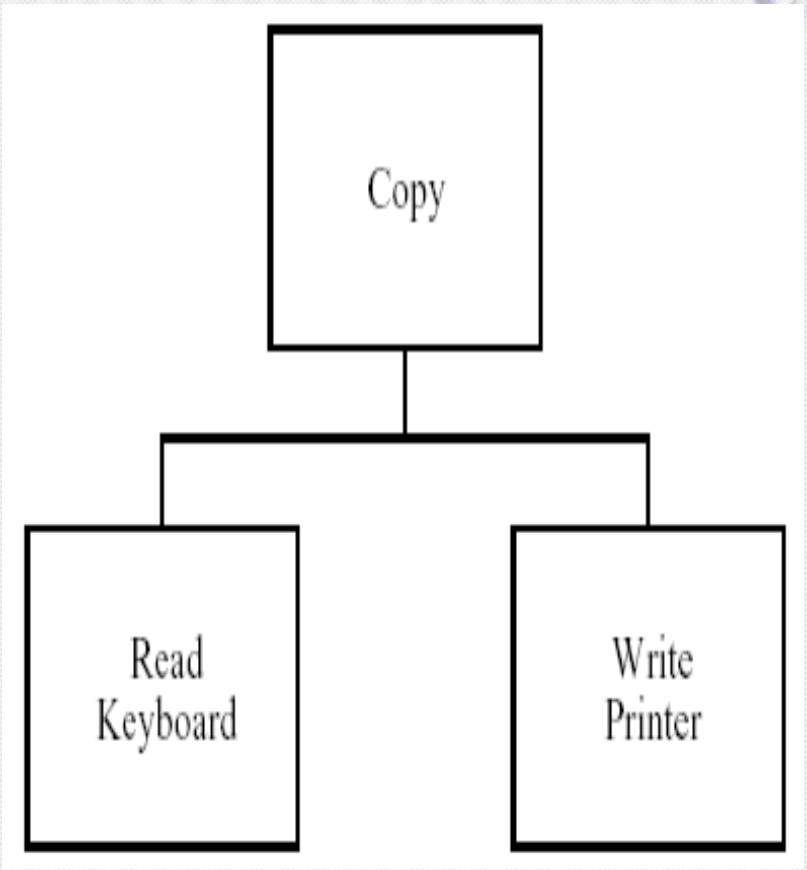
依赖倒置原则(DIP)

- 依赖于抽象，而非具体实现
  - 抽象可较容易地对设计进行扩展，不会导致混乱
  - 构件依赖的具体构件越多，其扩展起来就越困难
- 定义：
  - A. 高层模块不应该依赖于低层模块。二者都应该依赖于抽象。
  - B. 抽象不应该依赖于细节。细节应该依赖于抽象。
- 倒置的含义：
  - 传统软件开发方法，比如结构化分析和设计，
    - 总是倾向于创建高层模块依赖于低层模块、
    - 抽象则依赖于细节的软件结构
  - 设计良好的面向对象的程序的依赖关系结构相对于传统过程式方法设计的通常的结构而言就是被“倒置”

DIP示例

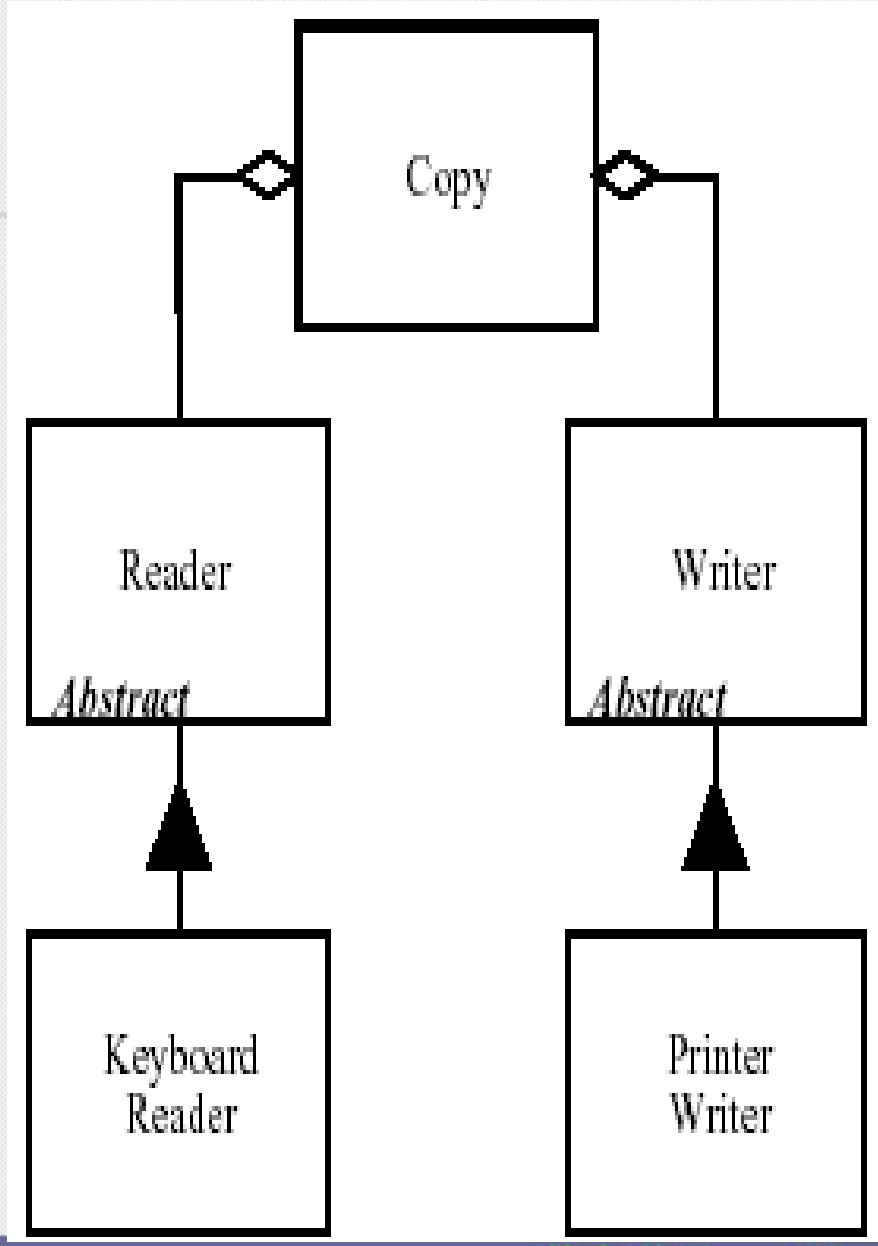
- 从一个例子说起：
  - 一个简单的程序，其任务就是实现将键盘输入的字符拷贝到打印机上

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

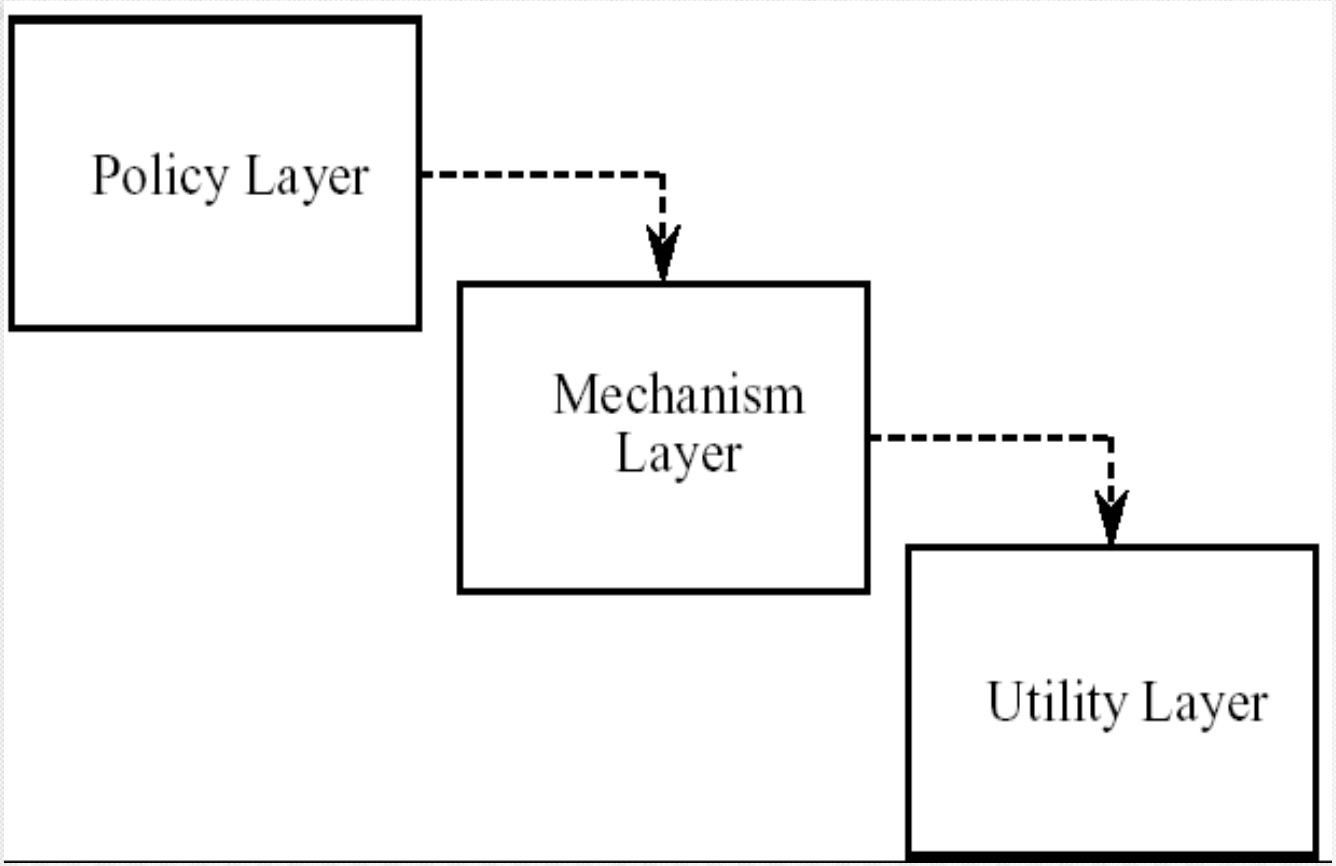


- 依赖倒置
  - 问题是包含高层策略的模块依赖于它所控制的低层模块

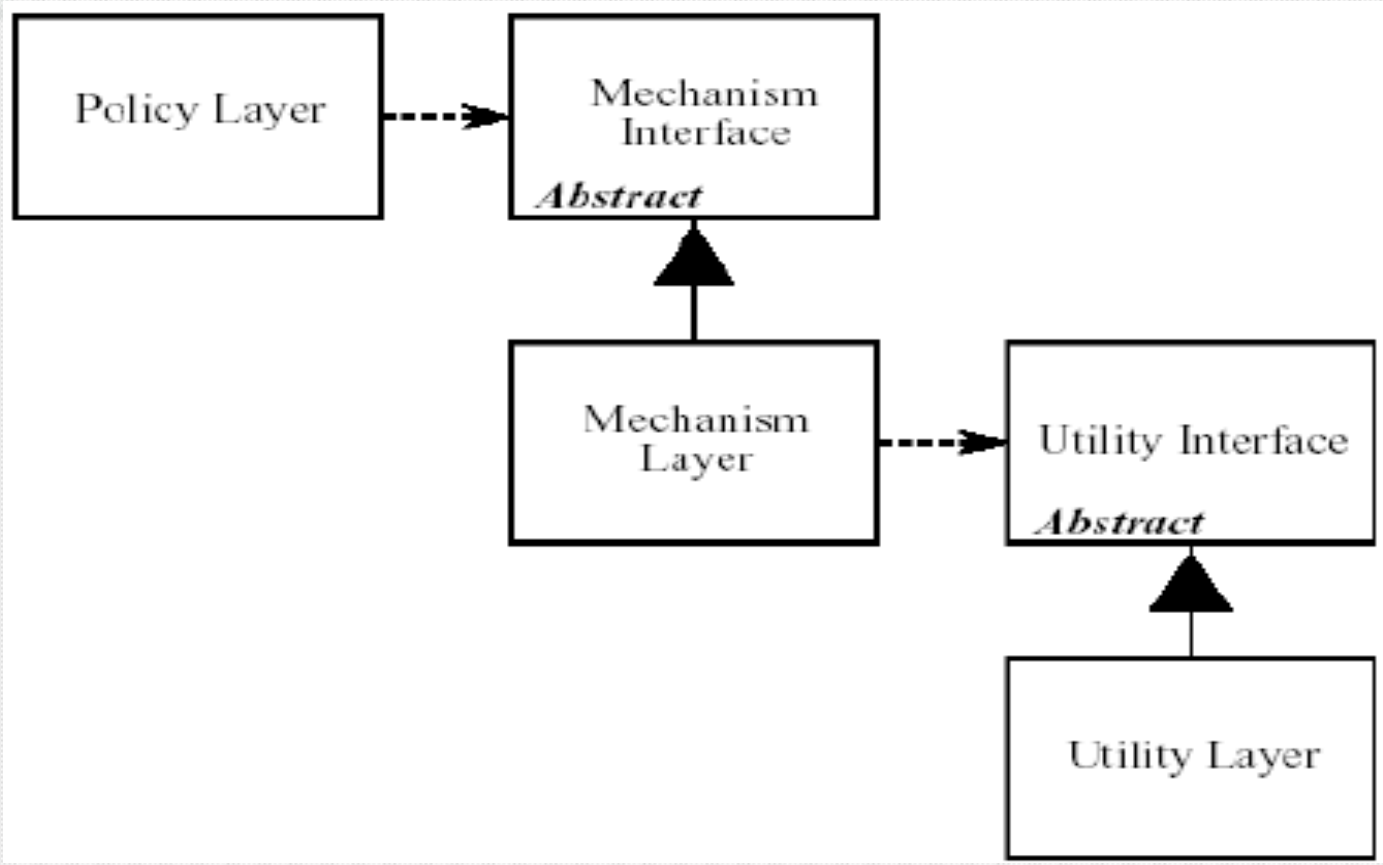
```
class Reader {
public:
    virtual int Read（） = 0;
};
class Writer {
public:
    virtual void Write（char） = 0;
};
void Copy(Reader& r, Writer& w)
{
    int c;
    while((c=r.Read() != EOF)
        w.Write（c）;
}
```



DIP图示



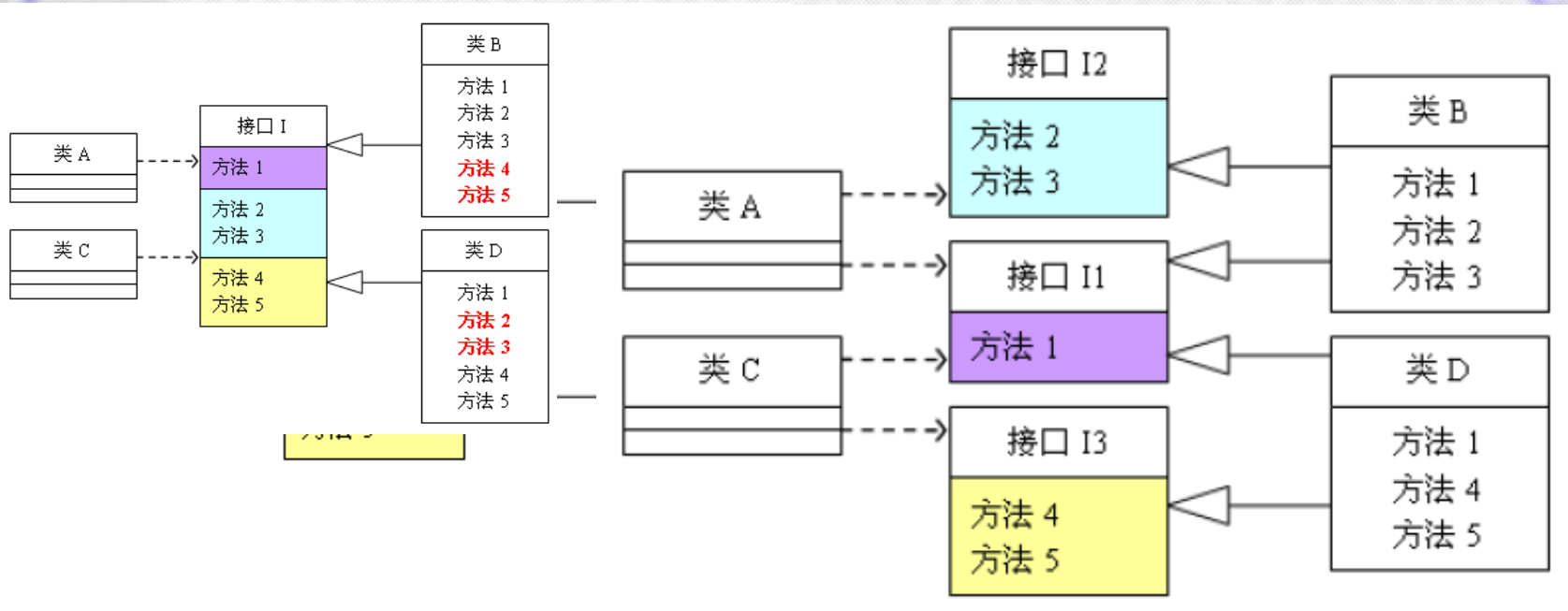
DIP图示


Policy Layer
Mechanism Interface
Abstract
Mechanism Layer
Utility Interface
Abstract
Utility Layer

接口分离原则(ISP)

- “多个用户专用接口比一个通用接口要好”
- 定义：
  - 客户端不应该依赖它不需要的接口
  - 类间的依赖关系应该建立在最小的接口上
- 多客户构件使用一个服务器类提供操作实例很多
- ISP建议为每一个主要客户类型都设计一个特定的接口
  - 只有那些与特定客户类型相关的操作，才应该出现在该客户的接口说明中。
  - 如果多个客户要求相同的操作，则这些操作应该在每一个特定的接口中都加以说明。

ISP示例



A依赖方法1、2、3
B依赖方法1、4、5
B、D中红色是不需要的方法

打包原则

- 发布复用等价性原则（REP）：
  - 复用的粒度就是发布的粒度
- 共同封装原则（CCP）：
  - 一同变更的类应该合在一起
- 共同复用原则（CRP）：
  - 不能一起复用的类不能被分到一组

13.2.2 构件级设计指导方针

- 构件
  - 对那些已经被确定为体系结构模型一部分的构件应该建立命名约定，并对其做进一步的细化和精化，使其成为构件级模型的一部分。
- 接口
  - 接口提供关于通信和协作的重要信息（也可以帮助我们实现OCP原则）。
- 依赖与继承
  - 为了提高可读性，依赖关系是自左向右，继承关系是自底（导出类）向上（基类）。

13.2.3 内聚性

- 传统观点: 构件的专一性，
- OO观点：
  - 内聚性意味着构件或类只封装那些相互关联密切，
  - 以及与构件或类自身有密切关系的属性和操作
- 内聚性的类型
  - 功能内聚：通过操作体现，当一个模块只完成某一组特定操作并返回结果时，就称此模块是功能内聚的。
  - 分层内聚：由包、构件和类来体现。高层能够访问低层的服务，但低层不能访问高层的服务。
  - 通信内聚：访问相同数据的所有操作被定义在一个类中。通常，这些类只着眼于数据的查询、访问和存储。

13.2.4 耦合性

- 传统观点:
  - 一个组件连接到其他组件和外部世界的程度
- 面向对象的观点:
  - 类之间彼此联系程度的一种定性度量
- 耦合程度
  - 内容耦合。当一个构件“暗中修改其他构件的内部数据”时，就会发生这种类型的耦合。这违反了基本设计概念当中的信息隐蔽原则。
  - 控制耦合。当操作A调用操作B，并且向B传递控制标记时，就会发生这种耦合。
  - 外部耦合。当一个构件和基础设施构件进行通信和协作时会发生这种耦合。

13.3 实施构件级设计

- 构件级设计本质上是细化的。
- 必须将分析模型和架构模型中的信息
  - 转化为一种设计表示，
  - 提供了用来指导构建活动的充分信息。

实施构件级设计步骤

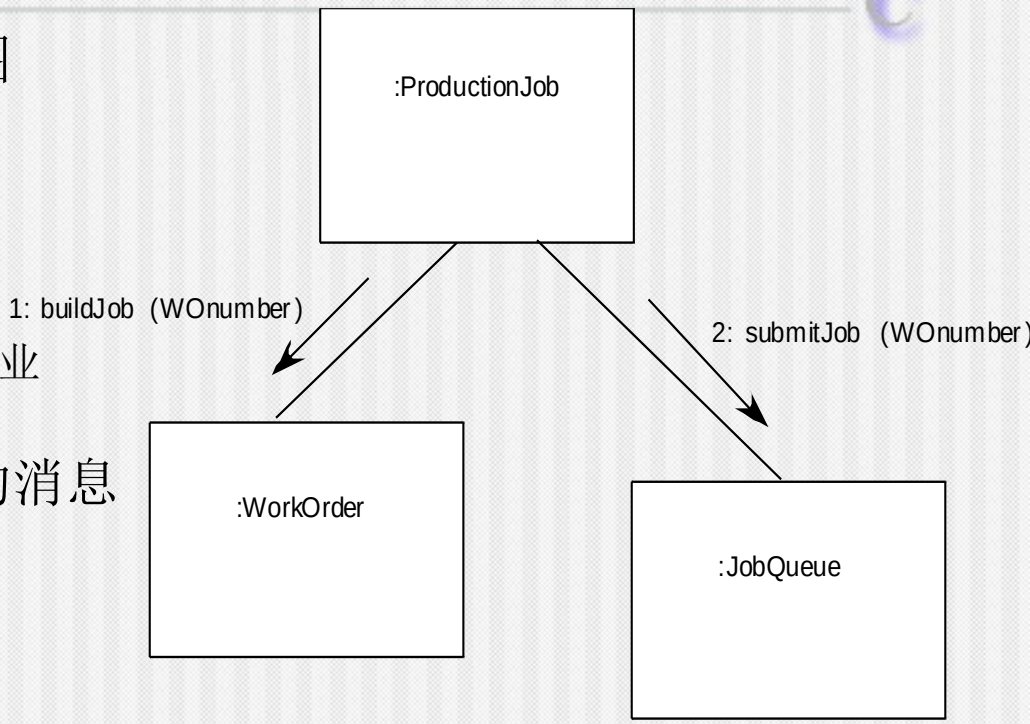
- 步骤1:标识出所有与**问题域**相对应的设计类。
  - 使用需求模型和架构模型，
  - 每个分析类和体系结构构件都要细化。
- 步骤2:确定所有与**基础设施域**相对应的设计类。
  - 在分析模型中并没有描述
  - 在体系结构设计中也经常忽略
  - 但是此时必须对它们进行描述。
  - 通常包括
    - GUI构件、
    - 操作系统构件、
    - 对象和数据管理构件等。

实施构件级设计步骤

- 步骤3:细化所有*不能作为复用构件*的设计类。
  - 详细描述实现类需要的所有接口、属性和操作。
  - 在实现这个任务时，考虑采用前述设计原则。
- 步骤3a:在类或构件的*协作时*说明*消息的细节*。
  - 分析模型中用协作图来显示分析类间的相互协作。
  - 在构件级设计过程中，
    - 某些情况下对系统中对象间传递消息的结构进行说明，
    - 用来表现协作细节
    - 可作为接口规格说明的前提，
    - 这些接口显示了系统中构件通信和协作的方式。

步骤3a示例

- 印刷系统的一个简单协作图
- 三个对象
  - ProductionJob、
  - WorkOrder和
  - JobQueue
  - 相互协作为生产线准备印刷作业
- 图中箭头表示对象间传递的消息
- 消息格式



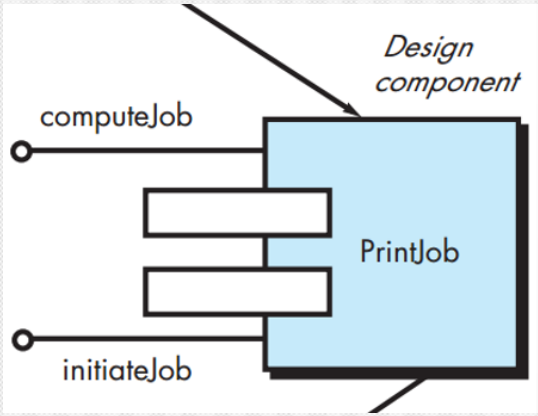
[guard condition] sequence expression (return value) :=
message name (argument list)

实施构件级设计步骤

- 步骤3b:为每一个构件确定适当的*接口*。
  - 【UML】一组外部可见的（即公共的）操作。
  - 接口不包括内部结构，没有属性，没有关联……。
  - 接口是某个抽象类的等价物，
    - 该抽象类提供了设计类之间的可控连接。
    - 为设计类定义的操作可以归结为一个或者更多的抽象类。
    - 抽象类内的每个操作应该是内聚的，
    - 即它应该展示那些关注于一个有限功能或者子功能的处理。
  - 参见ISP

步骤3b示例

- 图13-1中initiateJob接口没有展现出足够的内聚性而受到争议。
- 实际上，它完成三个不同的子功能：
  - 建立工作单，
  - 检查任务的优先级，
  - 并将任务传递给生产线。
- 接口设计应该重构。
  - 重新检查设计类并
  - 定义一个新类***WorkOrder***，
  - 用来处理与装配工作单相关的所有活动。
  - buildWorkOrder()操作成为该类的一部分。
  - 另外定义包括操作checkPriority()的***JobQueue***类。
  - ProductionJob类包括传递给生产线的生产任务的所有相关信息。
  - initiateJob接口将采用图13-7所示的形式。
  - initiateJob现在是内聚的，集中在一个功能上。
  - 与ProductionJob、WorkOrder和JobQueue相关的接口几乎都是专一的。



为PrintJob重构接口和类定义

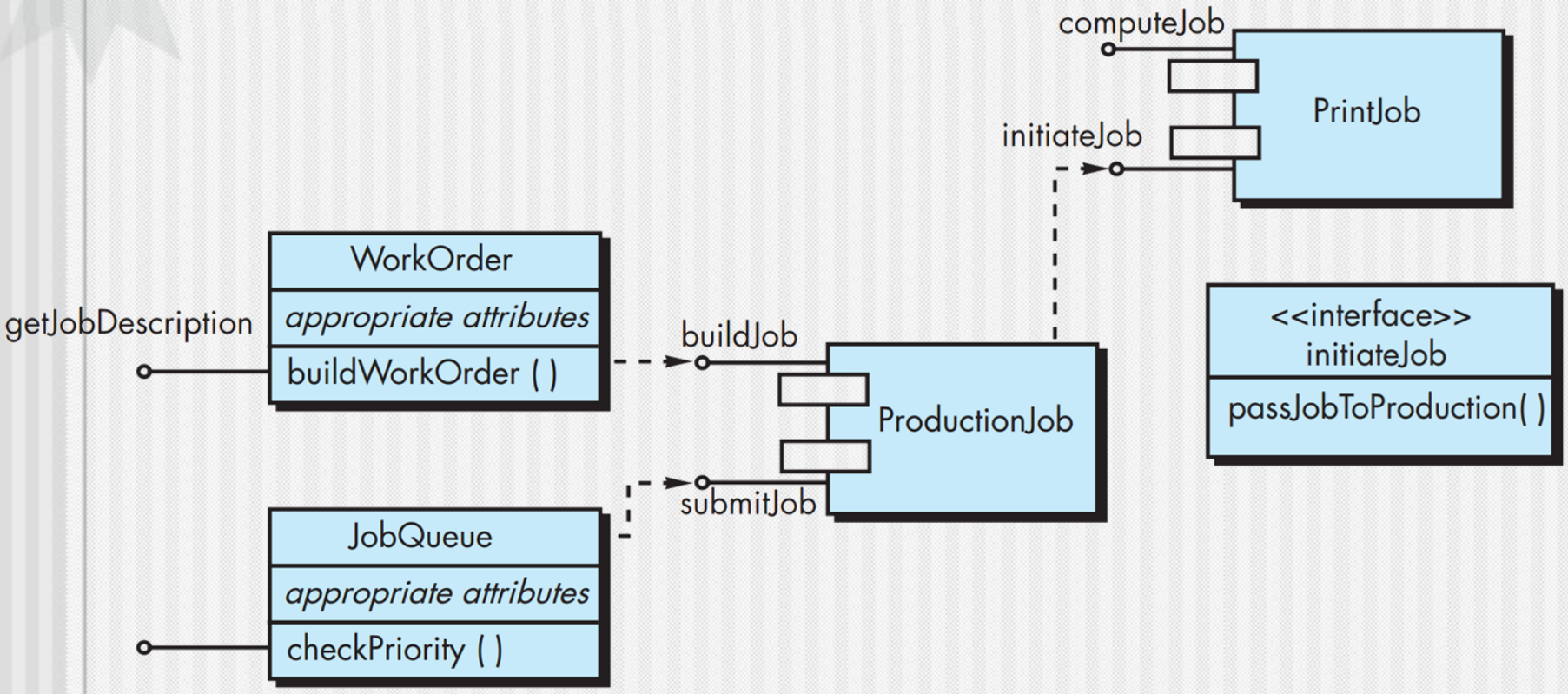


图13-7

实施构件级设计步骤

- 步骤3c: **细化属性**并且定义相应的数据类型和数据结构。
- UML采用下面的语法来定义属性的数据类型：

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

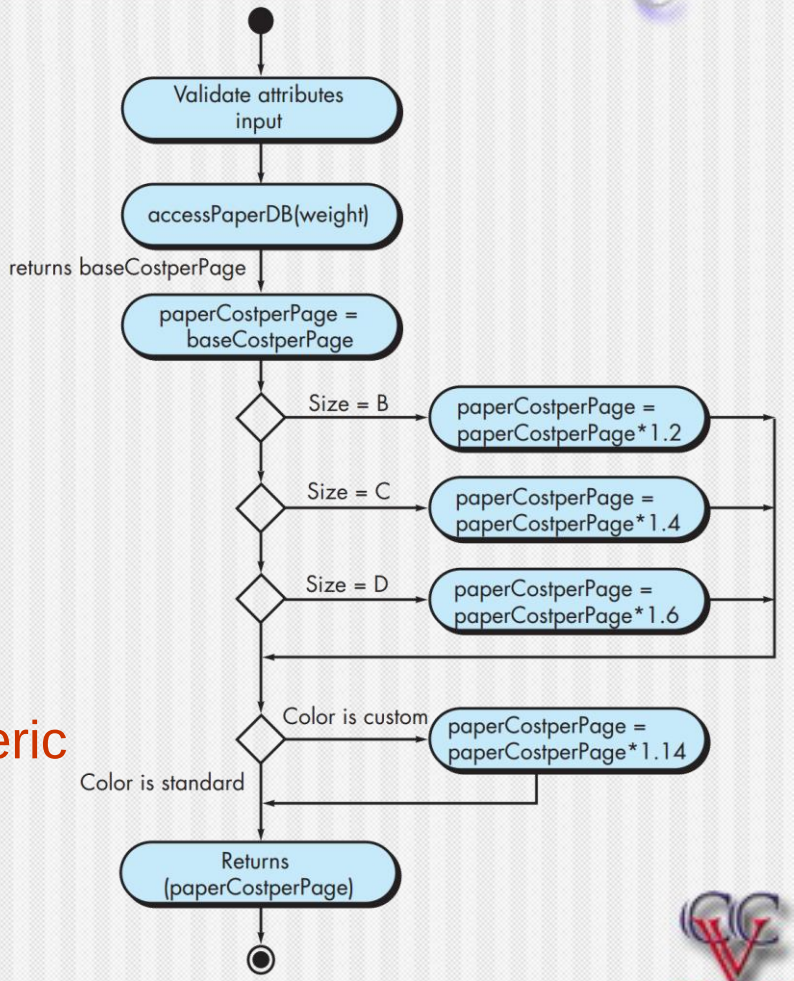
  - name是属性名，
  - type expression是数据类型；
  - initial-value是创建对象时属性的初始值；
  - property string用于定义属性的特征或特性。

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

实施构件级设计步骤

- 步骤3d:详细描述每个操作中的处理流。
  - 可以用流程图、伪代码或者UML活动图来完成。
  - 每个软件构件都逐步求精通过大量的迭代进行细化。
- 操作computePaperCost()可扩展如下:

computePaperCost(weight,size,color):numeric

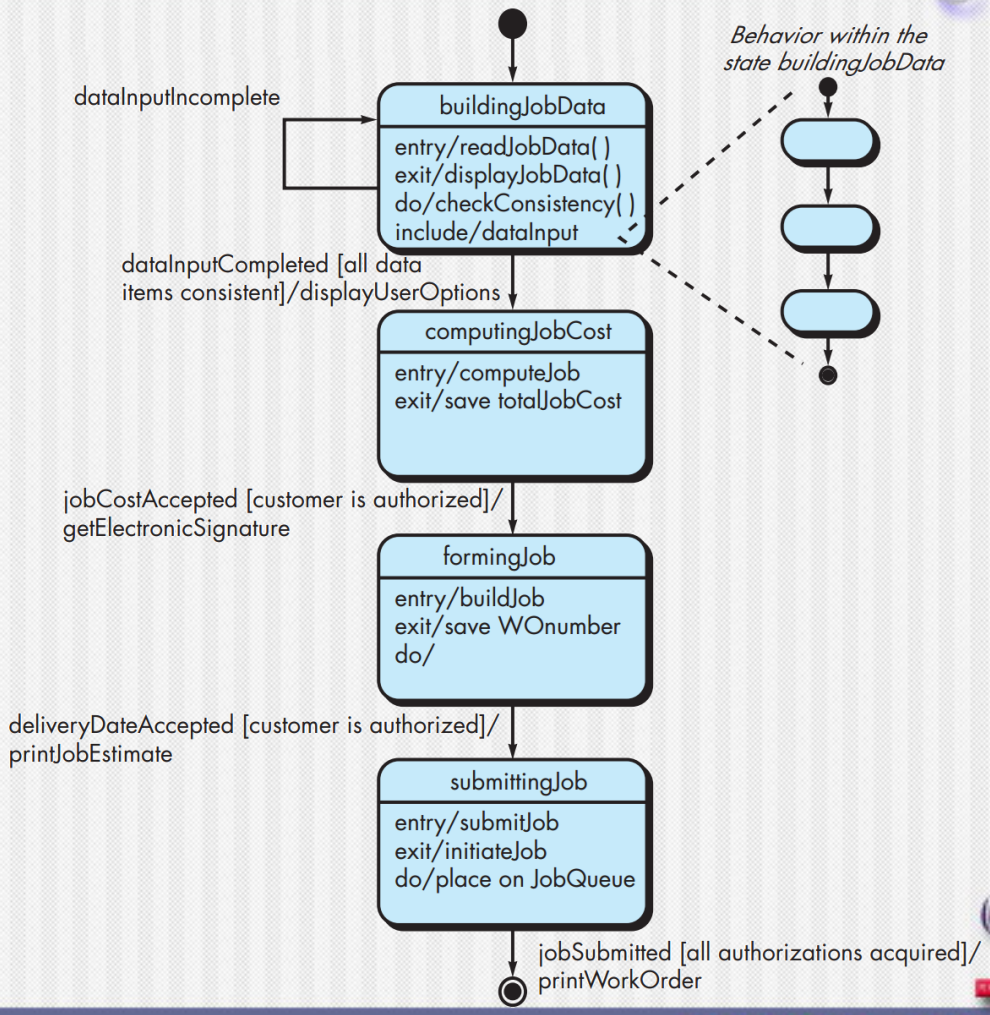


实施构件级设计步骤

- 步骤4:（持久化）
  - 说明持久数据源(数据库和文件)并
  - 确定管理数据源所需要的类。
- 持久数据存储起初都被指定为体系结构设计的一部分，
- 随着设计细化过程的不断深入，
- 提供持久数据源的结构和组织等额外细节常常是有用的。

实施构件级设计步骤

- 步骤5:开发并且细化类或构件的*行为表示*。
  - UML状态图被用作分析模型的一部分，
  - 表示系统的外部可观察的行为和更多分析类的对象的局部行为。
  - 在构件级设计过程中，有时对设计类的行为建模是必要的。
  - 行为模型经常包含其他设计模型中不明显的信息



实施构件级设计步骤

- 步骤6:细化部署图以提供额外的实现细节。
- 步骤7:
  - 考虑每一个构件级设计表示，
  - 并且时刻考虑其他选择。

13.5 设计传统构件

- 处理逻辑的设计是由算法设计和结构化程序设计的基本原则规定
- 数据结构的设计是通过为系统开发制定的数据模型来定义的
- 界面设计是由一个构件必须实现的协作来决定的

13.6 基于构件的开发

- 在软件工程领域，复用既是老概念，也是新概念
  - 早期，程序员就已经复用概念、抽象和过程，
  - 但是早期的复用更像是一种临时的方法。
  - 今天复杂的、高质量的软件系统要在短时间内开发完成
    - 就需要一种更系统的、更有组织性的复用方法
- 基于构件的软件工程是
  - 一种强调使用可复用的软件构件来
  - 设计与构造计算机系统的过程。

谢谢！