软件体系结构建模方法

lliao@seu.edu.cn

目　录

第2章 软件体系结构建模

（一）软件体系结构建模概述
（二） “4 + 1”视图模型
（三）软件体系结构的核心模型
（四）软件体系结构的生命周期模型
（五）软件体系结构抽象模型

一、软件体系结构建模概述

- 1. 软件模型
  - SA的首要问题是如何表示SA，即如何对SA建模。
  - **模型**：一般是指客观世界中存在事物的一种抽象。一般都需要通过某种形式表达出来。
    - 如：数学模型，自然语言或图形语言表达的模型。
  - **软件模型**：是软件的一种抽象，目前一般通过非数学模型来描述。

一、软件体系结构建模概述

- 1. 软件模型

  根据建模的侧重点不同，可以将SA模型分为五种：
  - 结构模型
  - 框架模型
  - 动态模型
  - 过程模型
  - 功能模型

一、软件体系结构建模概述

1. 软件模型

（1）结构模型：用组件，连接件和其他概念来刻画结构，力图**通过结构来反映系统的重要语义内容**，包括系统的配置、约束、隐含的假设条件、风格、性质等。研究结构模型的核心就是ADL;

（2）框架模型：与结构模型类似，但它侧重于整体的结构。它主要以一些特殊问题为目标建立只针对和适应该问题的结构。如应用广泛的MVC（模型-视图-控制器）框架模型解决了用户界面与业务实现相分离的问题。

（3）动态模型：是对结构模型和框架模型的补充,研究系统的“大颗粒”的行为性质。例如,描述系统的重新配置和演化。动态可以指系统总体结构的配置、建立或撤除通信通道或计算的过程。

DARWIN EXAMPLE

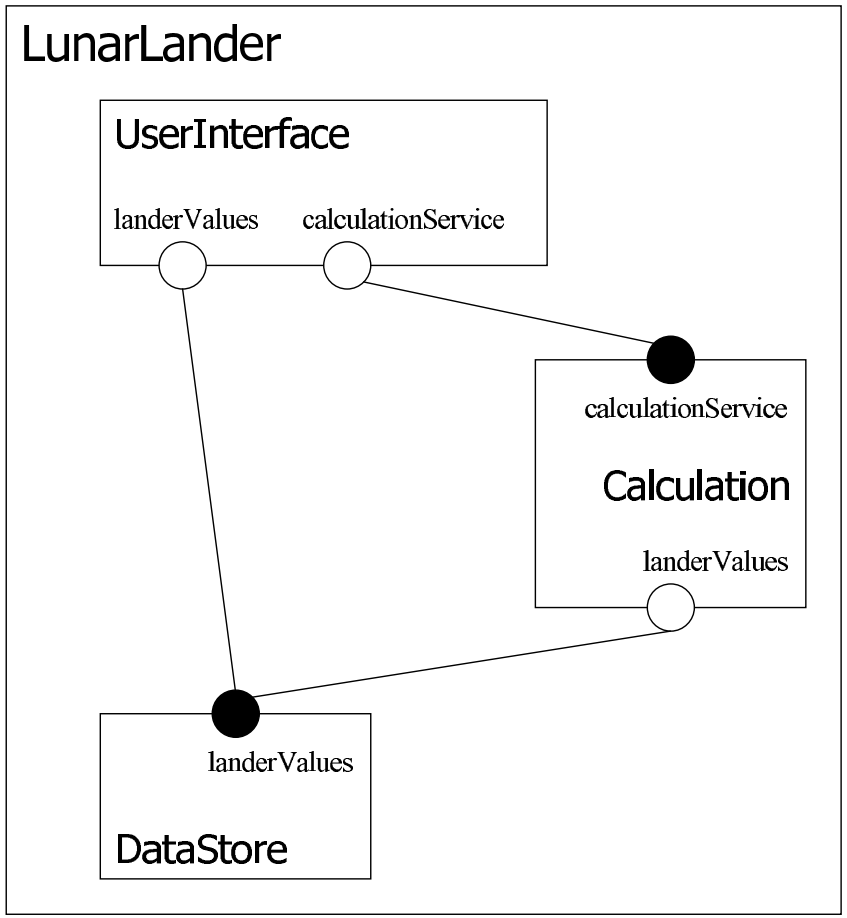
```
component DataStore{
    provide landerValues;
}

component Calculation{
    require landerValues;
    provide calculationService;
}

component UserInterface{
    require calculationService;
    require landerValues;
}

component LunarLander{
inst
    U: UserInterface;
    C: Calculation;
    D: DataStore;
bind
    C.landerValues -- D.landerV
    U.landerValues -- D.landerV
    U.calculationService -- C.calculationService;
}
```

Canonical Textual Visualization



Graphical Visualization

BACK

一、软件体系结构建模概述

1. 软件模型

（4）过程模型：研究构造系统的步骤和过程，因而结构是遵循某些过程脚本的结果。

（5）功能模型：该模型认为SA是由一组功能组件按层次组成，下层向上层提供服务。

这5种模型各有所长，将5种模型有机地统一在一起，形成一个完整的模型来刻画软件体系结构更合适。

二、“4＋1”视图模型

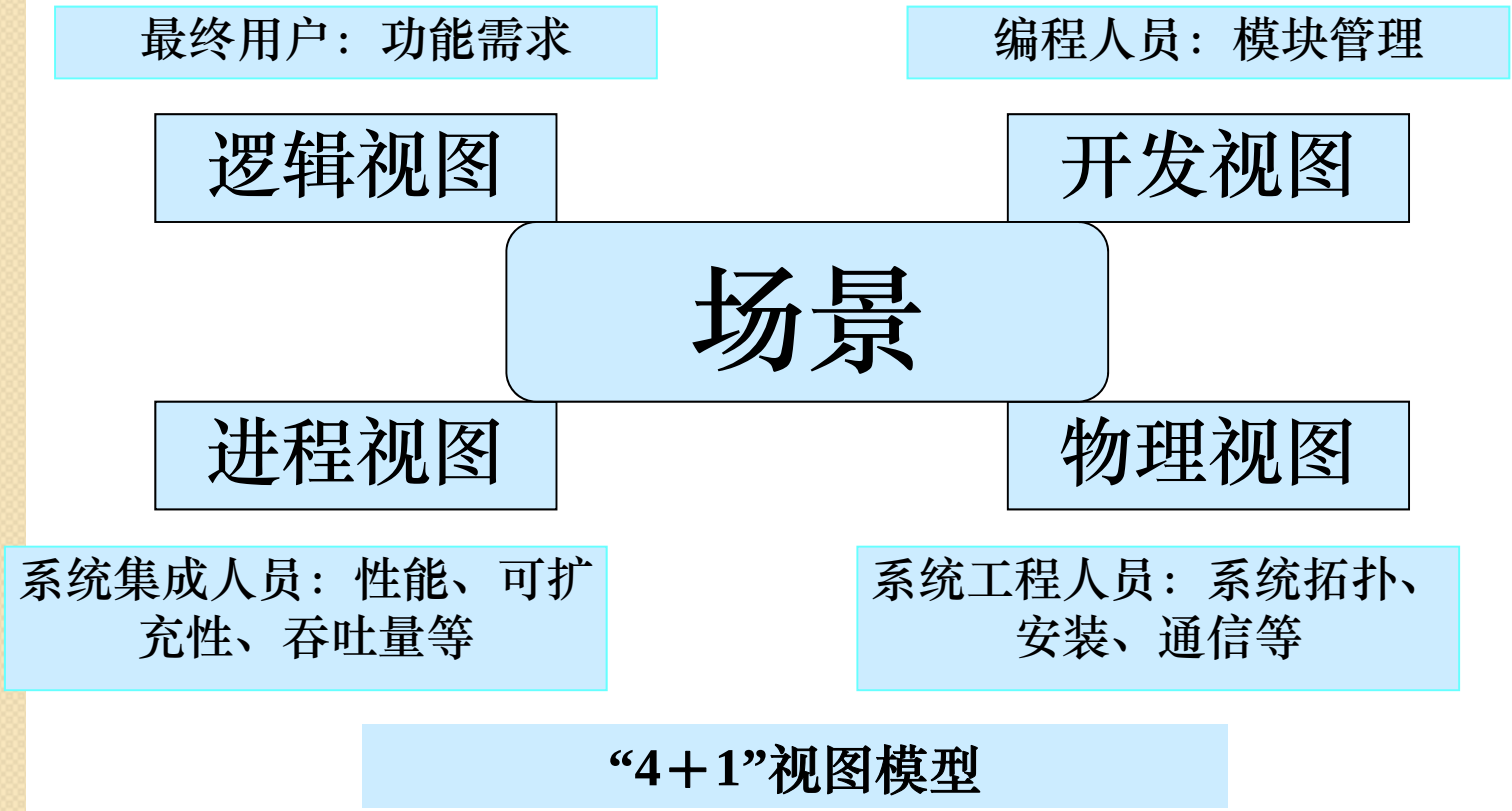
- Philippe Kruchten (Rational Software Corp.)于1995年在他的论文“**Architectural Blueprints—The “4+1” View Model of Software Architecture**”提出了软件体系结构的“4 + 1”视图模型.

- 该模型使用体系结构为中心、场景驱动、迭代的开发方法来设计。

二、“4＋1”视图模型

- “4+1”视图模型从5个不同视角来描述软件体系结构，每个视图只关心系统的一个侧面，5个视图结合在一起才能反映系统的软件体系结构的全部内容

二、“4＋1”视图模型


最终用户：功能需求
编程人员：模块管理
逻辑视图
开发视图
场景
进程视图
物理视图
系统集成人员：性能、可扩充性、吞吐量等
系统工程人员：系统拓扑、安装、通信等
“4＋1”视图模型

二、“4＋1”视图模型

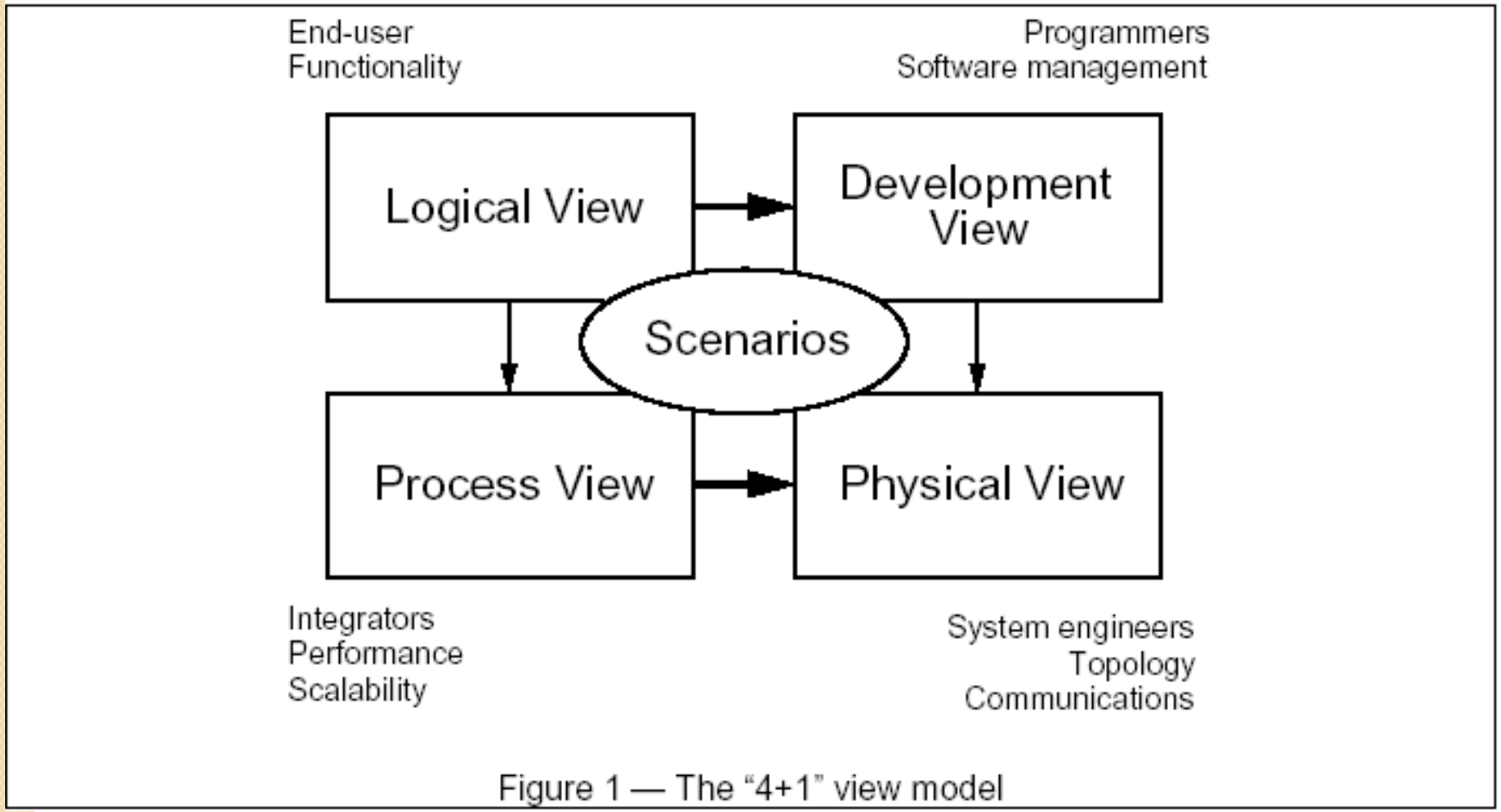


Figure 1 — The “4+1” view model

二、“4＋1”视图模型

- The **logical view**, which is the object model of the design (when an object-oriented design method is used),
- the **process view**, which captures the concurrency and synchronization aspects of the design,
- the **physical view**, which describes the mapping(s) of the software onto the hardware and reflects its distributed aspect,
- the **development view**, which describes the static organization of the software in its development environment.

二、“4＋1”视图模型

- The description of an architecture—the decisions made—can be organized around these four views, and then illustrated by a few selected **use cases**, or **scenarios** which become a **fifth view**. The architecture is in fact partially evolved from these scenarios as we will see later.

逻辑视图

- 逻辑视图主要支持系统功能需求,即系统提供最终用户的服务
- 在逻辑视图中,系统分解成一系列的功能抽象,这些抽象主要来自问题领域.
- 在面向对象技术中，通过抽象、封装和继承，**可以用对象模型来代表逻辑视图，用类图来描述逻辑视图。**

逻辑视图…

（I）逻辑视图的标记方法—Booch标记法

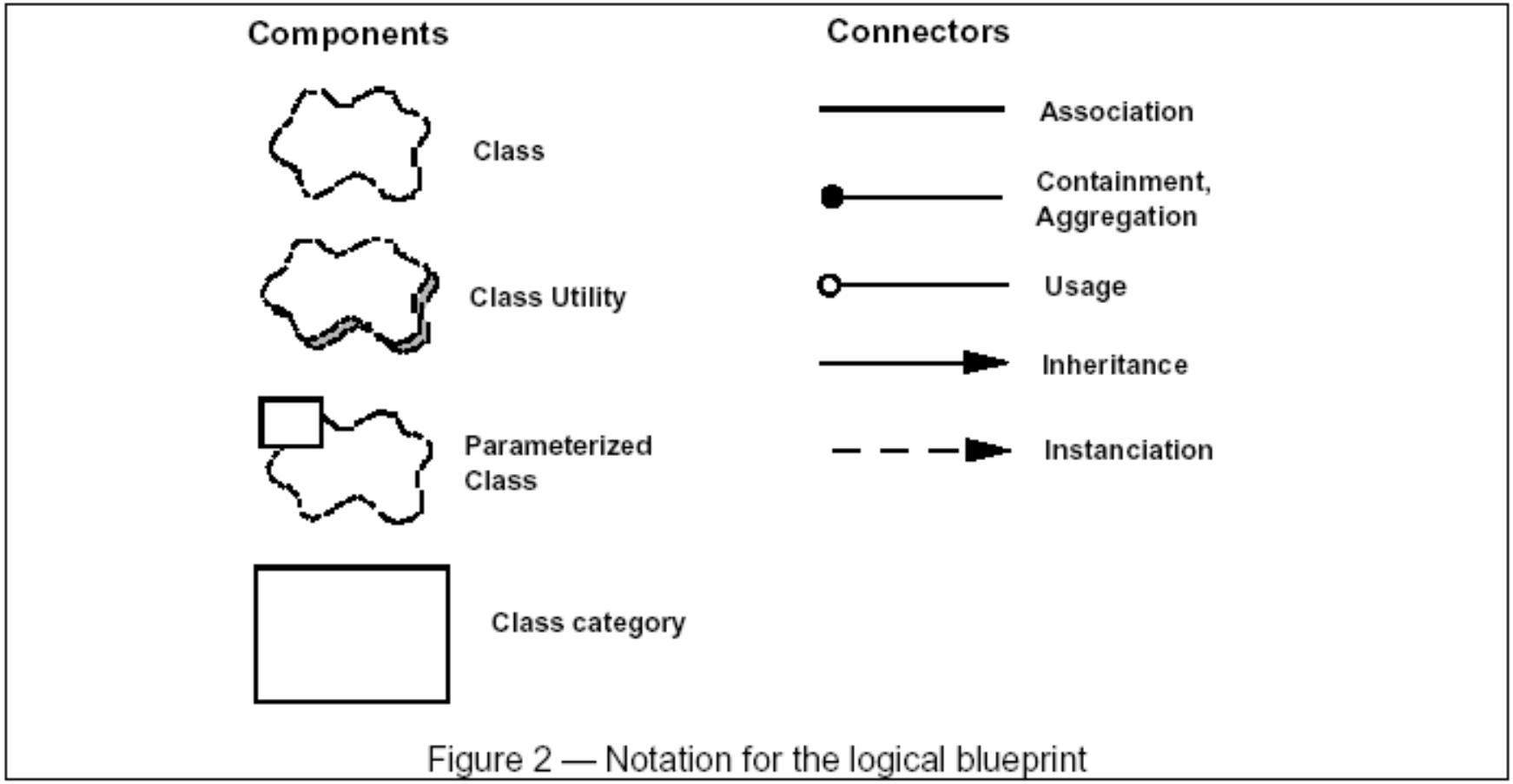
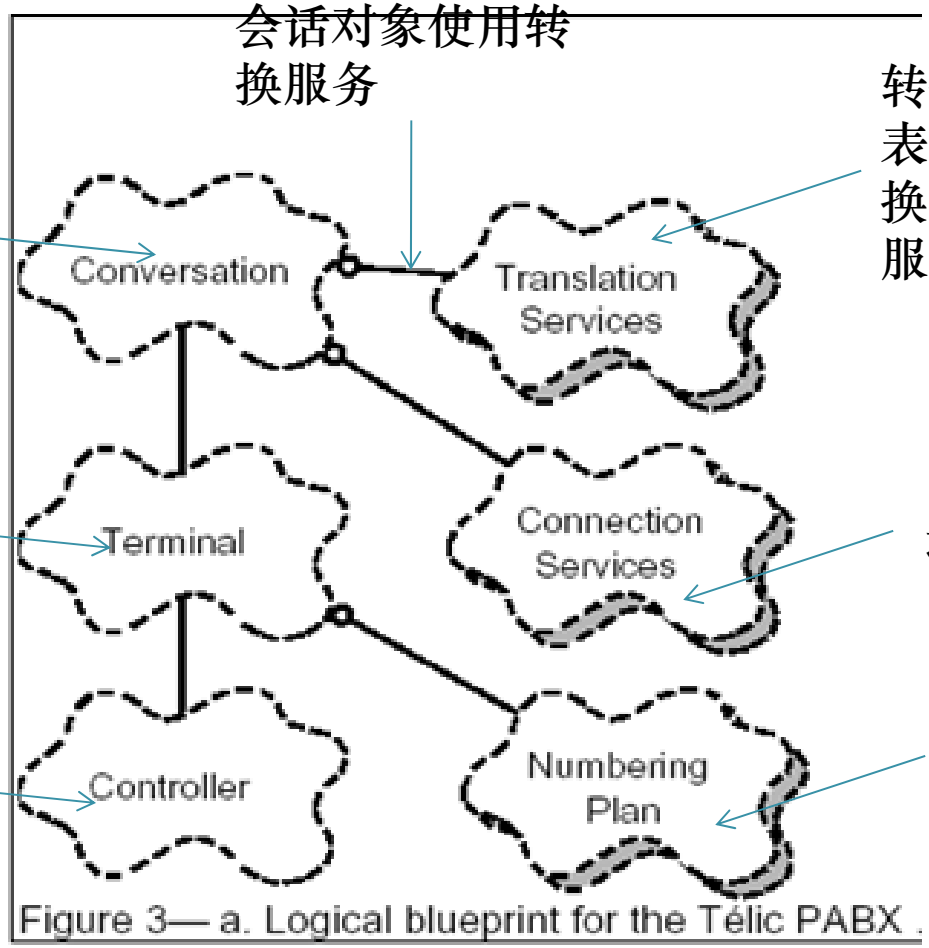
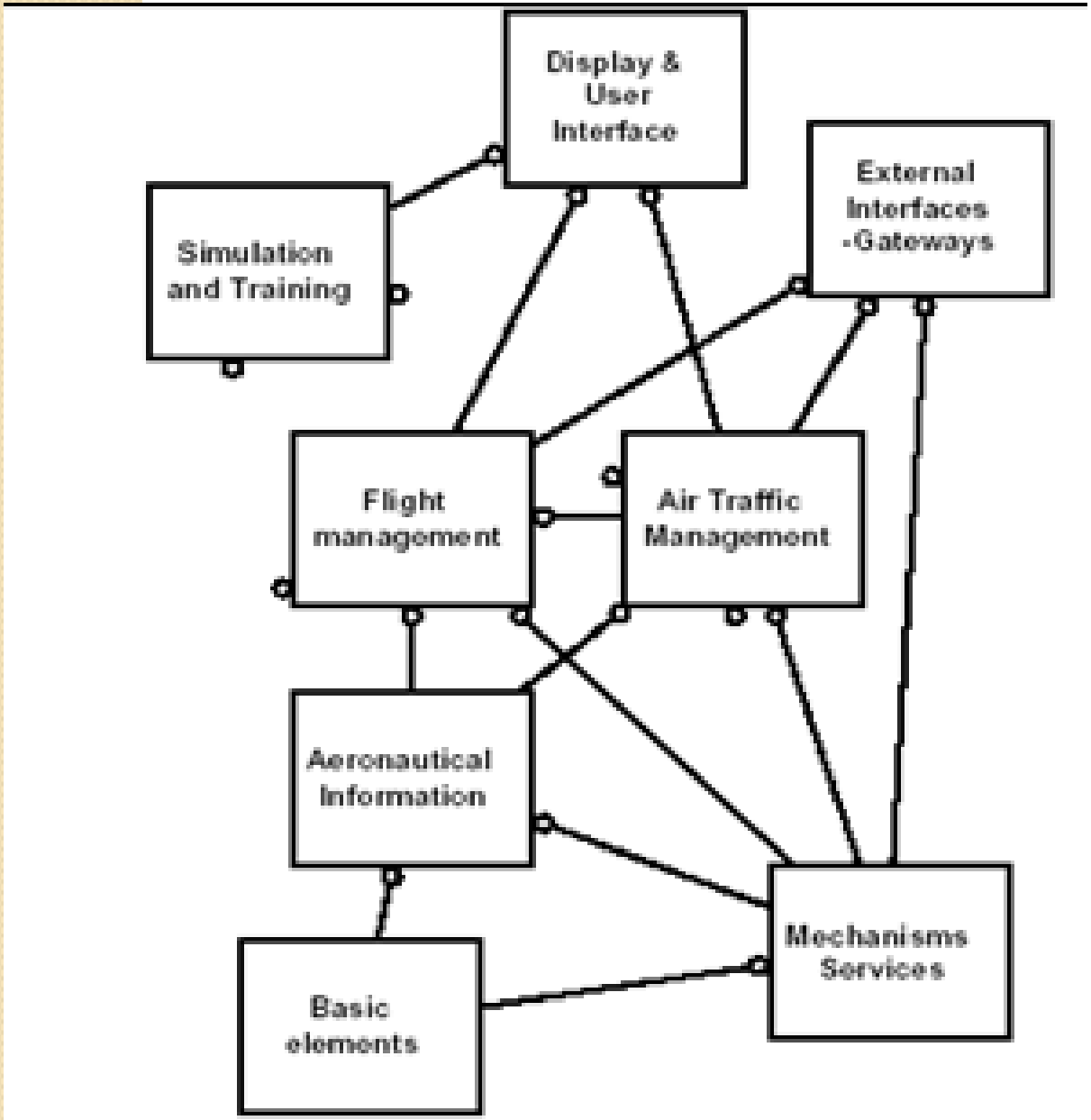


Figure 2 — Notation for the logical blueprint

某通信系统ACS体系结构





. b. Blueprint for an Air Traffic Control System

对于规模更大的系统来说，体系结构中包含数十甚至数百个类，可以用类层次图来表示。

左图描述了空中交通管理系统的顶层类图

逻辑视图

- （1）逻辑视图的标记方法—Booch标记法
- （2）逻辑视图的风格—面向对象的风格，其主要的设计准则是试图在整个系统中保持单一的、连贯的对象模型。

开发视图

- 开发视图（development view）也称为模块视图（module view），关注软件开发环境下实际模块的组织。
- 软件可以打包成小的程序块（程序库或子系统），可以由一位或几位开发人员来开发。
- 系统的开发视图可以用模块和子系统图来表达

开发视图

- 开发视图主要侧重于软件模块的组织和管理。
- 开发视图的表示：

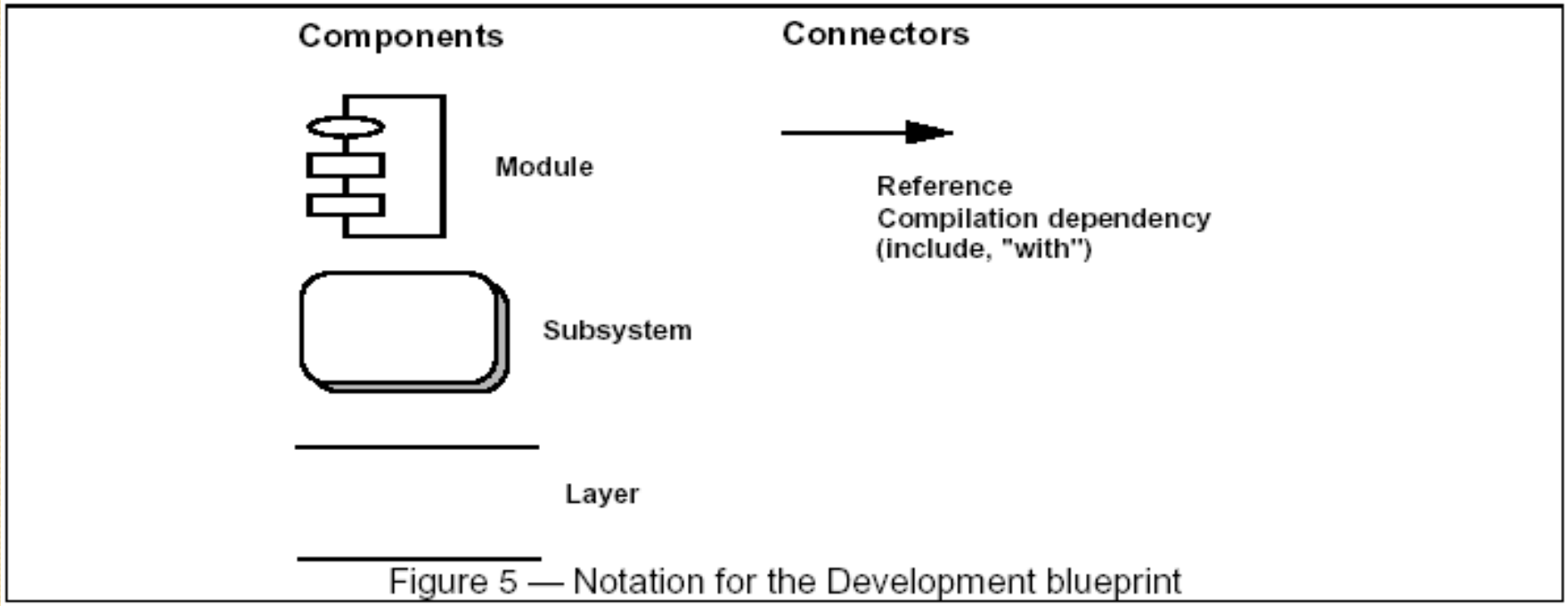


Figure 5 — Notation for the Development blueprint

开发视图…

- 开发视图的风格

  使用层次式风格，定义4或6层子系统。每层均有良好的职责、设计规则，即某层子系统依靠同层次的或更低层次的子系统，从而减少了具有复杂模块依赖关系的网路开发量，得到层次式的简单发布策略。

  各个层次，层次越低，通用性越强。以保证需求变更时，所做的改动最小。

开发视图…

空中交通管理（ATC）系统的分层开发视图

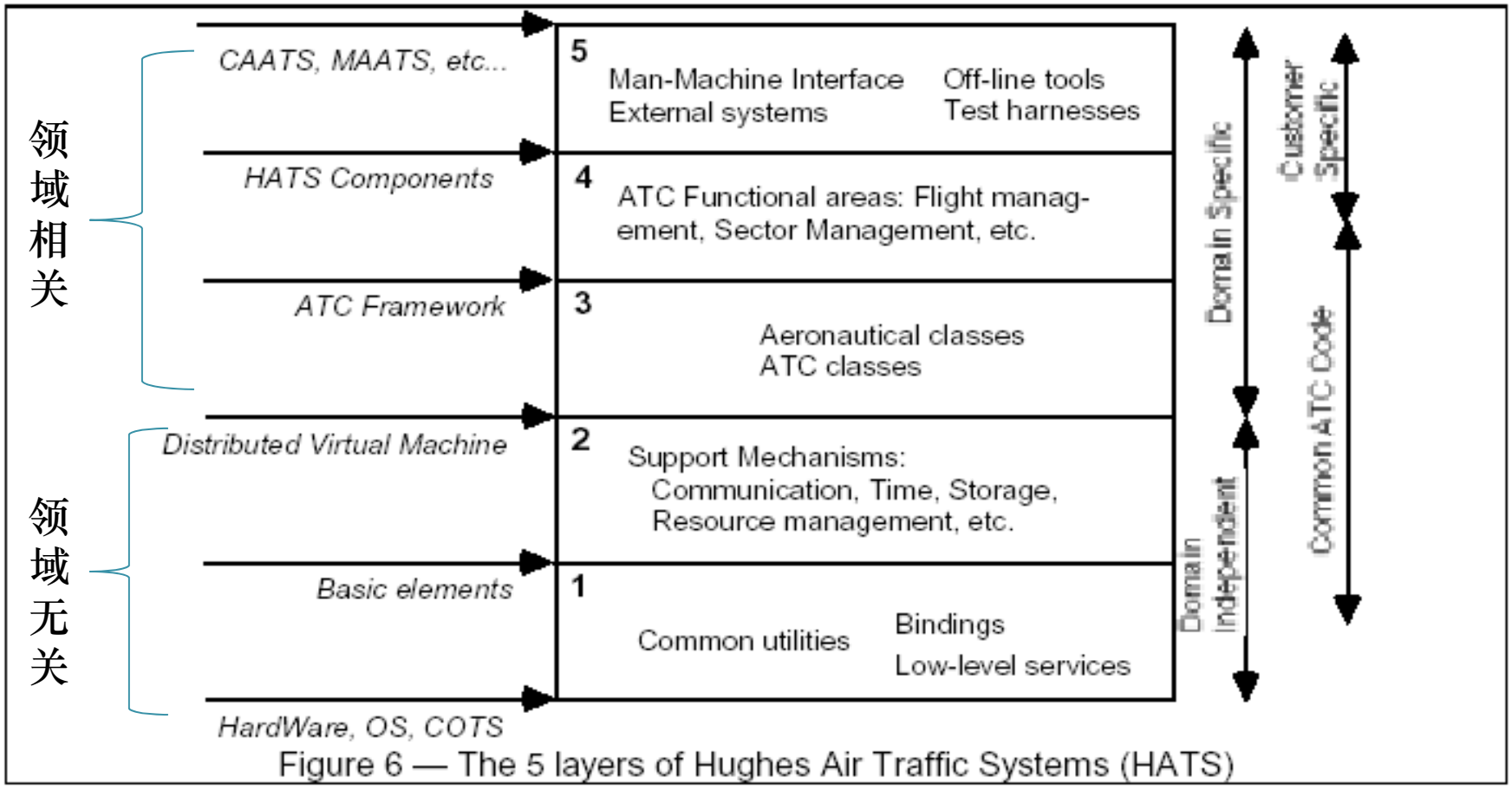


Figure 6 — The 5 layers of Hughes Air Traffic Systems (HATS)

开发视图...

- 开发结构示例

  图－6是图－3（b）（ P18 ）的开发视图。第1层和第2层组成了一个领域无关的分布式基础设施，贯穿于整个产品线中，并且与硬件平台、操作系统或数据库管理系统等无关。第3层增加了空中交通管制系统的框架，以形成一个领域特定的软件体系结构。第4层使用该框架建立一个功能平台，第5层则依赖于具体客户和产品，包含了大部分用户接口以及与外部系统的接口。

进程视图

- 进程视图（process view）也称为并发视图，侧重于系统的**运行特性**，主要关注一些非功能性的需求，例如系统的性能和可用性。
- **进程视图强调并发性、分布性、系统集成性和容错能力**，以及从逻辑视图中的主要抽象如何适合进程结构—即在哪个控制流程上，对象的操作被真正地执行。

进程视图…

（I）进程视图的标记方法

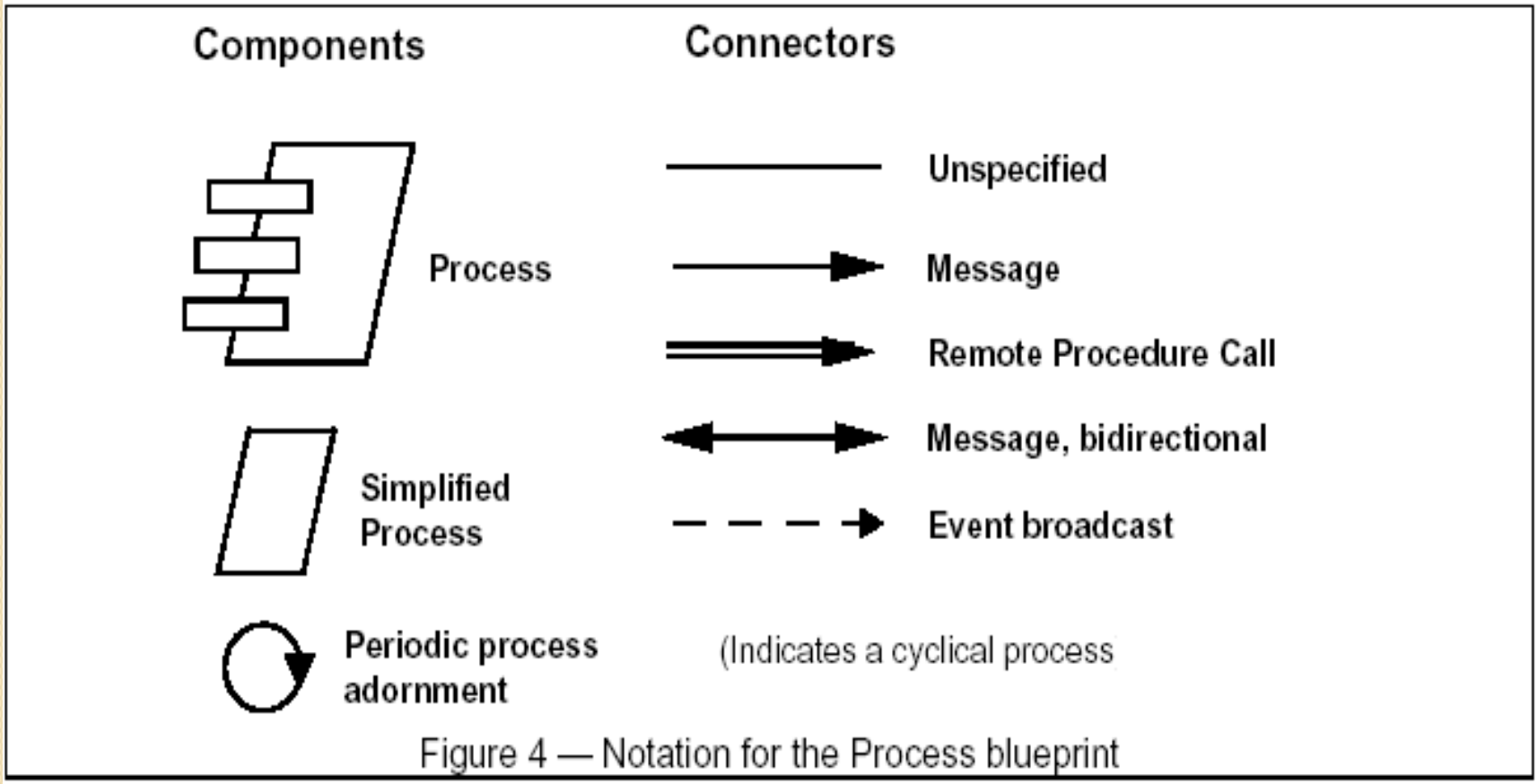
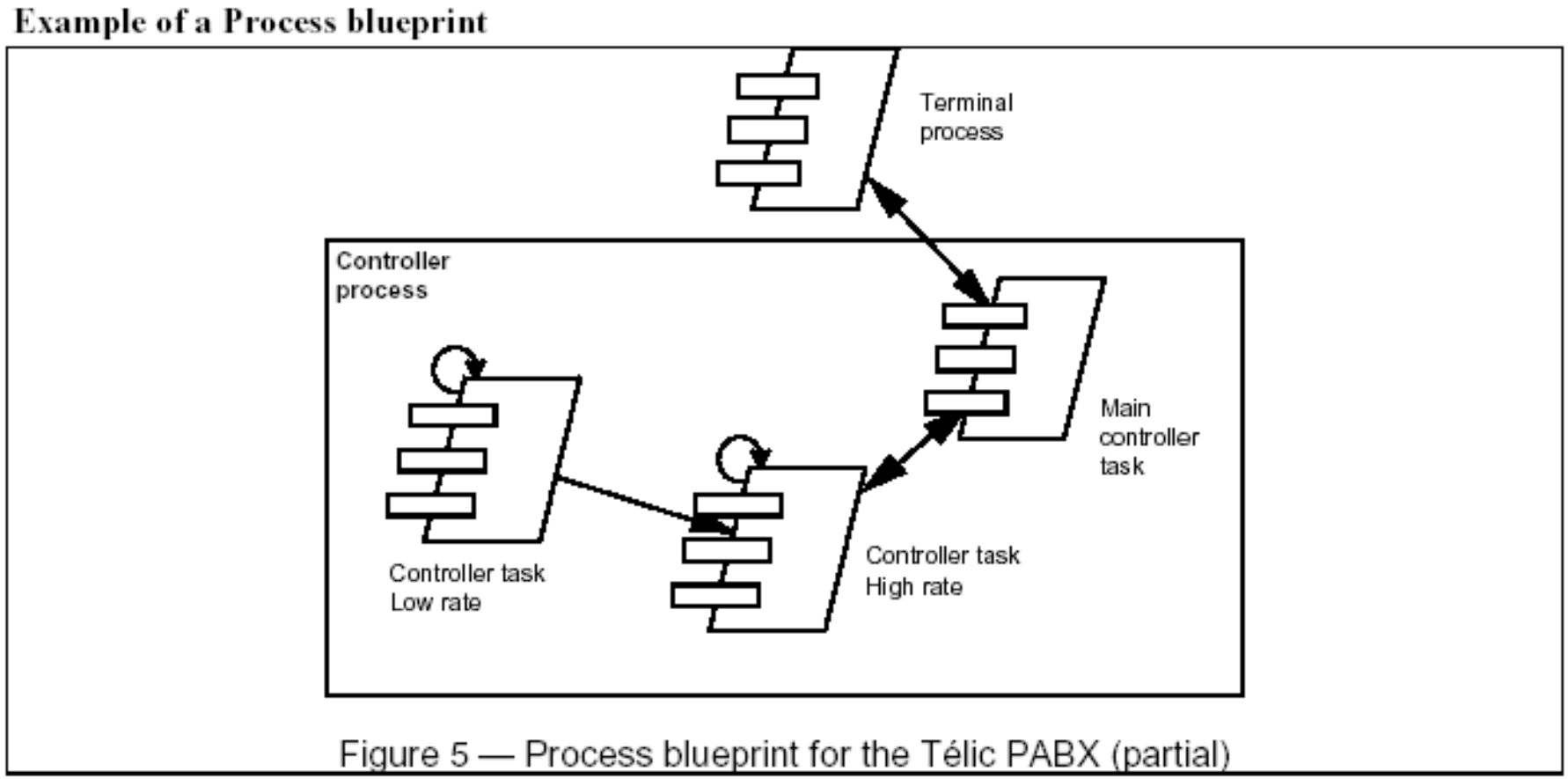


Figure 4 — Notation for the Process blueprint

进程视图…

通信系统ACS的局部进程视图



进程视图

- （2）进程视图可以描述成多层抽象，每个级别分别关注不同的方面

- （3）多种风格适用于进程视图，如：Pipes&Filters, Client/Server (Multiple Client/Single Server, Single Client/Multiple Server)。

物理视图

- 物理视图（physical view）主要考虑如何把软件映射到硬件上;
- 它通常要考虑到系统性能、规模、可靠性等;解决系统拓扑结构、系统安装、通信等问题。

物理视图

- 软件在计算机网络或处理节点上运行，被识别的各种元素（网络、过程、任务和对象），需要被映射至不同的节点；
- 我们希望使用不同的物理配置：
  - 一些用于开发和测试，另外一些则用于不同地点和不同客户的部署
- 因此软件至节点的映射需要高度的灵活性及对源代码产生最小的影响。

物理视图

- 物理视图的标记法

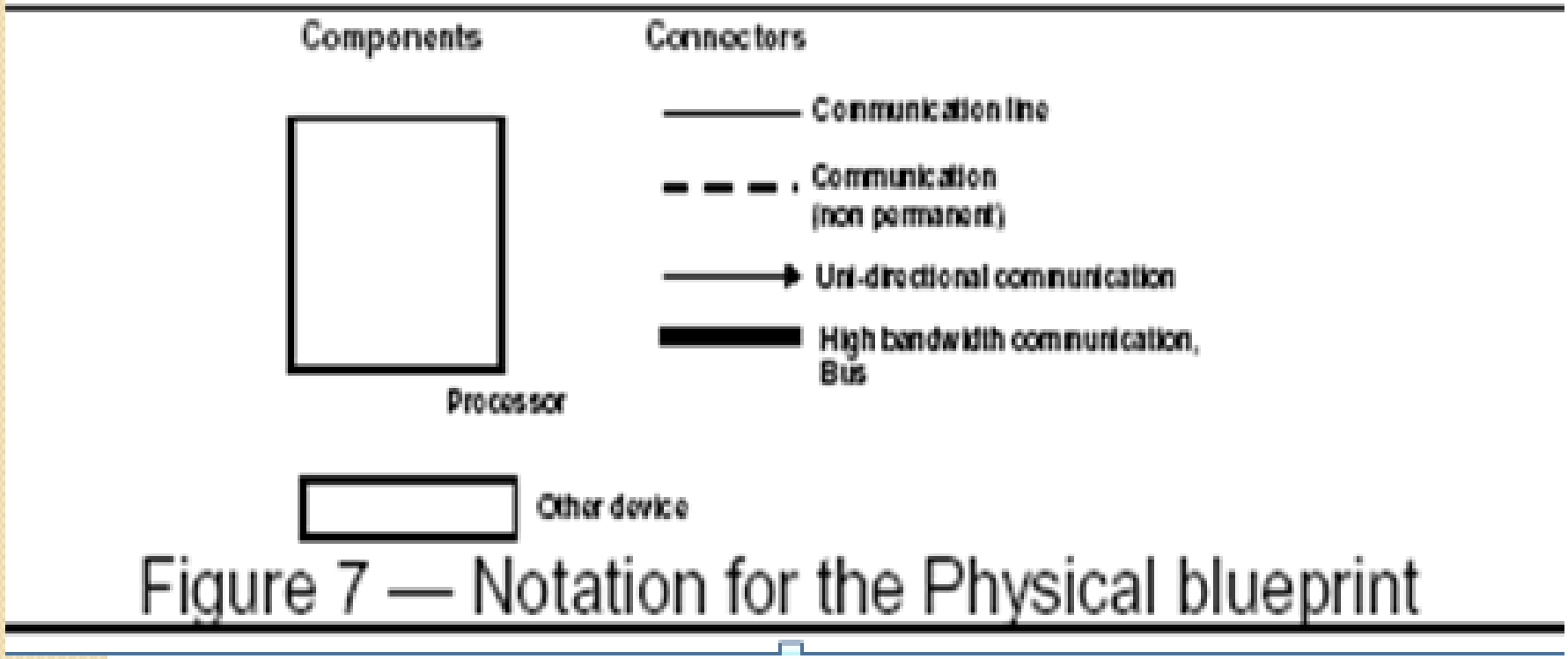


Figure 7 — Notation for the Physical blueprint

物理视图

C,F,K是三个不同容量的计算机类型，支持三个不同的可执行文件

- 物理视图示例

图－8显示了大型ACS的可能硬件配置

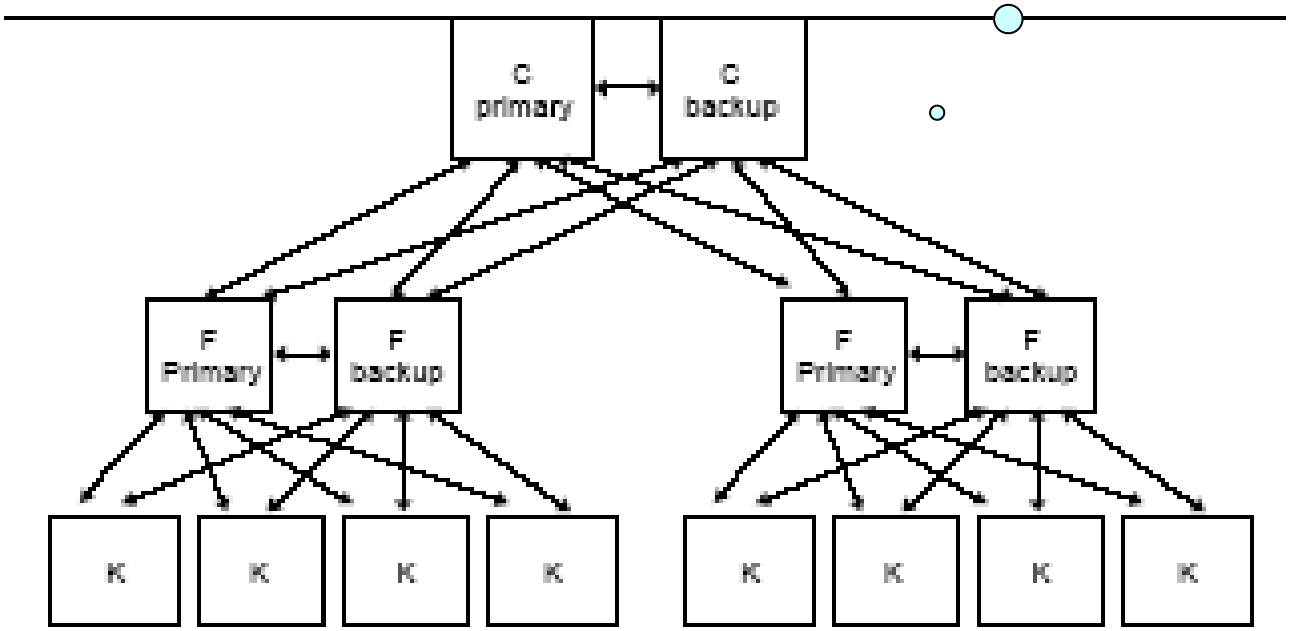


Figure 8 — Physical blueprint for the PABX

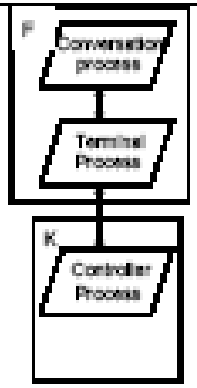


Figure 9 — A small PABX physical architecture with process allocation

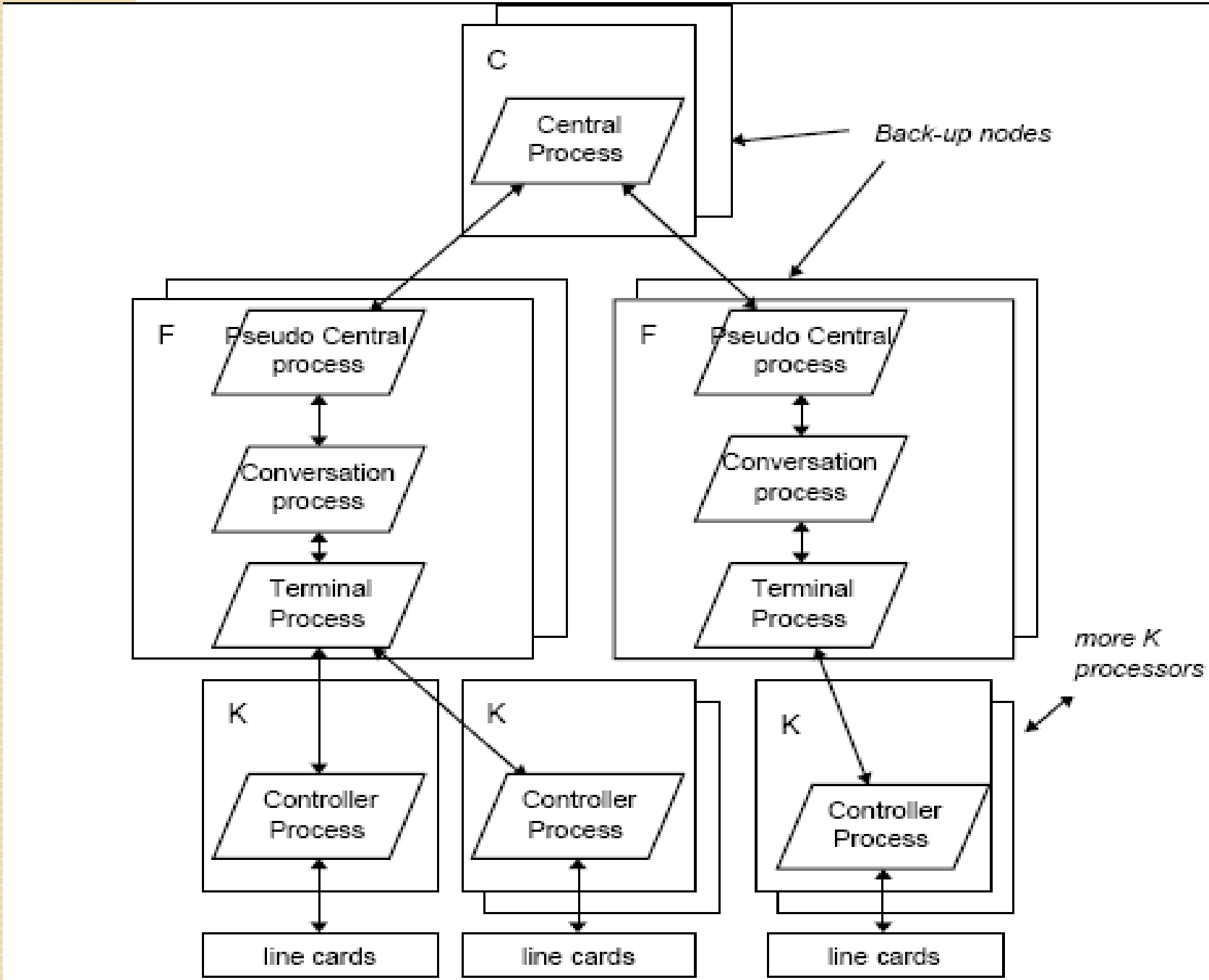


Figure 10 — Physical blueprint for a larger PABX showing process allocation

场景

- 场景（scenarios）可以看做那些重要系统活动的抽象，它使4个视图有机联系起来。
- 四种视图的元素通过少量的重要场景（如更具普遍性的use case）来无缝地协同工作。
- 它可以帮助架构师找到体系结构的构件和它们之间的作用关系。
- 从某种意义上讲，场景是最重要的需求抽象。

场景

- 场景是其它视图的冗余（因而“+1”），但它有两点作用：
  - （1）在SA设计过程中，作为一个驱动来发现SA元素；
  - （2）在SA设计结束时，从书面上作为对SA原型测试的起点。

场景

- 场景的设计用**对象场景图**和**对象交互图**来表示。

- **场景蓝图的标记法**

  对组件而言，标记法同逻辑视图类似，但使用进程视图的连接符号来表达对象之间的交互。

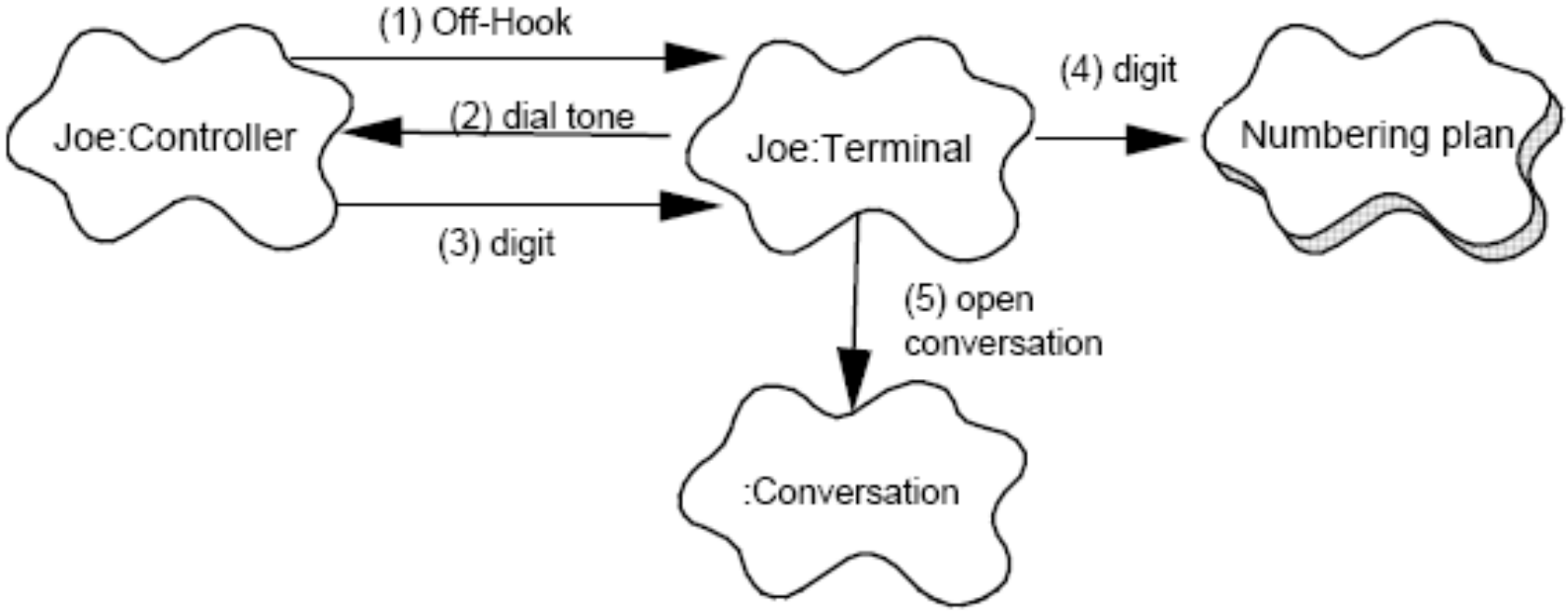


Figure 11 — Embryo of a scenario for a local call—selection phase

1. Joe的电话控制器检测和校验摘机状态的变换,并发送消息唤醒相应的终端对象。
2.终端分配一些资源，并要求控制器发出拨号音。
3. 控制器接受拨号并传递给终端。
4. 终端使用拨号方案来分析数字流。
5. 有效的数字序列被键入，终端开始会话。

二、“4＋1”视图模型

- 小结
  - （1）逻辑视图和开发视图描述系统的静态结构，而进程视图和物理视图描述系统的动态结构。
  - （2）对于不同的软件系统来说，侧重的角度也有所不同。例如，对于管理信息系统，比较侧重于从逻辑视图和开发视图来描述系统，而对于实时控制系统，则比较注重于从进程视图和物理视图来描述系统。
  - （3）逻辑视图和开发视图非常接近，但具有不同的关注点。我们发现项目规模越大，视图间的差距也越大。

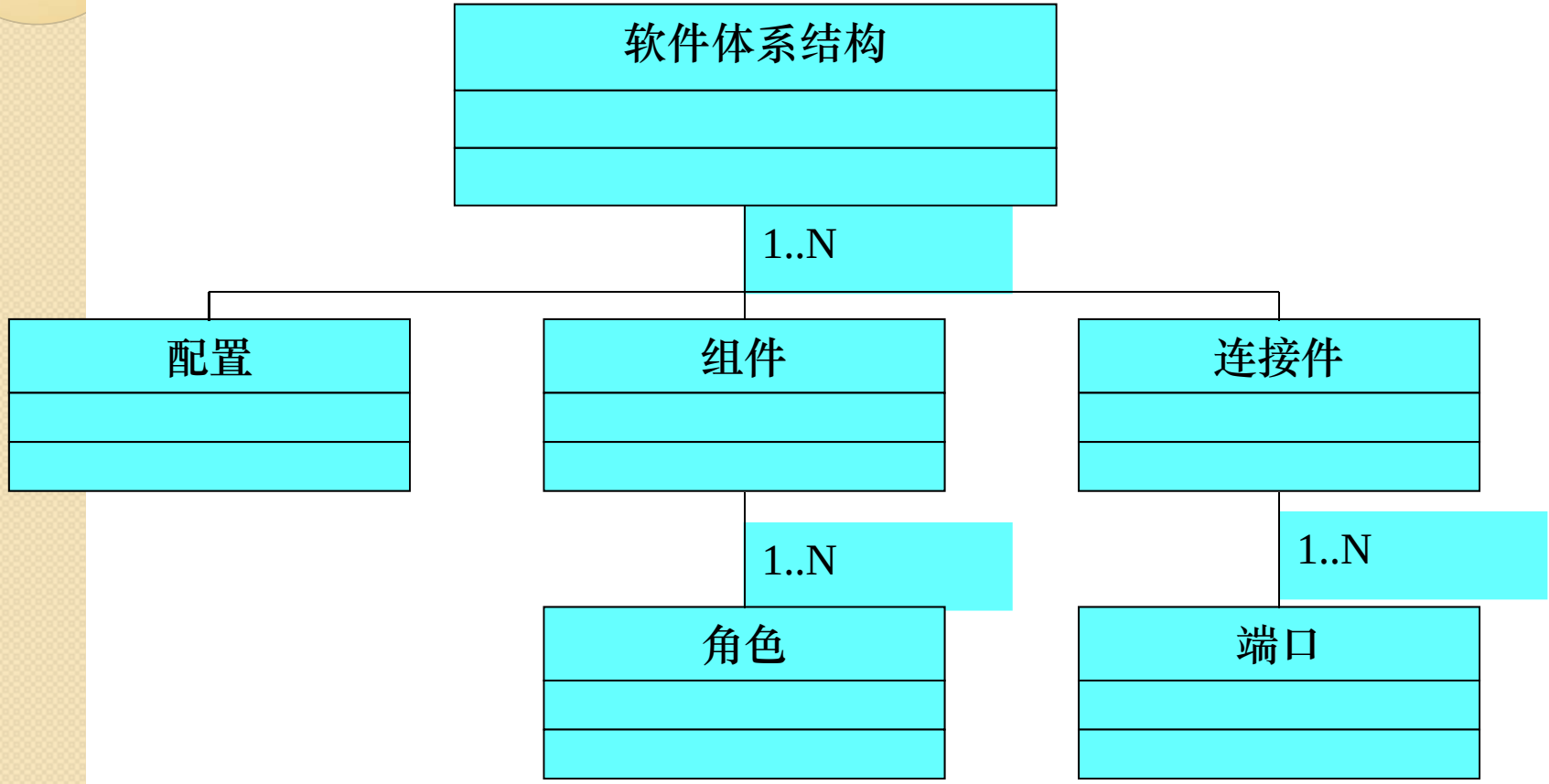
二、“4＋1”视图模型

- （4）并不是所有的软件架构都需要“4＋1”视图。无用的视图可以从架构描述中省略。
  - 比如：只有一个处理器，则可以省略物理视图；而如果仅有一个进程或程序，则可以省略过程视图。对于非常小型的系统，甚至可能逻辑视图与开发视图非常相似，而不需要分开的描述。
  - 但一般情况下，场景对于所有的情况均适用。

三、软件体系结构的核心模型

- SA核心模型由五种元素构成
  - 组件（component），
  - 连接件（connector），
  - 配置（configuration），
  - 端口（port）
  - 角色（role）。

三、软件体系结构核心模型…



三、软件体系结构的核心模型

- **组件**：具有某种功能的可重用的软件模块单元，表示了系统中主要的计算单元和数据存储。组件有两种：复合组件和原子组件。复合组件由其它复合组件和原子组件通过连接而成。
- **连接件**：表示了组件之间的交互，简单的连接件有：管道（pipe）、过程调用（procedure-call）、事件广播（event broadcast）等。复杂的连接件有：客户－服务器（client-server）通信协议，数据库和应用之间SQL连接等。
- **配置**：表示了组件和连接件的拓扑逻辑和约束。

三、软件体系结构核心模型...

- **端口**：组件作为一个封装的实体，只能通过其接口与外部交互，组件的接口由一组端口组成，每个端口表示了组件和外部环境的交汇点。通过不同的端口类型，一个组件可以提供多重接口。端口可以很简单，如过程调用。
- **角色**：连接件作为建模软件体系结构的主要实体，同样也有接口，连接件的接口由一组角色组成，连接件的每个角色定义了该连接件表示的交互的参与者。二元连接件有两个角色，如RPC的角色为caller 和callee，pipe 的角色是reading 和writing，消息传递的角色是sender 和receiver等。有的连接件有多于两个的角色，如事件广播有一个事件发布者角色和任意多个事件接收者角色。

```
Configuration GasStation
    Component Customer
        Port Pay = pay!x → Pay
        Port Gas = take → pump?x → Gas
        Computation = Pay.pay!x → Gas.take → Gas.pump?x → Computation
    Component Cashier
        Port Customer1 = pay?x → Customer1
        Port Customer2 = pay?x → Customer2
        Port Topump = pump!x → Topump
        Computation = Customer1.pay?x → Topump.pump!x → Computation
            [] Customer2.pay?x → Topump.pump!x → Computation
    Component Pump
        Port Oil1 = take → pump!x → Oil1
        Port Oil2 = take → pump!x → Oil2
        Port Fromcashier = pump?x → Fromcashier
        Computation = Fromcashier.pump?x →
            (Oil1.take → Oil1.pump!x → Computation)
            [] (Oil2.take → Oil2.pump!x → Computation)
    Connector Customer_Cashier
        Role Givemoney = pay!x → Givemoney
        Role Getmoney = pay?x → Getmoney
        Glue = Givemoney.pay?x → Getmoney.pay!x → Glue
    Connector Customer_Pump
        Role Getoil = take → pump?x → Getoil
        Role Giveoil = take → pump!x → Giveoil
        Glue = Getoil.take → Giveoil.take → Giveoil.pump?x → Getoil.pump!x → Glue
    Connector Cashier_Pump
        Role Tell = pump!x → Tell
        Role Know = pump?x → Know
        Glue = Tell.pump?x → Know.pump!x → Glue
    Instances
        Customer1: Customer
        Customer2: Customer
        cashier: Cashier
        pump: Pump
        Customer1_cashier: Customer_Cashier
        Customer2_cashier: Customer_Cashier
        Customer1_pump: Customer_Pump
        Customer2_pump: Customer_Pump
        cashier_pump: Cashier_Pump
    Attachments
        Customer1.Pay as Customer1_cashier.Givemoney
        Customer1.Gas as Customer1_pump.Getoil
        Customer2.Pay as Customer2_cashier.Givemoney
        Customer2.Gas as Customer2_pump.Getoil
        casier.Customer1 as Customer1_cashier.Getmoney
        casier.Customer2 as Customer2_cashier.Getmoney
        cashier.Topump as cashier_pump.Tell
        pump.Fromcashier as cashier_pump.Know
        pump.Oil1 as Customer1_pump.Giveoil
        pump.Oil2 as Customer2_pump.Giveoil
End GasStation.
```

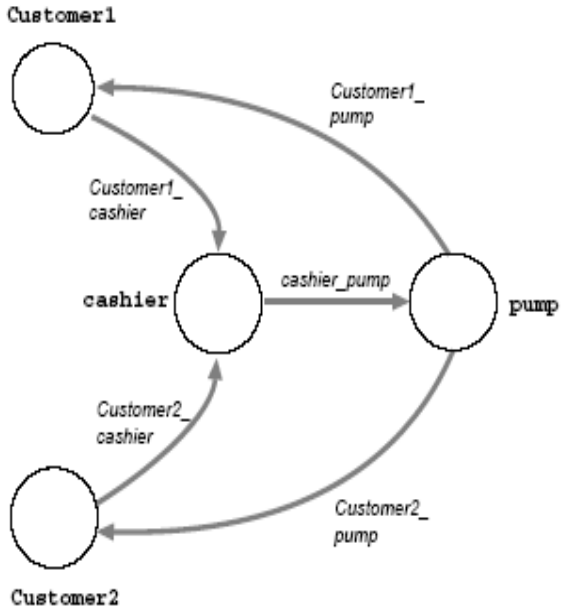


Figure 3: The architecture of the Gas Station system.

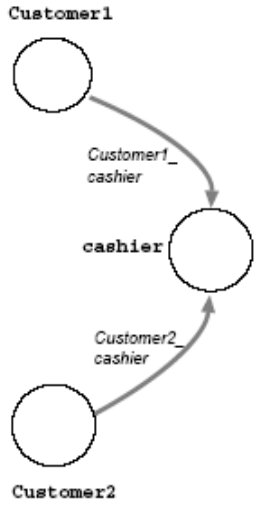


Figure 7: The architectural representation of the slice in Figure 6.

四、软件体系结构的生命周期

- 基于SA的软件开发分为如下几个阶段：
  - 需求分析阶段
    - 体系结构需求包括需求获取(功能需求和质量需求)、生成类图(Rational Rose)、对类分组、把类打包成组件和需求评审等.
    - 需求评审:组织一个有不同代表，如分析人员、客户、设计人员、测试人员，组成的小组，对体系结构需求及相关组件进行仔细的审查。审查内容：获取的需求是否真实反映了用户的要求，类的分组是否合理，组件合并是否合理等过程。

四、软件体系结构的生命周期

- 基于SA的软件开发分为如下几个阶段：
  - 建立软件体系结构阶段
    - 在这个阶段，体系结构设计师（architect）主要从结构的角度对整个系统进行分析，选择恰当的组件、组件间的相互作用关系以及对它们的约束，最后形成一个系统框架以满足用户需求，为设计奠定基础。
    - 首先，选择合适的体系结构风格。其次，把需求阶段已确认的组件映射到体系结构中，产生中间结构。最后，分析组件的相互作用和关系。一旦决定了关键组件之间的关系和相互作用，就可以在前面得到的中间结构的基础上进行细化。

四、软件体系结构的生命周期

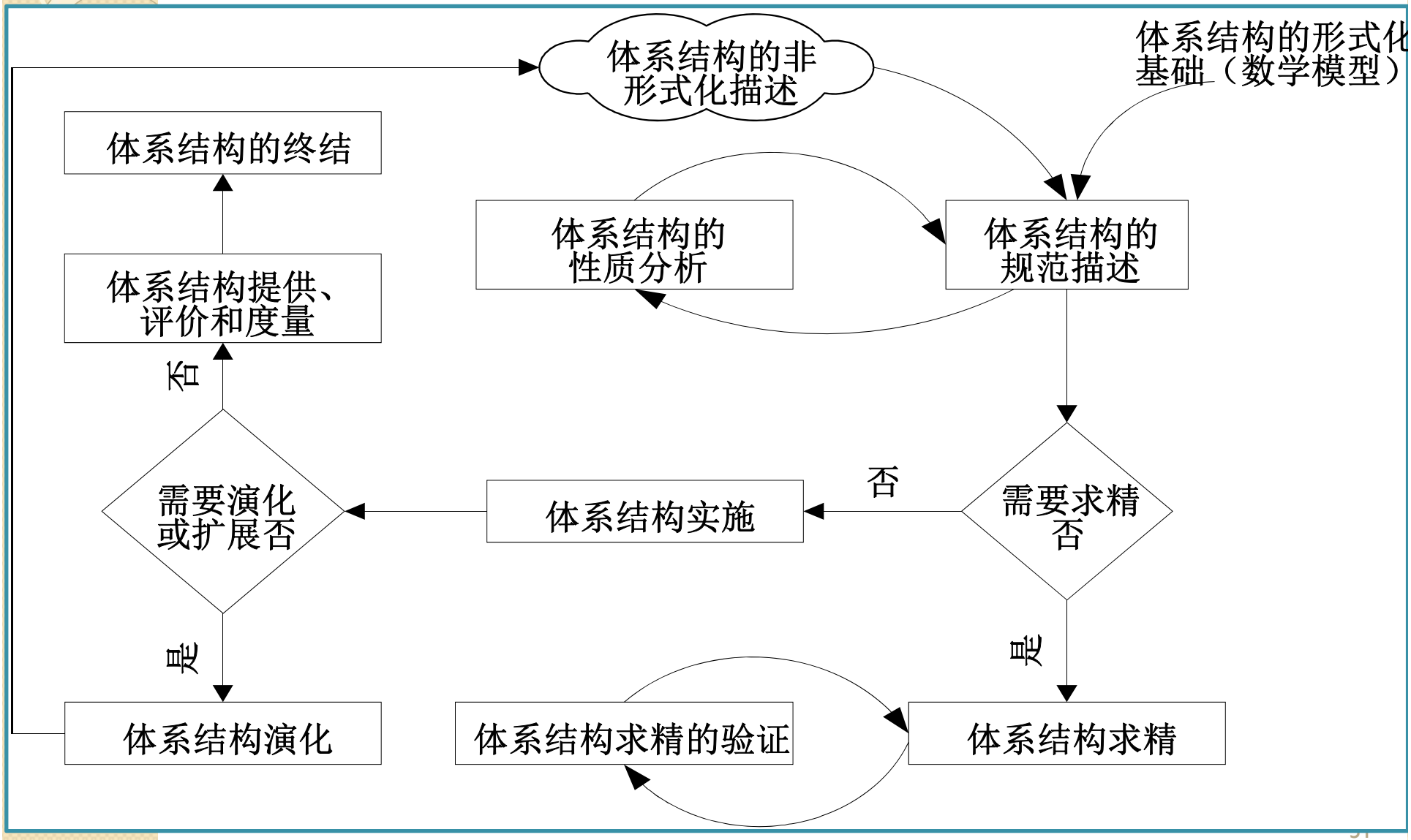
- 基于SA的软件开发分为如下几个阶段：
  - **设计阶段**—对系统进行模块化并决定各个组件间的详细接口、算法和数据类型的选定，对上支持建立体系结构阶段形成的框架，对下提供实现基础。
  - **实现阶段**—将设计阶段设计的算法及数据类型进行程序语言表示，满足设计体系结构和需求分析的要求，从而得到满足设计要求的目标系统。
  - **测试**—包括单个组件的功能测试和被组装应用的整体功能和性能测试。

四、软件体系结构的生命周期

- SA在系统开发的全过程中起着基础的作用，是设计的起点和依据，同时也是装配和维护的指南。和软件本身一样，SA也有其生命周期。

四、软件体系结构的生命周期

- 软件体系结构非形式化描述
- 软件体系结构的规范化描述和分析
- 软件体系结构的求精及其验证
- 软件体系结构的实施
- 软件体系结构的演化和扩展
- 软件体系结构的提供、评价和度量
- 软件体系结构的终结


体系结构的非
形式化描述
体系结构的形式化
基础（数学模型）
体系结构的终结
体系结构的
性质分析
体系结构的
规范描述
体系结构提供、
评价和度量
否
需要演化
或扩展否
体系结构实施
否
需要求精
否
是
是
体系结构演化
体系结构求精的验证
体系结构求精

四、软件体系结构的生命周期…

- 软件体系结构非形式化描述

  —自然语言描述；创造性和开拓性。

- 软件体系结构的规范化描述和分析

  —形式化数学理论模型描述，分析SA性质（deadlock free, safety,liveness etc）。

- 软件体系结构的求精及其验证

  —抽象到具体，逐步求精；验证判断具体SA与抽象SA语义一致性，并实现抽象SA。

- 软件体系结构的实施

  —将精化的SA实施于系统的设计中，并将SA的组件和连接件等有机地组织在一起，形成系统设计的框架，以便据此实施于软件设计和构造中。

四、软件体系结构的生命周期...

- **软件体系结构的演化和扩展**

  —在实施SA时，根据系统的需求(常常是非功能的需求，如性能、容错、安全性、互操作性、自适应性等非功能性质) 来扩展和改动SA，这称为SA的演化。

- **软件体系结构的提供、评价和度量**

  —定性评价和定量度量，以利于对SA的重用。

- **软件体系结构的终结**

  —SA进行多次演化、修改变得难以理解，不能适应系统的发展。

五、软件体系结构的抽象模型

- **构件 定义1**：构件是一个数据单元或一个计算单元，它由构件的对象的集合、属性的集合、动作的集合和端口的集合组成。
  - 构件可以抽象为：*C=(O,A,X,P)*
  - 其中，O为组成构件的对象的集合
  - A为构件属性的集合
  - X为构件动作的集合
  - P为构件端口的集合
- 构件之间的主要关系：顺序，选择，循环

五、软件体系结构的抽象模型

- 连接件

  通过对组件间交互规则的建模来实现组件间的连接。与组件不同，连接件不需编译。

- 定义**6**：连接件是组件运算的实现，它是六元组<ID, Role, Beha, Msgs, Cons, Non-Func$_i$> 其中：

  （1）ID是连接件的标识。

  （2） Role为连接件和组件交互点的集合。其中Role=<Id,Action, Event, LConstraints>. Id是Role的标识； Action是Role活动的集合，每个活动由事件的连接（谓词）组成； Event是Role产生的事件集合； LConstraints是Role约束集合。

  （3）Beha是连接件的行为集合。

  （4） Msge是连接件中各Role中事件产生的消息的集合。

  （5） Cons是连接件约束的集合，它包括连接件的初始条件、前置条件和后置条件，有时为了明确表示这三个条件，可把它写成Cons(init,pre-cond,post-cond)。

  （6） Non-Func$_i$是连接件的非功能说明，包括连接件的安全性、可靠性说明等。

五、软件体系结构的抽象模型

- 软件体系结构
- 定义7：设论域为U，（1）组件是一个软件体系结构；（2）连接件是一个软件体系结构；（3）组件经有限次连接（运算）后是软件体系结构。SA记为A＝<C,O>其中C表示组成体系结构的组件集合，O表示组件运算的集合。
- SA的性质

  （1）封闭性（envelopment）：即组件与组件、组件与体系结构、体系结构与体系结构连接后仍是一个体系结构。

  （2）层次性（hierarchy）：即体系结构可由组件连接而成，而体系结构又可以再经过连接组成新的更大的体系结构。

  （3）可扩充性（expansibility）：即一个满足条件的新组件可以通过连接加入到结构中。