

软件体系结构



第四章 软件体系结构描述

4.1 软件体系结构描述方法

4.2 软件体系结构描述框架标准

4.3 体系结构描述语言

4.4 典型的软件体系结构描述语言

4.5 软件体系结构与UML

4.6 可扩展标记语言

4.7 基于XML的软件体系结构描述语言

4.3 体系结构描述语言

1、ADL基本概念

目前，已定义的ADL超过20种。在国外，具代表性的ADL包括Aesop，C2，Darwin，MetaH，Rapide，Unicon，Wright，ACME等；国内包括XYZ/ADL，ABC / ADL，FRADL，A-ADL等。

关于ADL目前没有明确的定义，通常人们将“用来表示和分析软件体系结构设计的形式化符号称为ADL”。

4.3 体系结构描述语言

2、ADL构成要素

Tracz定义一个ADL包含4“C”:

- 1) 组件(components)
- 2) 连接子(connectors)
- 3) 配置(configurations)
- 4) 约束(constraints)。

4.3 体系结构描述语言

2、ADL构成要素

Booch认为软件体系结构描述应该着重识别、选择与确认那些对体系结构有重要影响的元素：

- 1) 主要的业务类；
- 2) 重要的机制；
- 3) 处理与过程；
- 4) 分层与子系统；
- 5) 接口。

4.3 体系结构描述语言

2、ADL构成要素

根据其UniCon的经历，Shaw与同事列出了ADL应该展示的属性如下：

- 1) 组件建模的能力，带属性断言、接口、实现；
- 2) 连接器建模的能力，带协议、属性断言与实现；
- 3) 抽象与封装；
- 4) 类型与类型检查；
- 5) 适应分析工具的能力。

4.3 体系结构描述语言

2、ADL构成要素

Luckham与Vera在研究Rapide的基础上提出，ADL应该满足以下要求：1) 组件抽象；2) 通信抽象；3) 通信完整性，要求在结果实现中，只有在一个体系结构中相联结的组件可以通信；4) 动态体系结构建模的能力；5) 分层复合；6) 相关性，或者是行为与体系结构间关联(映射)的能力；7) 组件建模的能力，带属性断言、接口、实现；8) 连接器建模的能力，带协议、属性断言与实现；9) 抽象与封装；10) 类型与类型检查；11) 适应分析工具的能力。

4.3 体系结构描述语言

一般认为，ADL的核心设计元素主要包括：构件(Component)、连接件(Connector)和体系结构配置(Architecture Configuration)。

其中，构件表示系统中主要的计算元素和数据存储，如客户端、服务器、数据库等；连接件定义了构件之间的交互关系，如过程调用、消息传递、事件广播等；体系结构配置描述了构件、连接件之间的拓扑关系。

4.3 体系结构描述语言

构件、连接件定义中的一个重要方面是对其外部特性的描述，即接口的定义。

通常，ADL还采用一种形式化技术作为语义信息描述的理论基础，如 π -calculus、偏序事件集理论、CSP、XYZ/E等。ADL对设计元素的定义方式以及采用的形式化理论，很大程度上决定了ADL所具备的描述能力和适用范围。

4.3 体系结构描述语言

1) 组件

组件是一个计算单元或数据存储。也就是说，组件是计算与状态存在的场所。在体系结构中，一个组件可能小到只有一个过程（如MetaH 过程）或大到整个应用程序（如C2、Rapid 中的分层组件，或MetaH中的宏），它可以要求自己的数据与/或执行空间，也可以与其它组件共享这些空间。

作为软件体系结构构造块的组件，其自身也包含了多种属性，如接口、类型、语义、约束、进化、非功能属性等。现有的ADL对这些属性的侧重有所不同。

4.3 体系结构描述语言

1) 组件

接口是组件与外部世界的一组交互点。与OO类或Ada包说明相似，ADL中的组件接口说明了组件提供的那些服务（消息、操作、变量）。为了能够充分地推断组件及包含它的体系结构，ADL提供了能够说明组件需要（即要求体系结构中其它组件为之提供的服务）的工具。这样，接口就定义了组件能够提出的计算委托及其用途上的约束。

4.3 体系结构描述语言

1) 组件

组件类型是实现组件重用的手段。组件类型保证了组件能够在体系结构描述中多次实例化，并且每个实例可以对应于组件的不同实现。抽象组件类型也可以参数化，进一步促进重用。现有的ADL都将组件类型与实例区分开来。

4.3 体系结构描述语言

1) 组件

由于基于体系结构开发的系统大都是大型、长时间运行的系统，因而系统的进化能力显得格外重要。组件的进化能力是系统进化的基础。ADL是通过组件的子类型及其特性的细化来支持进化过程的。目前，只有少数几种ADL部分地支持进化，对进化的支持程度通常依赖于所选择的实现（编程）语言。其它ADL将组件模型看作是静态的。ADL语言大多是利用语言的子类型来实现对进化支持的：利用OO方法，从其它类型派生出它的接口类型，形成结构子类型（Rapde）。

4.3 体系结构描述语言

2) 连接子

连接子是用来建立组件间的交互以及支配这些交互规则的体系结构构造模块。与组件不同，连接子可以不与实现系统中的编译单元对应。它们可能以兼容消息路由设备实现（如C2），也可以以共享变量、表入口、缓冲区、对连接器的指令、动态数据结构、内嵌在代码中的过程调用序列、初始化参数、客户服务协议、管道、数据库、应用程序之间的SQL语句等等形式出现。大多效ADL将连接子作为第一类实体，也有的ADL（Rapide与MetaH）则不将连接子作为第一类实体。

4.3 体系结构描述语言

2) 连接子

连接子作为建模软件体系结构的主要实体，同样也有接口。连接子的接口由一组角色组成，连接子的每一个角色定义了该连接子表示的交互参与者。

连接子的接口是一组它与所连接组件之间的交互点。为了保证体系结构中的组件连接以及它们之间的通信正确，连接子应该导出所期待的服务作为它的接口。它能够推导出体系结构配置的形成情况。体系结构配置中要求组件端口与连接子角色的显式连接。

4.3 体系结构描述语言

2) 连接子

体系结构级的通信需要用复杂协议来表达。为了抽象这些协议并使之能够重用，ADL应该将连接子构造为类型。构造连接子类型可以将作为用通信协议定义的类型系统化，并独立于实现，或者作为内嵌的、基于它们的实现机制的枚举类型。

4.3 体系结构描述语言

2) 连接子

为完成对组件接口的有用分析、保证跨体系结构抽象层的细化一致性、强制互联与通信约束等，体系结构描述提供了连接子协议以及变换语法。

为了确保执行计划的交互协议，建立起内部连接子依赖关系，强制用途边界，就必须说明连接子约束。ADL可以通过强制风格不变性来实现约束（Aesop、C2、SADL），或通过接受属性限制给定角色中的服务（Unicon），而在Wright中，则通过为每个角色说明接口协议来实现约束。

4.3 体系结构描述语言

3) 配置

体系结构配置或拓扑是描述体系结构的组件与连接子的连接图。体系结构配置提供信息来确定组件是否正确连接、接口是否匹配、连接子构成的通信是否正确，并说明实现要求行为的组合语义。

4.3 体系结构描述语言

3) 配置

体系结构适合于描述大的、生存期长的系统。利用配置来支持系统的变化，使不同技术人员都能理解并熟悉系统。为帮助在一个较高的抽象层上理解系统（或系统族），就需要对软件体系结构进行说明。

为了使开发者与其有关人员之间的交流容易些，ADL必须以简单的、可理解的语法来配置结构化（拓扑的）信息。理想的情况是从配置说明中澄清系统结构，即不需研究组件与连接子，就能使构建系统的各种参与者理解系统。体系结构配置说明除文本形式外，有些ADL还提供了图形说明形式，文本描述与图形描述可以互换。多视图、多场景的体系结构说明方法在最新的研究中得到了明显的加强。

4.3 体系结构描述语言

3) 配置

为了在不同细节层次上描述软件系统，ADL将整个体系结构作为另一个较大系统的单个组件。也就是说，体系结构具有复合或等级复合的特性。另一方面，体系结构配置支持采用异构组件与连接子。这是因为软件体系结构的目的是促进大规模系统的开发，即倾向于使用已有的组件与不同粒度的连接子。这些组件与连接子的设计者、形式化模型、开发者、编程语言、操作系统、通信协议可能都不相同。另外一个事实是，大型的、长期运行的系统是在不断增长的。因而，ADL必须支持可能增长的系统的说明与开发。大多数ADL提供了复合特性，所以，任意尺度的配置都可以相对简洁地在足够的抽象高度表示出来。

4.3 体系结构描述语言

3、ADL与其它语言的区别

按照Mary Shaw和David Garlan的观点，典型的ADL在充分继承和吸收传统程序设计语言的精确性和严格性特点的同时，还应具有以下能力和特性：

- 构造能力
- 抽象能力
- 重用能力
- 组合能力
- 异构能力
- 分析和推理能力

4.3 体系结构描述语言

3、ADL与其它语言的区别

根据上述特点，可以将下面这样的语言排除在ADL之外：

- 高层设计符号语言
- 模块接口语言MIL
- 编程语言
- 面向对象(OO)的建模符号
- 形式化说明语言。

4.3 体系结构描述语言

3、ADL与其它语言的区别

ADL关注概念体系结构并显式地将连接子作为第一类实体，从而将模块接口语言(MIL)、编程语言、OO建模符号等排除在ADL之外。MIL描述的是使用已实现系统中模块间的关系并只支持连接的一种类型。编程语言描述系统的实现，其体系结构隐含在子程序定义与调用中。

4.3 体系结构描述语言

3、ADL与其它语言的区别

ADL通常要包含一形式化语义理论，用以刻画体系结构的底层框架，它影响了ADL对某些专门领域系统的建模适应性。

ADL与需求语言的区别在于后者描述的是问题空间，而前者则扎根于解空间中。

ADL与建模语言的区别在于后者对整体行为的关注要大于对部分的关注，而ADL集中在组件的表示上。

4.4 典型的软件体系结构描述语言

1、UniCon

Unicon是Shaw等开发的体系结构描述语言，是一种实现约束语言，它支持异构型组件和连接器的描述。

Unicon侧重于描述系统的结构，它提供了一种将组件和连接器以“即插即用”的构建风格组合成系统的方式，Unicon提供了一组预定义的组件和连接器类型，体系结构设计人员可以从中选择合适的组件或连接器。

4.4 典型的软件体系结构描述语言

1、UniCon

Unicon的两种基本元素是组件和连接器。

组件决定计算能力，组件通过接口向外部导出它实现的计算能力，组件用接口定义一组扮演者(Player)，组件可以通过扮演者与外界交互。

在Unicon中，通过定义类型、特性列表(attribute-value pairs)和用于和连接器相连的交互点(Players)来描述构件。

4.4 典型的软件体系结构描述语言

1、UniCon

连接器负责组件之间的交互，它描述若干通信协议以及进行交互时需要的附件机制，连接器定义了角色(Role)，角色是用来指明连接器进行的交互活动。

连接器也是通过类型、特性列表和交互点来描述，连接器的交互点称为roles。

Unicon支持单独存在的、对称和非对称的连接件，用来规定构件之间的交互机制。

角色、扮演者具有类型和属性，用来表明构件希望参与交互的性质和相关的交互细节。

4.4 典型的软件体系结构描述语言

1、UniCon

系统配置的时候，构件的扮演者与连接件的角色相互关联。运用Unicon提供的Connection Expert和配置工具能够构造运行的软件系统。

4.4 典型的软件体系结构描述语言

1、UniCon

不足之处:

- 1) 仅仅支持固定的交互类型和连接件，新型交互类型必须通过编程提供新的Connection Expert才能实现；
- 2) 没有提供描述一类系统的机制和方法，提供的分析方法依赖于与连接件密切相关的分析工具并且同实现相关，因此，Unicon在体系结构风格规约方面存在不足；
- 3) Unicon规约的组件和连接件同实现相关，体系结构模型与实现相互对应，是实现规约语言，与模块互连规约并无本质区别。

4.4 典型的软件体系结构描述语言

2、Wright

Wright是Robert Allen等开发的体系结构描述语言。它的形式语义基础是通信顺序进程CSP(Communication Sequence process)。Wright具有和Unicon类似的体系结构规约框架，即从构件、连接件和配置三个方面描述体系结构。但与Uniocn不同的是，Wright不是面向实现，而是在更高抽象层次进行体系结构的形式规约。

4.4 典型的软件体系结构描述语言

2、Wright

Wright运用CSP并对其进行一定扩充，描述构件端口的交互行为和它的计算行为，描述连接件的交互协议和角色规约。因此，Wright提供显式、独立的连接件规约和支持任意复杂连接件的规约。

构件通过其接口点(ports)和行为(computation)来定义，表明了ports之间是如何通过构件的行为而具有相关性的。

连接器通过协议(protocol)来定义，而协议刻画了与连接器相连的构件的行为。对连接器接口点(roles)的描述表明了对参与交互的构件的“期望”以及实际的交互进行过程。

4.4 典型的软件体系结构描述语言

2、Wright

Wright定义连接件和构件的实例，在相应的端口和角色之间建立连接(Attachment)，从而得到系统配置，并能够推导得出描述该系统配置行为的CSP进程。一旦构件和连接器的实例被声明，系统组合便可以通过构件的port和连接器的role之间的连接来完成。

wright同时通过绑定机制(Binding)支持系统的层次构造。

4.4 典型的软件体系结构描述语言

2、Wright

Wright不仅能提供单个系统体系结构的分析和规约，而且支持体系结构风格的分析和规约。它通过组件和连接件类型来定义体系结构风格术语，用组件和连接件的相关约束机制来表示体系结构风格的属性。

4.4 典型的软件体系结构描述语言

2、Wright

Wright的另一个突出特点是提供了大量的体系结构静态分析机制。借助体系结构的CSP形式规约，能够进行端口、角色连接的兼容性检测、端口与构件计算行为的一致性检测、角色与粘接(Glue)规约一致性的检测、系统死锁分析等，从而在体系结构开发阶段，能够及时发现其中的错误，提升体系结构设计的质量，降低系统开发的风险。

4.4 典型的软件体系结构描述语言

2、Wright

不足之处:

- 1) 因为CSP固有的静态特性，Wright在描述动态体系结构方面存在不足;
- 2) Wright未能考虑体系结构的精化和实现，而体系结构的精化和实现都是体系结构中非常重要的研究课题。

4.4 典型的软件体系结构描述语言

3、C2

C2是由Richard N. Taylor等开发的体系结构描述语言。C2基于消息，适合描述图形用户界面软件的体系结构。C2的构件模型分为上下两端，分别是消息的发送和接受端口。它的连接机制是消息总线，用于对消息进行过滤和转发。

4.4 典型的软件体系结构描述语言

3、C2

C2构件包含有4个内部部分：内部对象(internal object)、包装器(wrapper)、对话(dialog)和域转换器(domain translator)。内部对象存贮构件状态并实现构件所提供的操作。内部对象上的包装器监控所有的操作请求，并通过底端接口发送通知。对话负责把接收到的外部消息映射成内部对象上的操作。域转换器是可选的，它可以修改一些消息使其能被其他构件理解，这样，一个构件就能在特定的体系结构中适用。

4.4 典型的软件体系结构描述语言

3、C2

构件负责维护状态，进行操作，通过两个接口(顶端接口和底端接口)和其他构件交换消息。每一个接口包括一个可以发送的消息集合和一个可以接收的消息集合。构件之间的消息或者是请求构件执行一个操作，或者是一个通知(有关某个构件已经进行了一个操作或已经改变了状态)。

4.4 典型的软件体系结构描述语言

3、C2

构件之间不能发送消息，必须通过连接件。构件的每一个接口最多只能连接到一个连接件上。一个连接件可以连接多个构件和连接件。请求消息只能在体系结构中自下而上地发送，而通知消息只能自上而下地发送。而且，构件之间的通信只能通过消息传递来实现，不允许使用共享内存方式通信。

4.4 典型的软件体系结构描述语言

3、C2

体系结构的3个组成要素：构件、连接件和体系结构配置，都有一个演化过程。C2对于构件演化的支持是通过类的子类型实现的。

4.4 典型的软件体系结构描述语言

3、C2

不足之处：

C2所提供的体系结构规约的语法标记符号没有相关语义的约束，所以它不支持体系结构风格的规约和分析，只能应用于特定领域的研究工作。

4.4 典型的软件体系结构描述语言

4、Rapide

Rapide是Luchham等开发的体系结构描述语言，是一种可执行的ADL，它通过定义并模拟基于事件的行为对分布式并发系统进行建模。Rapide基于偏序事件集对构件的计算行为和交互行为进行建模。

Rapide与Darwin和Unicon不同，它独立于实现，运用通信事件序列描述构件的行为。

Rapide的通信事件分为两类：外部动作(Extern Action)和公共动作(Public Action)。Rapide通过观察外部动作，并在它们和公共动作之间建立关联，从而描述构件和系统的计算行为。

4.4 典型的软件体系结构描述语言

4、Rapide

Rapide由五种子语言构成：

- 类型(Types)语言—定义接口类型和函数类型，支持通过继承已有的接口来构造新的接口类型。
- 模式(Pattern)语言—定义具有因果、独立、时序等关系的事件所构成的事件模式。
- 可执行(Executable)语言—包含描述构件行为的控制结构。
- 体系结构(Architecture)语言—通过定义同步和通信连接来描述构件之间的事件流。
- 约束(Constraint)语言—定义构件行为和体系结构所满足的形式化约束，其中约束为需要的或禁止的偏序集模式。

4.4 典型的软件体系结构描述语言

4、Rapide

Rapide通过连接(Connection)机制建立构件之间的交互。连接机制在构件接口的外部动作和公共动作之间建立关联,规定构件间事件发生的因果关系。作为连接的结果,当连接模式左边的事件启动时,它右边的事件随之发生。

4.4 典型的软件体系结构描述语言

4、Rapide

不足之处:

- 1) 类似于Drawin的绑定，Rapide只能描述构件之间非对称的交互关系；
- 2) Rapide的连接机制嵌入配置定义当中，两者不能分离。因此，Rapide不允许单独对连接件进行描述和分析；
- 3) 并且没有提供相关机制，把多个连接机制捆绑成为一个整体，构成复杂的交互模式，因此，Rapide描述构件交互模式的能力存在不足；
- 4) Rapide通过体系结构仿真执行一致性检测和相关分析。进行体系结构仿真时，生成系列偏序事件集，检测它们是否与接口、构件行为和连接机制的规约兼容。而仿真本质上仅仅意味着体系结构的测试，而不是严密的分析。

4.4 典型的软件体系结构描述语言

5、SADL

SADL语言提供了对软件体系结构的精确的文本表示，同时保留了直观的框线图模型。SADL语言明确区分了多种体系结构对象，如构件和连接件，并明确了它们的使用目的。SADL语言不仅提供了定义体系结构的功能，而且能够定义对体系结构的特定类的约束。

4.4 典型的软件体系结构描述语言

5、SADL

SADL的一个独特方面是对体系结构层次体系(architecture hierarchy)的表示和推理。一个垂直层次体系(vertical hierarchy)的作用在于,它能过渡在体系结构抽象和传统程序设计语言中更基本的结构概念之间的差异。通常用不同的语言来描述一个垂直层次体系的每一层,反映出在表示上的变化。SADL模式支持结构求精(structural refinement),即把一个体系结构系统地转化成另一个包含不同体系结构概念的体系结构。

4.4 典型的软件体系结构描述语言

6、Aesop

Aesop是 David Garlan等研发的一种体系结构描述语言，也是体系结构设计环境。

它是应用面向对象的类型框架对体系结构进行描述，应用类定义体系结构的基本元素，用它们的子类定义体系结构风格的基本术语，运用相互连接的对象来描述软件配置。

它提供了软件体系结构的风格实现、分析和约束检测的系列工具。

4.4 典型的软件体系结构描述语言

6、Aesop

由构件、连接器、端口、角色、配置和绑定，提供了一个体系结构风格的设计空间，但不提供语义支持。

类是构造系统的基本成分，用类来表示一般的构件、连接器、端口和角色。表示体系结构成分的对象可以带有附加的规格说明，这样的规格说明本身并不代表对象，而是定义了由相应的成分构造系统时的约束，由其他工具解释和分析。

通过扩展基类来定义体系结构风格，基类代表了具有约束的体系结构成分。新的体系结构风格也可以通过已有风格的类进行扩充得到。

4.4 典型的软件体系结构描述语言

6、Aesop 不足之处:

Aesop并未提供体系结构风格描述的独立机制和定义它们的相关语义，它只是通过工具和类型系统隐含地提供相关信息。

4.4 典型的软件体系结构描述语言

7、ACME

ACME是David Garlan等研制的体系结构描述语言，它是一种简单、通用的软件体系结构描述语言，同时也作为一个体系结构交换语言，能够与不同的体系结构描述语言所描述的系统设计进行互换。

ACME由多个体系结构研究小组联合开发，旨在提供体系结构描述基本和共同的部分，从而借此实现多个体系结构描述语言的相互转换。因此，ACME更多地是作为一个体系结构语言之间转换的中介语言。

4.4 典型的软件体系结构描述语言

7、ACME

ACME的核心概念以7种类型的实体为基础：构件、连接件、系统、端口、角色、表述和表述映射。ACME支持体系结构的分级描述，特别是每个构件或连接件都能用一个或多个更详细、更低层的描述来表示。

4.4 典型的软件体系结构描述语言

7、ACME

元素名	功能
构件 (Component)	描述系统中的基本计算单元和数据存储单元
连接件 (Connector)	描述构件间相互作用的连接规制,通常是构件间的通信和协调活动
系统 (System)	描述构件和连接件之间的组合结构
端口 (Port)	描述构件与其环境进行交互的界面元素或接口
角色 (Role)	描述连接件的界面元素或接口
表达 (Representation)	根据要求对同一实体的多个层次进行观察、分析和表示
表达连接图 (Representation-Map)	用表格形式描述表达实体层间的关系连接

4.4 典型的软件体系结构描述语言

7、ACME

ACME的7种元素足以用来定义体系结构的层次结构，但是，体系结构还包含有许多其他附加信息，而且不同的ADL里增加的描述信息不尽相同。作为一种用于交流的ADL，ACME为了解决这一问题，使用属性列表来表示对于结构的说明信息。

每个元素附带一个属性表，描述它们的具体属性，每一个属性由名字、可选类型和值构成，对体系结构元素进行注释。从ACME的观点来看，属性是不被解释的值。仅当属性被开发工具用于分析、转换、操作等时，它才是有用的。

4.4 典型的软件体系结构描述语言

7、ACME

ACME并没有对体系结构的构成进行抽象。这会导致在描述复杂系统时，需要反复对一些公共结构做规格说明。

为此，ACME语言引入了模板机制，它是一种类型化、参数化的宏，用于对反复出现的模式做规格说明。在应用或初始化这些模式时，只要给它们提供适当类型的参数就可以了。模板定义了句法结构，可以被扩展到需要生成新的声明的位置。在ACME中，风格定义了相关的模板集合，提供了一种在体系结构设计中捕捉并复用公共结构的机制。

4.4 典型的软件体系结构描述语言

7、ACME

ACME主要考虑的是体系结构的构造，因此并不包含体系结构的计算语义，而是依靠一个开放的语义框架。这个框架提供了基本的结构语义，它用构建属性的方式，允许特定的ADL把体系结构和运行时的行为结合起来。开放语义框架提供了从语言的结构外观对基于关系和约束的逻辑形式的直接映射。在这个框架里，ACME规格说明表示一个由此推导出的逻辑谓词，称为它的指示规则。这个谓词可以用于逻辑推理，或用于和规格说明所试图描述的现实世界中的事物做逼真度的比较。

4.4 典型的软件体系结构描述语言

8、 Darwin

Darwin是Magee和Kramer开发的体系结构描述语言，是一种陈述式语言，运用 π 演算提供系统结构规约的语义，它为一类系统提供通用的说明符号，这类系统由使用不同的交互机制的组件组成。

4.4 典型的软件体系结构描述语言

8、 Darwin

Darwin通过接口定义构件类型，接口包括提供服务接口和请求服务接口，系统配置则定义构件实例并在提供服务接口和请求服务接口之间建立绑定。

构件提供的服务作为一个名字(Name)，绑定作为一个进程，它传输该服务名字到请求该服务的构件。

Darwin提供延迟实例化(Lazy instantiation)和直接动态实例化(Direct dynamic instantiation)两种技术支持动态演化，但存在局限。

4.4 典型的软件体系结构描述语言

8、 Darwin

不足之处:

- 1) 未提供任何方法和技术描述构件或其相关服务的性质， Darwin服务类型的语义由底层平台实现决定， 没有相关语义解释。因此Darwin并不能提供体系结构行为分析的基础和方法；
- 2) 未提供显式的连接件概念。它的提供/请求连接模型仅仅支持非对称的交互模式， 并且不能独立于构件进行描述；

4.4 典型的软件体系结构描述语言

8、 Darwin

不足之处:

3) 定义体系结构风格的时候，通常是定义它的交互模型，而把构件的定义留给体系结构师。因此，Darwin不能有效支持体系结构风格的规约；

4) 运用参数化配置方法描述一类系统的规约，同样存在局限，不能很好支持体系结构风格规约。

4.4 典型的软件体系结构描述语言

9、XYZ/ADL

XYZ/ADL是中国科学院软件研究所唐稚松院士等提出和开发的体系结构形式化描述语言。XYZ/ADL的形式化机制是他们提出的线性时序逻辑语言XYZ/E。XYZ/ADL运用Pre-Post断言描述构件和连接件的逻辑功能，并运用XYZ/E的文本描述，结合图形表达方式，描述复杂构件和连接件的内部计算行为，并对常见的体系结构风格进行了描述。然而，XYZ/ADL亦有需要完善之处，例如，提供有效方法用以提升规约的抽象级别和给出相关的组装推导机制。

4.4 典型的软件体系结构描述语言

10、ABC/ADL

ABC/ADL是北大青鸟工程提出的体系结构描述语言。ABC/ADL具备大多数ADL描述软件系统的高层结构的能力，还支持系统的逐步精化与演化，并支持系统自动化的组装和验证。采用XML作为元语言，可以使用除ABC工具外的更广范围的通用XML工具存取ABC/ADL；XML也提供了软件系统的运行时刻信息和设计阶段模型之间的信息交换和反射的能力。ABC工具支持对体系结构的图形化建模，并支持图形化模型和ABC/ADL之间的转换。

4.4 典型的软件体系结构描述语言

10、ABC/ADL

ABC/ADL用两个Schema来定义体系结构的建模元素。Style.xsd定义了描述体系结构风格的语言元素：风格、构件模板、连接子模板和连接约束等；adl.xsd定义了描述体系结构的语言元素，包括构件、连接子和配置等。

4.4 典型的软件体系结构描述语言

11、FRADL

FRADL是吉林大学提出的体系结构描述语言。FRADL是以框架和角色模型为基础，吸收诸如Wright, C2, Rapid语言的语法特点的一种软件体系结构描述语言。但是FRADL与这些体系结构描述语言有较大的差别。FRADL有如下几个特点：(1)支持面向对象技术的所有特性，包括类、消息发送、继承等；(2)以框架为构件，从而支持大粒度的软件重用；(3)以角色模型为基础，构造具有控制的连接器，精确表达构件的交互协议；(4)构件的行为描述和连接器的处理描述特别适合构造分布并发对象，并支持有效的系统特性分析，如避免死锁；(5)支持不同抽象级别的结构描述，可用于系统分析和设计阶段。

4.4 典型的软件体系结构描述语言

12、A-ADL

A-ADL是东北大学提出的一种基于智能体的体系结构描述语言。A-ADL采用计算/联接智能体替代部件/连接器作为体系结构单元，基于体系结构原语、规则及多视图，解决了多智能体系统的动态性和语义问题。

4.5 软件体系结构与UML

1、UML概述

公认的面面向对象建模语言出现于70年代中期，从1989年到1994年，其数量从不到十种增加到了五十多种。在众多的建模语言中，语言的创造者努力推崇自己的产品，并在实践中不断完善。但是，OO方法的用户并不了解不同建模语言的优缺点及相互之间的差异，因而很难根据应用特点选择合适的建模语言，于是爆发了一场“方法大战”。90年代中，一批新方法出现了，其中最引人注目的是Booch、OOSE和OMT-2等。

4.5 软件体系结构与UML

Booch是面向对象方法最早的倡导者之一，他提出了面向对象软件工程的概念。1991年，他将前面在Ada的工作扩展到整个面向对象设计领域。Booch 1993比较适合于系统的设计和构造。

4.5 软件体系结构与UML

Rumbaugh等人提出了面向对象的建模技术(OMT)方法，采用了面向对象的概念，并引入各种独立于语言的表示符。这种方法用对象模型、动态模型、功能模型和用例模型，共同完成对整个系统的建模，所定义的概念和符号可用于软件开发的分析、设计和实现的全过程，软件开发人员不必在开发过程的不同阶段进行概念和符号的转换。OMT-2特别适用于分析和描述以数据为中心的信息系统。

4.5 软件体系结构与UML

Jacobson于1994年提出了OOSE方法，其最大特点是面向用例，并在用例的描述中引入了外部角色的概念。用例的概念是精确描述需求的重要武器，它贯穿于整个开发过程，包括对系统的测试和验证。OOSE比较适合支持商业工程和需求分析。

此外，还有Coad/Yourdon方法，即著名的OOA/OOD，它是最早的面向对象的分析和设计方法之一。该方法简单、易学，适合于面向对象技术的初学者使用，但由于该方法在处理能力方面的局限，目前已很少使用。

4.5 软件体系结构与UML

概括起来，首先，面对众多的建模语言，用户由于没有能力区别不同语言之间的差别，因此很难找到一种比较适合其应用特点的语言；其次，众多的建模语言实际上各有千秋；第三，虽然不同的建模语言大多类同，但仍存在某些细微差别，极大地妨碍了用户之间的交流。

因此在客观上，极有必要在精心比较不同的建模语言优缺点及总结面向对象技术应用实践的基础上，组织联合设计小组，根据应用需求，取其精华，去其糟粕，求同存异，实现对建模语言的统一。

4.5 软件体系结构与UML

1994年10月，Grady Booch和Jim Rumbaugh开始致力于这一工作。他们首先将Booch 93和OMT-2统一起来，并于1995年10月发布了第一个公开版本，称之为统一方法 UM0.8(Unified Method)。1995年秋，OOSE的创始人Ivar Jacobson加盟到这一工作。经过Booch、Rumbaugh和Jacobson三人的共同努力，于1996年6月和10月分别发布了两个新的版本，即UMLO.9和UMLO.91，并将UM重新命名为UML(unified Modeling Language)。

4.5 软件体系结构与UML

1996年，一些机构将UML作为其商业策略，UML的开发者得到了来自公众的正面反应，并倡议成立了UML成员协会，以完善、加强和促进UML的定义工作。当时的成员有DEC、HP、I-Logix、IBM、ICON Computing、MCI System house、Microsoft、Oracle、Rational Software、TI、Unisys等。这一机构对UML1.0(1997年1月)及 UML1.1(1997年11月17日)的定义和发布起了重要的促进作用。

4.5 软件体系结构与UML

1996年底，UML已稳定占面向对象技术市场的85%，成为可视化建模语言事实上的工业标准。1997年11月17日，OMG采纳UML 1.1作为基于面向对象技术的标准建模语言。现在UML已经经历了1.1、1.2、1.4三个版本的演变，发布了最新的2.0版标准（目前的最新版本是2.4.1）。

4.5 软件体系结构与UML

UML是一种定义良好、易于表达、功能强大且普遍适用的建模语言。它溶入了软件工程领域的新思想、新方法和新技术。它的作用域不限于支持面向对象的分析与设计，还支持从需求分析开始的软件开发的全过程。

4.5 软件体系结构与UML

首先，UML融合了Booch、OMT和OOSE方法中的基本概念，而且这些基本概念与其他面向对象技术中的基本概念大多相同，因而，UML必然成为这些方法以及其他方法的使用者乐于采用的一种简单一致的建模语言；

其次，UML不仅仅是上述方法的简单汇合，而是在这些方法的基础上广泛征求意见，集众家之长，几经修改而完成的，UML扩展了现有方法的应用范围；

第三，UML是标准的建模语言，而不是标准的开发过程。尽管UML的应用必然以系统的开发过程为背景，但由于不同的组织和不同的应用领域，需要采取不同的开发过程。

4.5 软件体系结构与UML

2、UML的主要内容

UML是一种明确定义了语法和语义的可视化建模语言，它基于主流的软件开发方法及开发经验，语法和语义由元模型、自然语言和约束来说明。

作为一种建模语言，UML的定义包括UML语义和UML表示法两个部分。

4.5 软件体系结构与UML

(1) UML语义

UML语义的定义采用了形式化技术，但并不是完全形式化的定义。UML对语法结构给出了精确的定义，对其动态语义则是用自然语言描述的。UML的语法是一种与表示符号无关的抽象语法，它可以映射到不同的符号体系中。尽管没有完全形式化，但其定义方式颇为复杂：采用了四级元模型体系结构，即：

4.5 软件体系结构与UML

(1) UML语义

1)元元模型 (Meta-Meta Model)，是元模型的基础体系结构，定义一种说明元模型的语言；

2)元模型 (Meta Model)，元元模型的一个实例，定义一种说明模型的语言；

3)模型(Model)，元模型的一个实例，定义一种描述信息领域的语言；

4)用户对象 (User Object)，模型的一个实例，定义一个特定的信息领域。

4.5 软件体系结构与UML

对各级模型元素的定义方式是，首先给出它的抽象语法(采用UML的类图表示法描述元素之间的关系)，然后给出其形式化规则(采用正文和对象约束语言)，最后描述其语义(采用准确的正文)。UML按这种方式总共定义了118个元素，划分到3个包和9个子包中分别加以定义。

4.5 软件体系结构与UML

语义约束用对象约束语言OCL规约表示。OCL基于一阶谓词逻辑，每个OCL表达式都以UML模型元素为背景，用于说明一些在可视化的系统模型中不能充分表达的建模信息。OCL是一种形式化的语言，它并不表示实现方面的问题，只用于对模型作详细说明，也不可编译执行。

UML模型从以下几个方面说明软件系统：类及其属性、操作和类之间的静态关系，类的包及其依赖关系，类的状态及其行为，对象之间的交互行为等。

4.5 软件体系结构与UML

元模型为UML的所有元素在语法和语义上提供了简单、一致、通用的定义性说明，使开发者能在语义上取得一致，消除了因人而异的表达方法所造成的影响。此外UML还支持对元模型的扩展定义。

4.5 软件体系结构与UML

(2) UML表示法

UML表示法为开发者或开发工具使用这些图形符号提供了文本语法，为系统建模提供了标准。这些图形符号和文字所表达的是应用级的模型，在语义上它是UML元模型的实例。

4.5 软件体系结构与UML

(2) UML表示法

UML2.0支持13种图，可以分成两大类：结构图和行为图。与UML 1.X相比，组合结构图、交互概览图等都是新增的。而原来的协作图改名为通信图，状态图改名为状态机图。原来的集合词汇实现图(Implementation Diagram)被取消。

结构图包括：类图、组合结构图、构件图、部署图、对象图和包图。

行为图包括：活动图、交互图、用例图 and 状态机图，其中交互图是顺序图、通信图、交互概览图和时序图的统称。

4.5 软件体系结构与UML

1) 用例图

用例图主要用来图示化系统的主事件流程，它主要用来描述客户的需求，即用户希望系统具备的完成一定功能的动作，通俗地理解，用例就是软件的功能模块，所以是设计系统分析阶段的起点，设计人员根据客户的需求来创建和解释用例图，用来描述软件应具备哪些功能模块以及这些模块之间的调用关系。

用例图包含了用例和参与者，用例之间用关联来连接，以求把系统的整个结构和功能反映给非技术人员（通常是软件的用户），对应的是软件的结构和功能分解。

4.5 软件体系结构与UML

1) 用例图

参与者不是特指人，是指系统以外的，在使用系统或与系统交互中所扮演的角色。因此参与者可以是人，可以是事物，也可以是时间或其他系统等。还有一点要注意的是，参与者不是指人或事物本身，而是表示人或事物当时所扮演的角色。

用例是对包括变量在内的一组动作序列的描述，系统执行这些动作，并产生传递特定参与者的价值的可观察结果。

4.5 软件体系结构与UML

1) 用例图

单个参与者可与多个用例联系；反过来，一个用例可与多个参与者联系。对同一个用例而言，不同参与者有着不同的作用：他们可以从用例中取值，也可以参与到用例中。

用例是从系统外部可见的行为，是系统为某一个或几个参与者（Actor）提供的一段完整的服务。从原则上来讲，用例之间都是独立、并列的，它们之间并不存在着包含从属关系。但是为了体现一些用例之间的业务关系，提高可维护性和一致性，用例之间可以抽象出包含(include)、扩展(extend)和泛(generalization)几种关系。

4.5 软件体系结构与UML

1) 用例图

包含关系：使用包含（**Inclusion**）用例来封装一组跨越多个用例的相似动作（行为片断），以便多个基（**Base**）用例复用。基用例控制与包含用例的关系，以及被包含用例的事件流是否会插入到基用例的事件流中。基用例可以依赖包含用例执行的结果，但是双方都不能访问对方的属性。

4.5 软件体系结构与UML

1) 用例图

扩展关系：将基用例中一段相对独立并且可选的动作，用扩展（**Extension**）用例加以封装，再让它从基用例中声明的扩展点（**Extension Point**）上进行扩展，从而使基用例行为更简练和目标更集中。扩展用例为基用例添加新的行为。扩展用例可以访问基用例的属性，因此它可以根据基用例中扩展点的当前状态来判断是否执行自己。

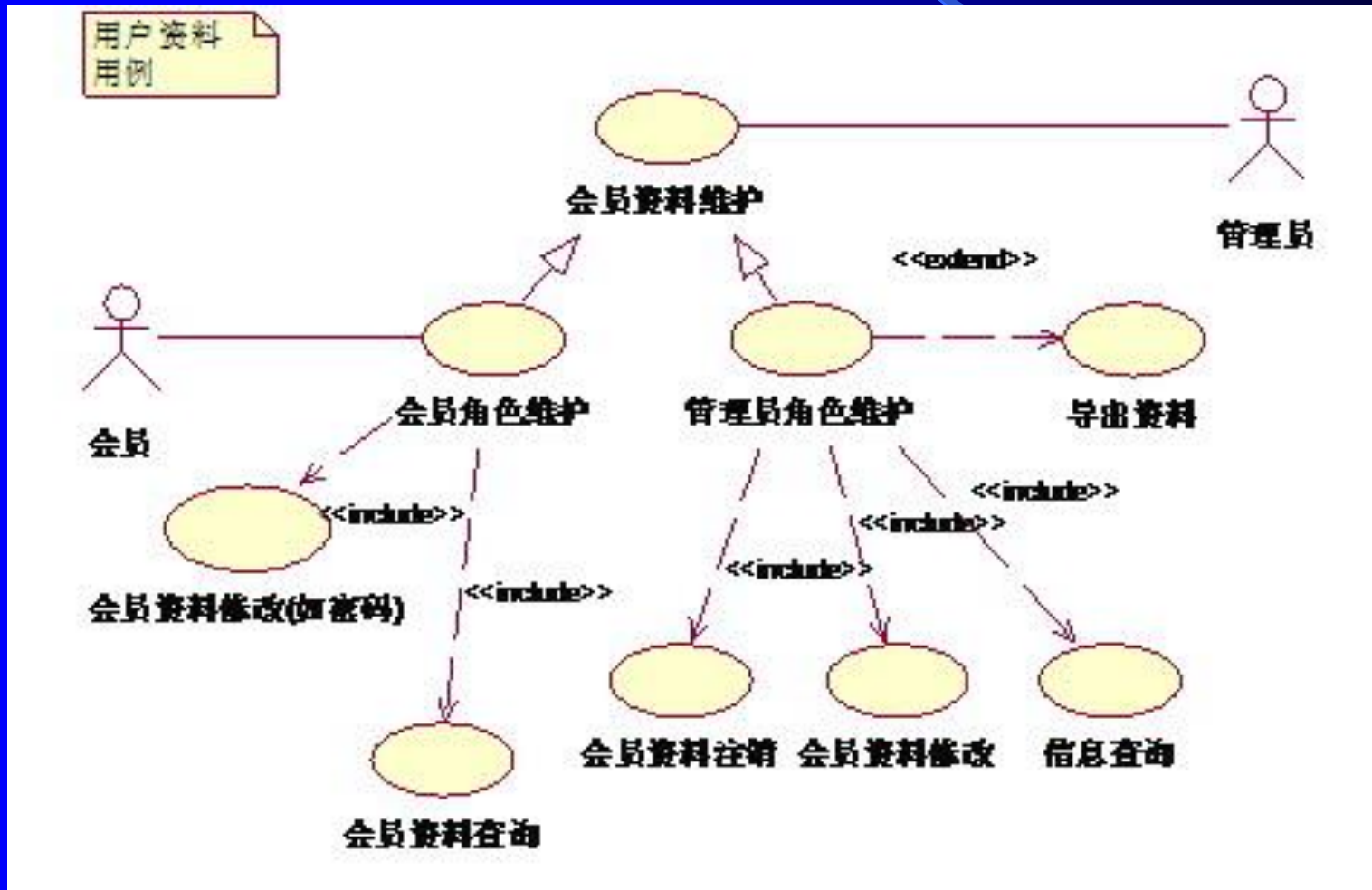
4.5 软件体系结构与UML

1) 用例图

泛化关系：子用例和父用例相似，但表现出更特别的行为；子用例将继承父用例的所有结构、行为和关系。子用例可以使用父用例的一段行为，也可以重载它。父用例通常是抽象的。在实际应用中很少使用泛化关系，子用例中的特殊行为都可以作为父用例中的备选流存在。

4.5 软件体系结构与UML

1) 用例图



4.5 软件体系结构与UML

2) 类图

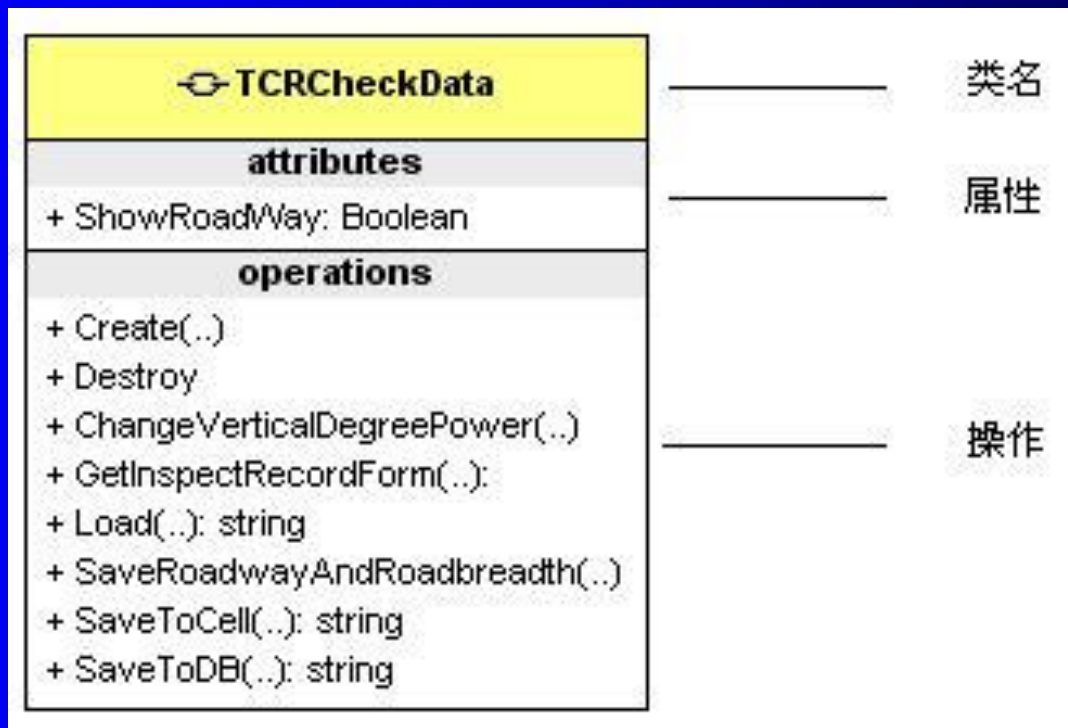
类是具有相似结构、行为和关系的一组对象的描述符。类是面向对象系统中最重要构造块。

类图用来描述类和类之间的静态关系，在系统的整个生命周期都是有效的。与数据模型不同，它不仅显示了信息的结构，同时还描述了系统的行为。类图是定义其它图的基础。在类图的基础上，状态图、协作图等进一步描述了系统其他方面的特性。

4.5 软件体系结构与UML

2) 类图

类图用矩形框来描述类，包括名称、属性和操作三个部分。



4.5 软件体系结构与UML

2) 类图

a) 名称：每个类都必须有一个名字，用来区分其它的类。类名是一个字符串，称为简单名字。

4.5 软件体系结构与UML

2) 类图

b) 属性：是指类的命名的特性，常常代表一类取值。类可以有任意多个属性，也可以没有属性。可以只写上属性名，也可以在属性名后跟上类型甚至缺省取值。根据图的详细程度，每条属性可以包括属性的可见性、属性名称、类型、缺省值和约束特性。UML规定类的属性的语法为：

“可见性 属性名： 类型 = 缺省值 {约束特性}”。

4.5 软件体系结构与UML

2) 类图

常用的可见性有Public、Private和Protected三种，在UML中分别表示为“+”、“-”和“#”。

类型表示该属性的种类，它可以是基本数据类型，例如整数、实数、布尔型等，也可以是用户自定义的类型，一般它由所涉及的程序设计语言确定。

约束特性则是用户对该属性性质一个约束的说明，例如“{只读}”说明该属性是只读属性。

4.5 软件体系结构与UML

2) 类图

c) 操作：是类的任意一个实例对象都可以调用的，并可能影响该对象行为的实现。该项可省略。操作用于修改、检索类的属性或执行某些动作。它们被约束在类的内部，只能作用到该类的对象上。

UML规定操作的语法为：

可见性 操作名 (参数表)： 返回类型 {约束特性}。

4.5 软件体系结构与UML

2) 类图

类图用类之间的连线来表示关系。

- a) 依赖 (Dependency)
- b) 关联 (Association)
- c) 聚合 (Aggregation)
- d) 合成 (Composition)
- e) 泛化 (Generalization)
- f) 实现 (Realization)

4.5 软件体系结构与UML

2) 类图

a) 依赖 (Dependency)

实体之间一个“使用”关系暗示一个实体的规范发生变化后，可能影响依赖于它的其他实例。更具体地说，它可转换为对不在实例作用域内的一个类或对象的任何类型的引用。其中包括一个局部变量，对通过方法调用而获得的一个对象的引用，或者对一个类的静态方法的引用（同时不存在那个类的一个实例）。也可利用“依赖”来表示包和包之间的关系。由于包中含有类，所以你可根据那些包中的各个类之间的关系，表示出包和包的关系。

4.5 软件体系结构与UML

2) 类图

a) 依赖 (Dependency)

Java	UML
<pre>public class Employee { public void calcSalary(CalculatorStrategy { ... } }</pre>	<pre>classDiagram Employee ..> Calculator</pre> <p>The UML diagram shows two class boxes. The left box is labeled 'Employee' and the right box is labeled 'Calculator'. A dashed arrow points from the 'Employee' box to the 'Calculator' box, indicating a dependency.</p>

4.5 软件体系结构与UML

2) 类图

b) 关联 (Association)

实体之间的一个结构化关系，表明对象是相互连接的。箭头是可选的，它用于指定导航能力。如果没有箭头，暗示是一种双向的导航能力。在Java中，关联转换为一个实例作用域的变量。可为一个关联附加其他修饰符。多重性 (Multiplicity) 修饰符暗示着实例之间的关系。在示范代码中，Employee可以有0个或更多的TimeCard对象。但是，每个TimeCard只从属于单独一个Employee。

4.5 软件体系结构与UML

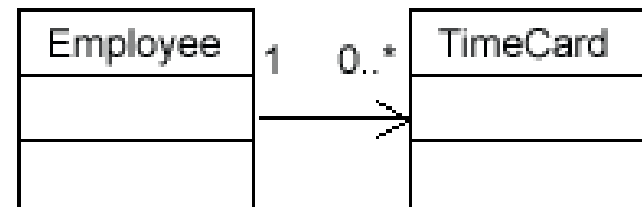
2) 类图

b) 关联 (Association)

Java

```
public class Employee {  
    private TimeCard _tc;  
    public void maintainTimeCard() {  
        ...  
    }  
}
```

UML



4.5 软件体系结构与UML

2) 类图

c) 聚合 (Aggregation)

聚合是关联的一种形式，代表两个类之间的整体/局部关系。聚合暗示着整体在概念上处于比局部更高的一个级别，而关联暗示两个类在概念上位于相同的级别。聚合也转换成Java中的一个实例作用域变量。

关联和聚合的区别纯粹是概念上的，而且严格反映在语义上。聚合还暗示着实例图中不存在回路。换言之，只能是一种单向关系。

4.5 软件体系结构与UML

2) 类图

c) 聚合 (Aggregation)

Java	UML
<pre>public class Employee { private EmpType et; public EmpType getEmpType() { ... } }</pre>	<pre>classDiagram Employee "1" o-- "0..*" EmpType</pre>

4.5 软件体系结构与UML

2) 类图

d) 合成 (Composition)

合成是聚合的一种特殊形式，暗示“局部”在“整体”内部的生存期职责。合成也是非共享的。所以，虽然局部不一定要随整体的销毁而被销毁，但整体要么负责保持局部的存活状态，要么负责将其销毁。

局部不可与其他整体共享。但是，整体可将所有权转交给另一个对象，后者随即将承担生存期职责。Employee和TimeCard的关系或许更适合表示成“合成”，而不是表示成“关联”。

4.5 软件体系结构与UML

2) 类图

d) 合成 (Composition)

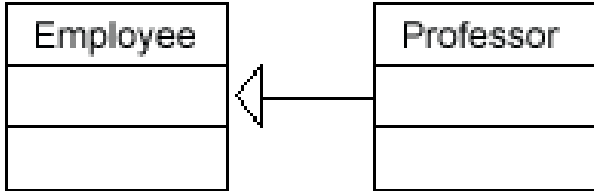
Java	UML
<pre>public class Employee { private TimeCard tc; public void maintainTimeCard() { ... } }</pre>	<pre>classDiagram Employee "1" *-- "0..*" TimeCard</pre>

4.5 软件体系结构与UML

2) 类图

e) 泛化 (Generalization)

泛化表示一个更泛化的元素和一个更具体的元素之间的关系。泛化是用于对继承进行建模的UML元素。

Java	UML
<pre>public abstract class Employee { } public class Professor extends Employee { }</pre>	 <pre>classDiagram Employee < -- Professor</pre> <p>The UML diagram illustrates a generalization relationship between two classes. On the left is the 'Employee' class, and on the right is the 'Professor' class. A solid line with an open arrowhead points from the 'Professor' class to the 'Employee' class, indicating that 'Professor' is a specialized version of 'Employee'.</p>

4.5 软件体系结构与UML

2) 类图

f) 实现 (Realization)

实现关系指定两个实体之间的一个合同。换言之，一个实体定义一个合同，而另一个实体保证履行该合同。

Java	UML
<pre>public interface CollegePerson { } public class Professor implements CollegePers }</pre>	<pre>classDiagram CollegePerson < .. Professor</pre> <p>The UML diagram shows two class boxes. The left box is labeled 'CollegePerson' and the right box is labeled 'Professor'. A dashed line with an open triangle arrowhead points from the 'Professor' box to the 'CollegePerson' box, indicating a realization relationship.</p>

4.5 软件体系结构与UML

3) 对象图

对象图用于描述被建模系统的模型元素实例之间的结构化信息，其所表达的是特定时间被建模系统在结构上的部分或是全部视图。

一个UML对象图主要集中在特定的对象实例和其槽，以及实例之间的连接。

同一个类图所对应的对象图可以有多个，多个对象图合在一起共同展示了随着时间的推移，在不同时间点系统的对象状态。

4.5 软件体系结构与UML

3) 对象图

与类图的抽象性相比，对象图是具体的，其通常用来提供所对应类图的结构示例，或者作为所对应类图的测试用例。

应当说每一幅对象图都有其侧重点，因而，每一幅对象图应当只侧重表达其所侧重的内容。

4.5 软件体系结构与UML

3) 对象图

在UML中，对象实例是采用实例规范（instance specification）来表示的，对象实例所具有的结构化特性（feature）是采用槽（slot）来表示的，对象实例与对象实例之间的关系则是采用连接（link）来表示。

与类图相对比，实例对应于类对象，槽对应于类属性的实例，而连接则对应于类与类之间关联的实例。有了这些对应关系，在掌握了类图的情况下，就更加的容易理解和掌握对象图了。

4.5 软件体系结构与UML

3) 对象图

对象实例格式为

对象名: 类名

注: 类名和对象名下面有下划线;

槽格式为

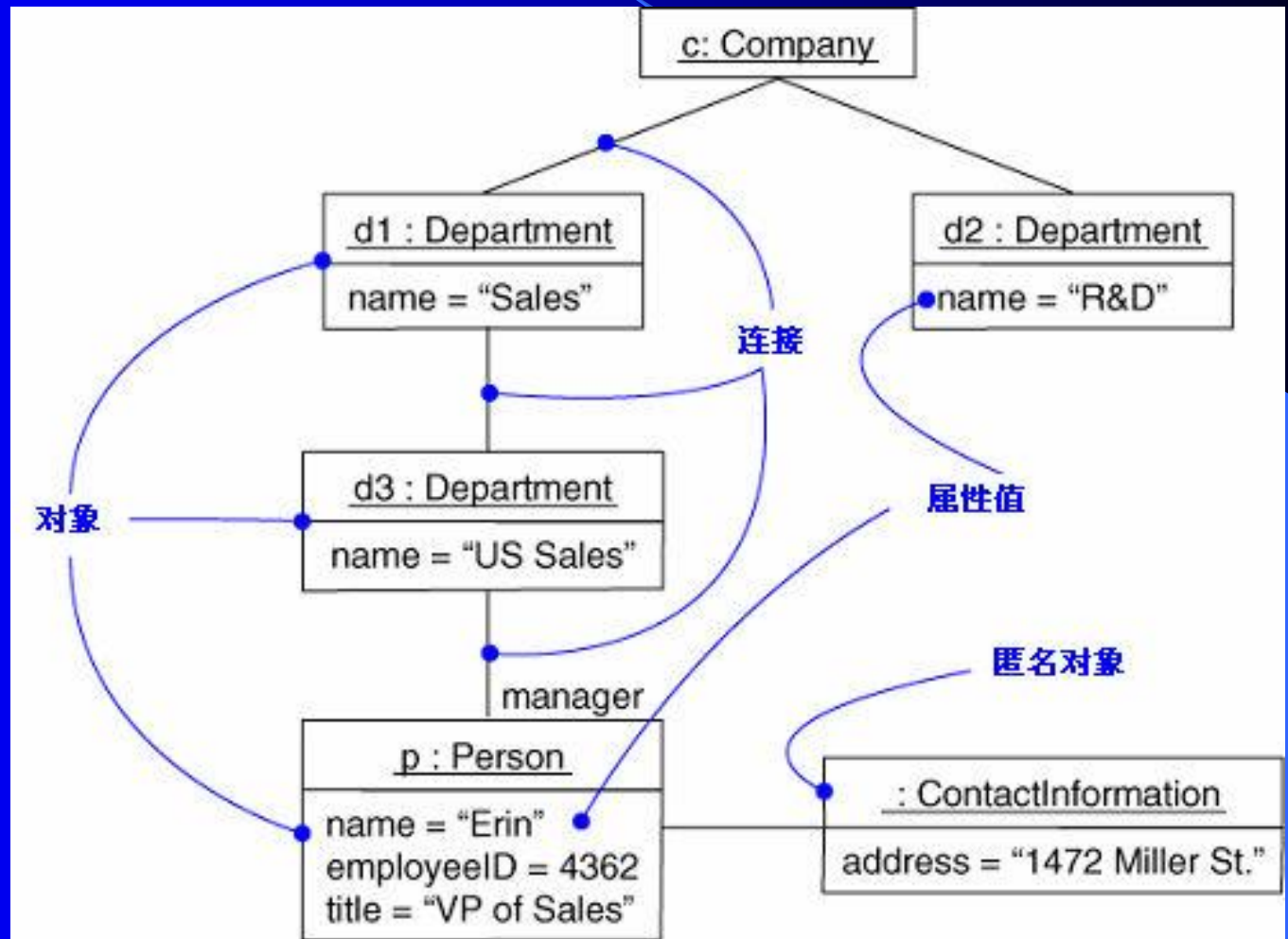
属性: 类型=值

注: 类型可以省略。

对象使用链连接、链拥有名称、角色, 但是没有多重性。对象代表的是单独的实体, 所有的链都是一对一的, 因此不涉及到多重性。

4.5 软件体系结构与UML

3) 对象图



4.5 软件体系结构与UML

4) 顺序图

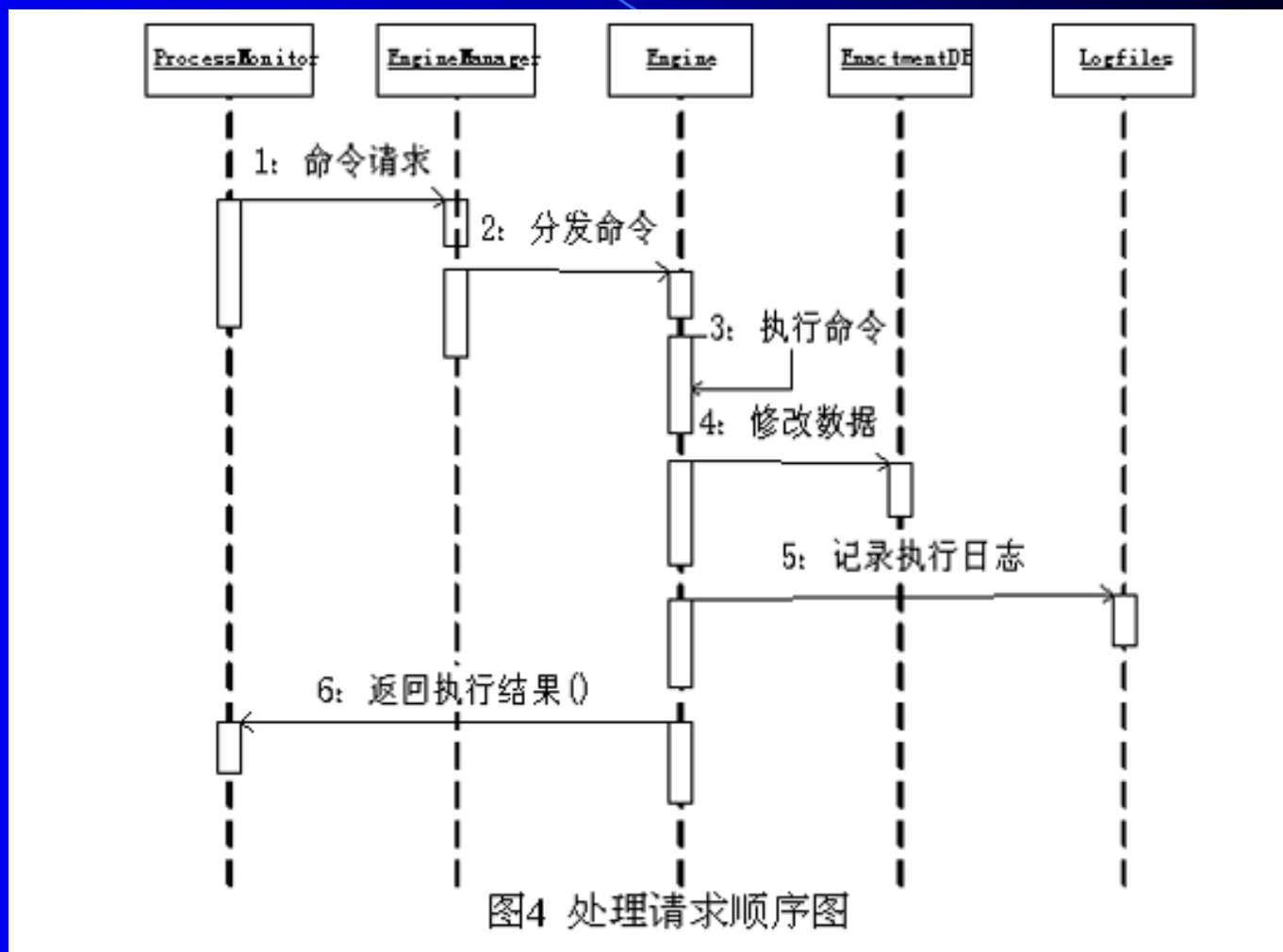
顺序图用来反映若干个对象之间的动态协作关系，也就是随着时间的推移，对象之间是如何交互的。

顺序图是将交互关系表示为一个二维图。纵向是时间轴，时间沿竖线向下延伸。横向轴代表了在协作中各独立对象的类元角色。类元角色用生命线表示。当对象存在时，角色用一条虚线表示，当对象的过程处于激活状态时，生命线是一个双道线。

消息显示为从一个角色生命线出发至另一个角色生命线的箭头，箭头用从上而下来的时间顺序来安排。

4.5 软件体系结构与UML

4) 顺序图



4.5 软件体系结构与UML

5) 通信图

UML2.0 通信图是对 UML1.x 协作图的改进，与协作图相比较，通信图中增加了消息的嵌套和并发表示，这样可以使行为被重用。

顺序图基于时间，按时间顺序显示出现的任务；而通信图显示任务和消息(对象)的交互方式。

4.5 软件体系结构与UML

5) 通信图

UML通信图用于描述一组相互协作的对象间的交互与链接。链接显示了实际对象以及这些对象之间是如何联系在一起的。UML通信图可以用于展示操作的执行、用例的执行或仅仅是系统中的一个交互情节，它开始于一个用于启动整个交互或协作的消息（如一个操作调用）。在UML通信图中，涉及到以下几个主要概念：

4.5 软件体系结构与UML

5) 通信图

a) 类：说明一系列拥有相同的属性、操作、方法、关系、行为的对象集。用一个方框表示，方框里显示的是类的名字。

b) 实例：具有身份和值的独立实体。也用方框表示，但要在类名下面加横线，在类名前面加冒号。

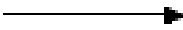



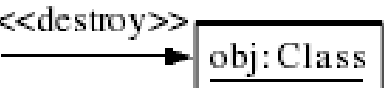
c) 链：是连接两个实例之间的连线。它表明在两个实例之间具有某种形式的可见性联系和导航关系。

d) 消息：从一个对象到另一个对象的信息的传递。可以用依附于链的带标记的箭头表示。

4.5 软件体系结构与UML

5) 通信图

表 1 顺序图中主要的消息箭头图符及含义

类型名称	图符	含义
同步消息 (Synchronization)		发送者等待接收者结束执行所要求的操作
异步消息 (Asynchronism)		发送者发送消息,不等待接收者的返回,并继续执行
返回消息 (Return)		前一消息的接收者返回控制焦点给消息的发送者
对象创建 (CreateObject)		发送者创建由接收者说明的对象
对象删除 (DeleteObject)		发送者销毁接收者说明的对象

4.5 软件体系结构与UML

5) 通信图

其语法为: [sequence-expression] [guardcondition]
[return-value]: message-name ([argument])。

Sequence-expression 表示顺序表达式, 表达式中的每个顺序项表示交互中的一个嵌套层次, 如1.2.3;

guardcondition为消息发送的守卫条件, 当
GuardType=T时, guardcondition的值为真;

return-value是在有返回值的情况下表示消息返回值的名称;

message-name 表示目标对象中引发的事件名称;
argument 是消息的参量列表。

4.5 软件体系结构与UML

5) 通信图

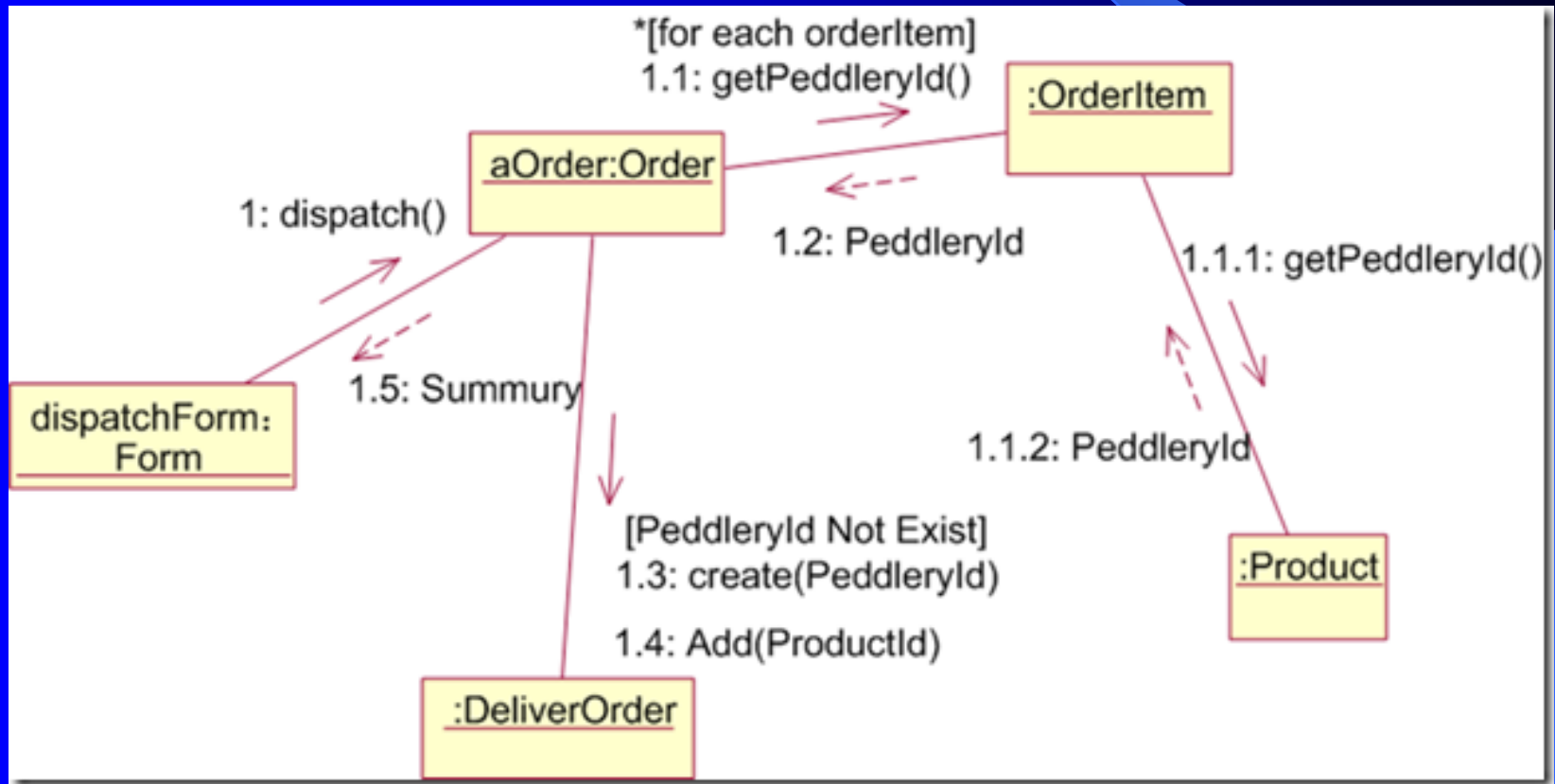
有时消息的发送需要满足某些守卫条件，因此，可以在顺序表达式后添加守卫条件 `guardcondition` 选项，主要的守卫条件类型及其描述如表 2 所示。

表 2 通信图中主要的守卫条件类型描述

名称	表示形式	描述
条件(Condition)	<code>guard</code>	消息只有在特定条件的值为真时才被调用。 <code>guard</code> 通常被表示为布尔约束，在消息发送时判断其是否被满足。
迭代(Iterative)	<code>*[iteration-clause]</code>	消息被重复执行， <code>iteration-clause</code> 为迭代子句，它可以用伪代码或程序语言表示。
并行迭代(ParallelIterative)	<code>* [iteration-clause]</code>	消息在迭代过程中是并发执行的。

4.5 软件体系结构与UML

5) 通信图



4.5 软件体系结构与UML

6) 状态机图

状态机图描述类的对象所有可能的状态以及事件发生时状态的转移条件。通常，状态机图是对类图的补充。

4.5 软件体系结构与UML

6) 状态机图

UML状态机图通过对类对象的生存周期建立模型来描述对象随时间变化的动态行为。

每一个对象都被看作是通过事件进行探测并做出回应来与外界其他部分通信的独立的实体。

事件表示对象可以探测到的事物的一种运动变化——如接收到从一个对象到另一个对象的调用或信号、某些值的改变或一个时间段的终结。任何影响对象的事物都可以是事件，真实世界所发生的事件的模型通过从外部世界到系统的信号来建造的。

4.5 软件体系结构与UML

6) 状态机图

状态是给定类的对象的一组属性值，这组属性值对所发生的事件具有相同性质的反应。换言之，处于相同状态的对象对同一事件具有同样方式的反应，所以当给定状态下的多个对象接收到相同事件时会执行相同的动作，然而处于不同状态下的对象会通过不同的动作对同一事件做出不同的反应。例如，当自动答复机处于处理事务状态或空闲状态时会对取消键做出不同的反应。

4.5 软件体系结构与UML

6) 状态机图

状态机用于描述类的行为，但它们也描述用例、协作和方法的动态行为。对于这些对象方面而言，一个状态代表了执行中的一步。我们通常用类和对象来描述状态机，但是它也可以被其他元素所直接应用。

在状态机图中，状态用带圆角的长方形表示，初始状态用实心填充的圆表示，结束状态用实心填充的圆外套一个圆圈表示。

UML状态机图的组成状态是一个被分解成顺序的或并发的子状态的状态。

4.5 软件体

6) 状态机图

状态种类	描 述	表 示 法
简单状态	没有子结构的状态	
并发组成状态	被分成两个或多个并发子状态的状态，当组成状态被激活时，所有的子状态均被并发激活	
顺序组成状态	包含一个或多个不连接的子状态的状态，特别是当组成状态被激活时，子状态也被激活	
初始状态	当状态，仅表明这是进入状态机真实状态的起点	
终止状态	特殊状态，进入此状态表明完成了状态机的状态转换历程中的所有活动	
结合状态	状态，将两个转换连接成一次就可以完成的转换	
历史状态	伪状态，它的激活保存了组成状态中先前被激活的状态	
子机器引用状态	引用子机器的状态，该子机器被隐式地插入子机器引用状态的位置	
状态	伪状态，用来在子机器引用状态中标识状态	

4.5 软件体系结构与UML

7) 活动图

活动图是一种特殊形式的状态机，用于对计算流程和工作流程建模。活动图中的状态表示计算过程中所处的各种状态，而不是普通对象的状态。通常，活动图假定在整个计算处理的过程中没有外部事件引起的中断，否则，普通的状态机更适于描述这种情况。

4.5 软件体系结构与UML

7) 活动图

活动图包含活动状态。活动状态表示过程中命令的执行或工作流程中活动的进行。与等待某一个事件发生的一般等待状态不同，活动状态等待计算处理工作的完成。当活动完成后，执行流程转入到活动图中的下一个活动状态。当一个活动的前导活动完成时，活动图中的完成转换被激发。活动状态通常没有明确表示出引起活动转换的事件，当转换出现闭包循环时，活动状态会异常终止。

4.5 软件体系结构与UML

7) 活动图

活动图也可以包含动作状态，它与活动状态有些相似，但是它们是原子活动并且当它们处于活动状态时不允许发生转换。动作状态通常用于短的记帐操作。

4.5 软件体系结构与UML

7) 活动图

活动图可以包含并发线程的分叉控制。并发线程表示能被系统中的不同对象和人并发执行的活动。通常并发源于聚集，在聚集关系中每个对象有着它们自己的线程，这些线程可并发执行。并发活动可以同时执行也可以顺序执行。

活动图不仅能够表达顺序流程控制还能够表达并发流程控制，如果排除了这一点，活动图很像一个传统的流程图。

4.5 软件体系结构与UML

7) 活动图

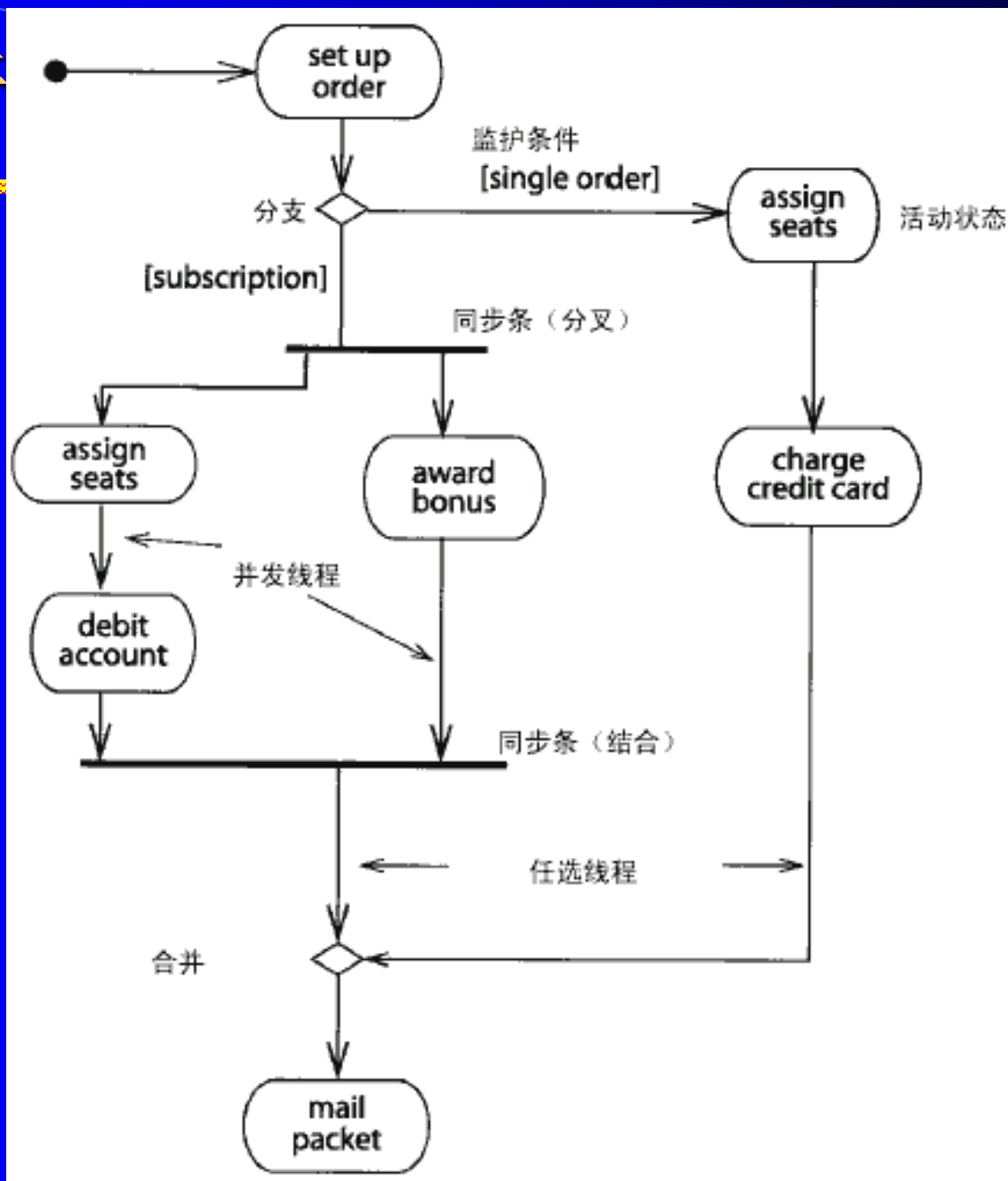
活动状态表示成带有圆形边线的矩形，它含有活动的描述（普通的状态盒为直边圆角）。

简单的完成转换用箭头表示。

分支表示转换的监护条件或具有多标记出口箭头的菱形。控制的分叉和结合与状态图中的表示法相同，是进入或离开深色同步条的多个箭头。

4.5 软件

7) 活动图



4.5 软件体系结构与UML

8) 构件图

构件图主要用于描述各种软件构件之间的依赖关系。所设计的系统中的构件的表示法及这些构件之间的关系构成了构件图。

4.5 软件体系结构与UML

8) 构件图

构件图由构件、接口、实现和依赖四部分组成：

构件：构件是系统中可替换的物理部分，它包装了实现而且遵从并提供一组接口的实现。描述了系统的一个可执行程序，一个库，一个Web程序等。

接口：接口是构件所提供服务，可以理解为一个方法，一个WebService，一个WCF，或者一个UI界面，接口可以有多个，但至少有一个，在UML中表示为一个圆形，可以在类图中对其进一步描述。

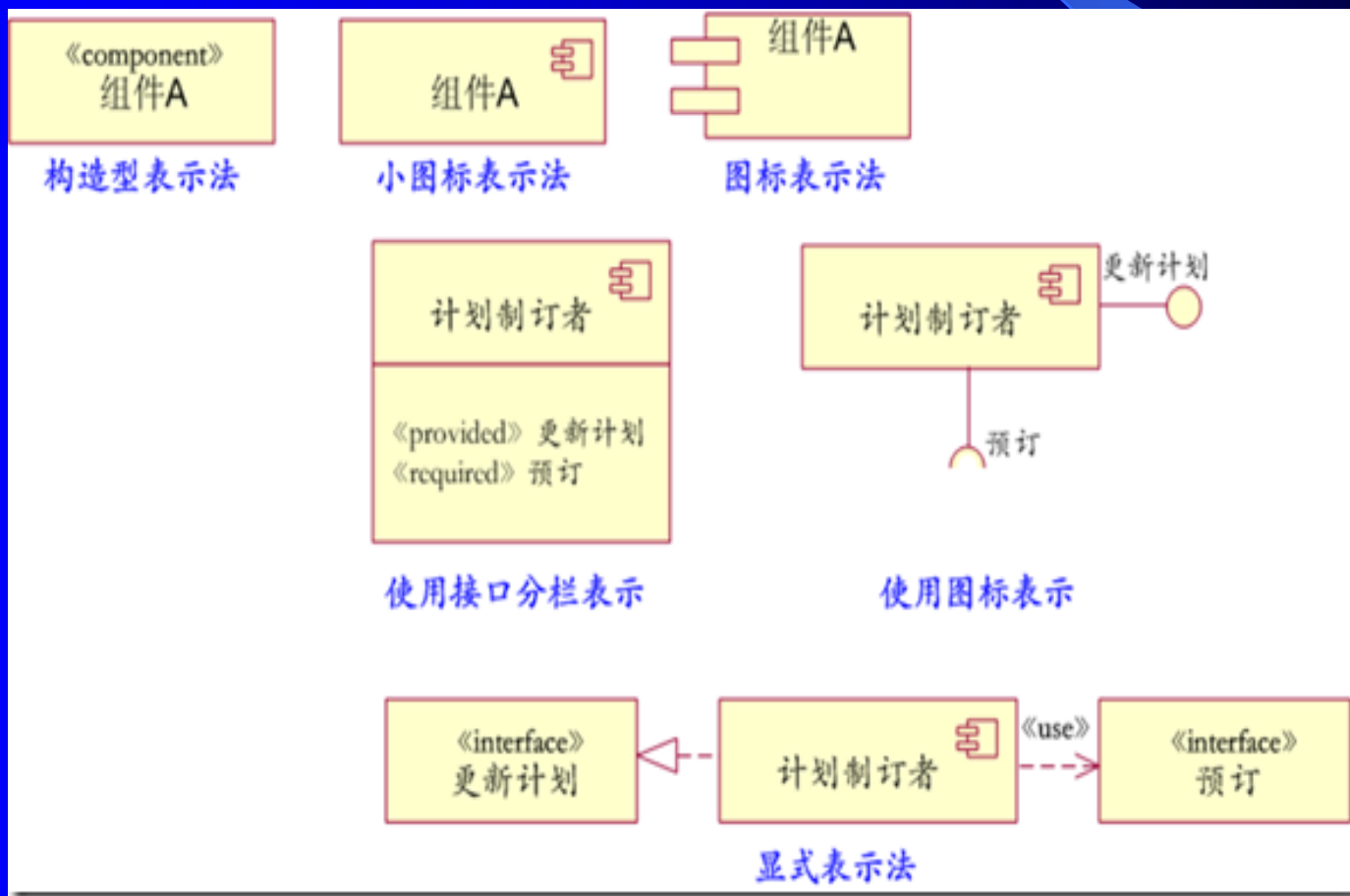
实现：实现就是构件与接口元之间的连线，代表谁实现了这个接口。

依赖：就是指组件使用了另一个组件的接口，依赖于另一个接口的存在。

4.5 软件体系结构与UML

8) 构件图

构件及接口表示法。



4.5 软件体系结构与UML

9) 部署图

一个UML部署图描述了一个运行时的硬件结点，以及在这些结点上运行的软件组件的静态视图。

部署图显示了系统的硬件（结点），安装在硬件上的软件，以及用于连接异构的机器之间的中间件。

4.5 软件体系结构与UML

9) 部署图

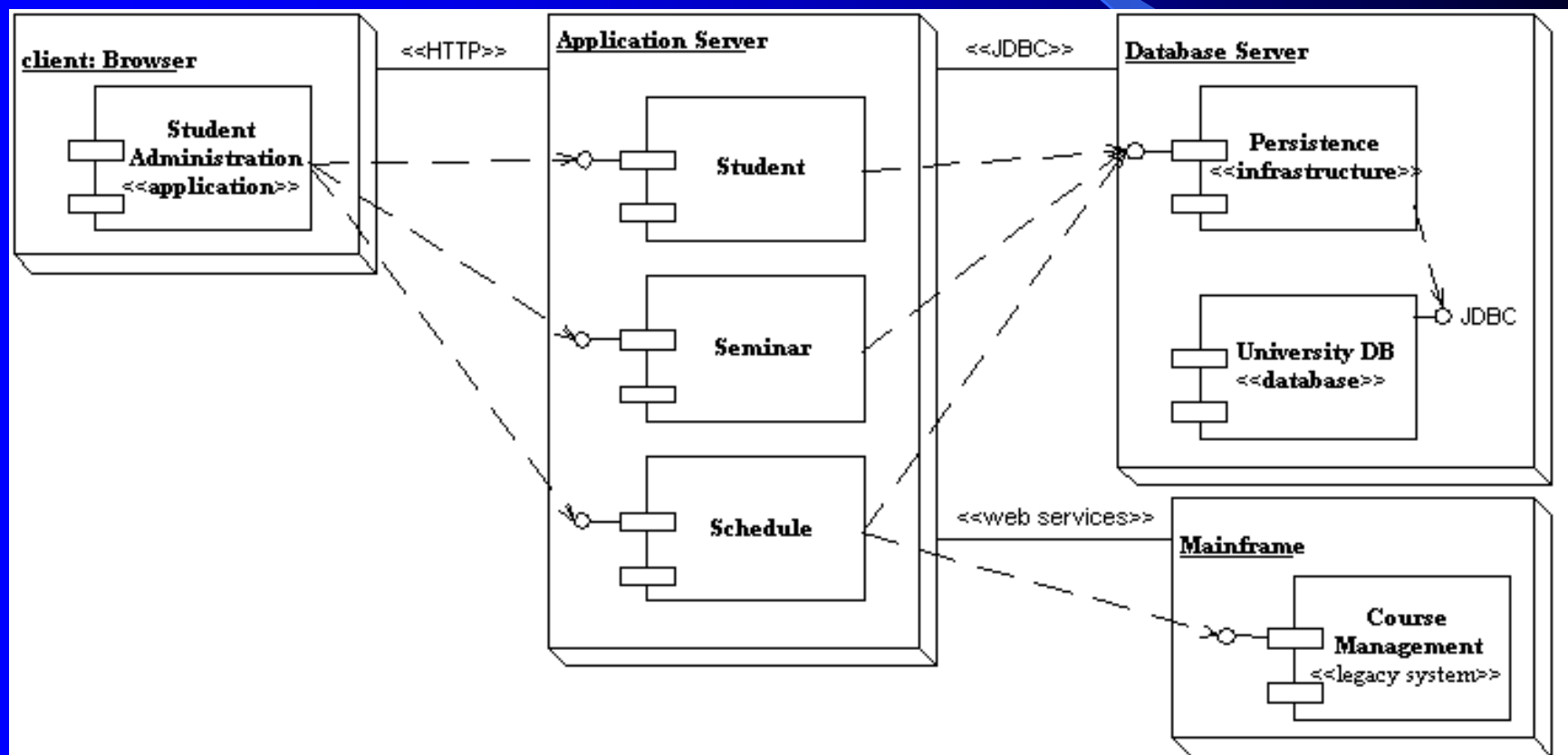
一个结点，通常描述成一个立体的盒子，表示一个计算设备，一般是一个单独的硬件设备，例如一台电脑，网络路由器，主机，传感器，或个人数字助理(PDA)。

组件，描述为矩形，左侧面还伸出两个较小矩形，这和UML组件图上使用的符号是相同的，它表示软件的中间产物，例如文件、框架、或领域组件。

通信关联，经常称为连接，被描述为连接结点间的线条。组件间的依赖则被建模成虚线箭头，这和其他UML图上使用的符号是一样的。

4.5 软件体系结构与UML

9) 部署图



4.5 软件体系结构与UML

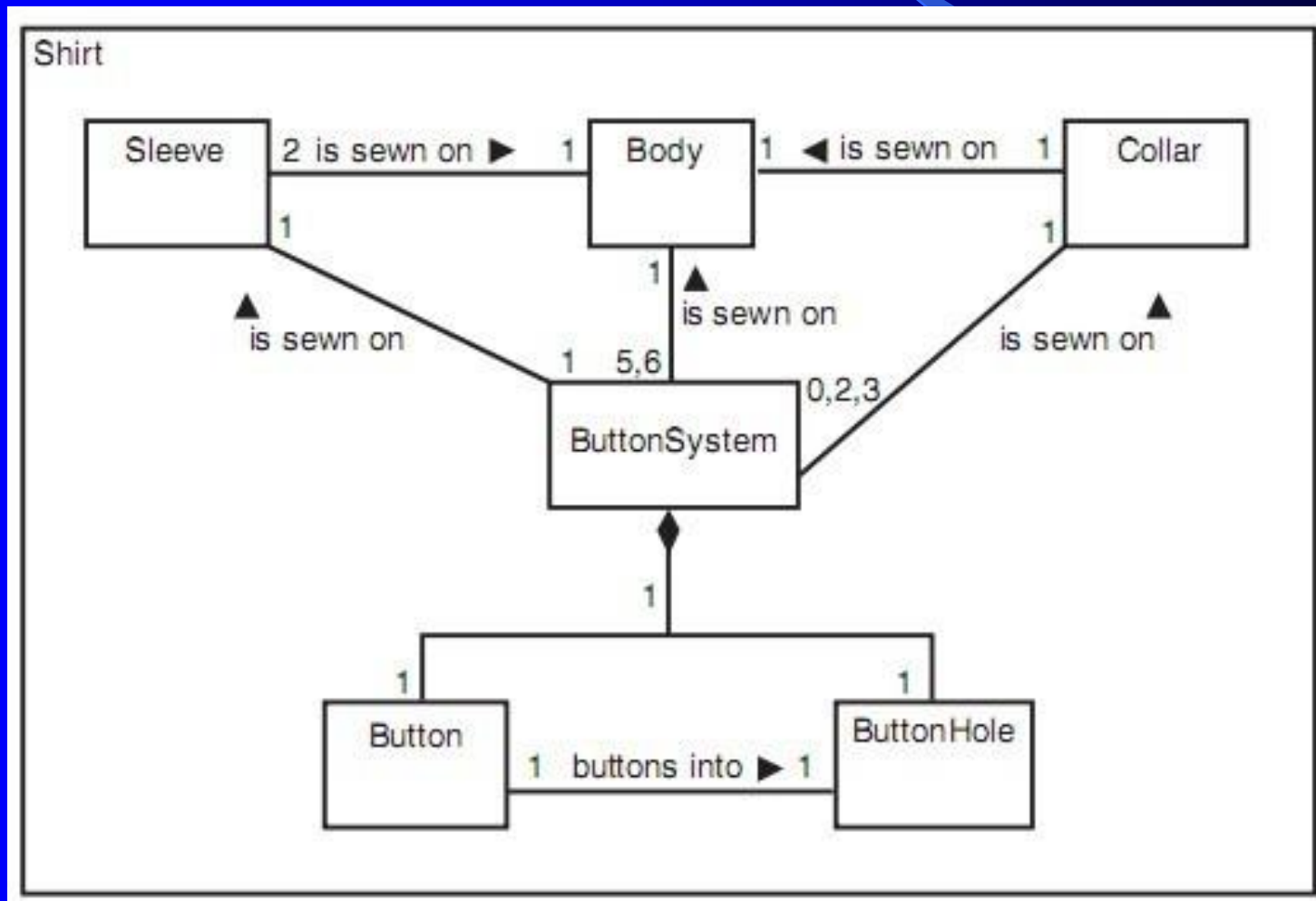
10) 组合结构图

组合结构图描述系统中的某一部分（即“组合结构”）的内部内容，包括该部分与系统其他部分的交互点，这种图能够展示该部分内容“内部”参与者的配置情况。

组合结构图中引入了一些重要的概念：例如端口（port），端口将组合结构与外部环境隔离，实现了双向的封装，既涵盖了该组合结构所提供的行为（Provided Interface），同时也指出了该组合结构所需要的服务（Required Interface）；再如协议（protocol），基于UML中的协作（collaboration）的概念，展示那些可复用的交互序列，其实质目的是描述那些可以在不同上下文环境中复用的协作模式。协议中所反映的任务由具体的端口承担。

4.5 软件体系结构与UML

10) 组合结构图



4.5 软件体系结构与UML

11) 包图

包图展现模型要素的基本组织单元，以及这些组织单元之间的依赖关系，包括引用关系（Package Import）和扩展关系（Package Merge）。在通用的建模工具中，一般可以用类图描述包图中的逻辑内容。

包可直接理解为命名空间，文件夹，是用来组织图形的封装，包图可以用来表述功能组命名空间的组织层次。

在面向对象软件开发的视角中，类显然是构建整个系统的基本构造块。但是对于庞大的应用系统而言，其包含的类将是成百上千，再加上其间“阡陌交纵”的关联关系、多重性等，必然是大大超出了人们可以处理的复杂度。这也就是引入了“包”这种分组事物构造块。

4.5 软件体系结构与UML

11) 包图

包的作用是：

- a) 对语义上相关的元素进行分组；
- b) 定义模型中的“语义边界”；
- c) 提供配置管理单元；
- d) 在设计时，提供并行工作的单元；
- e) 提供封装的命名空间，其中所有名称必须惟一

4.5 软件体系结构与UML

11) 包图

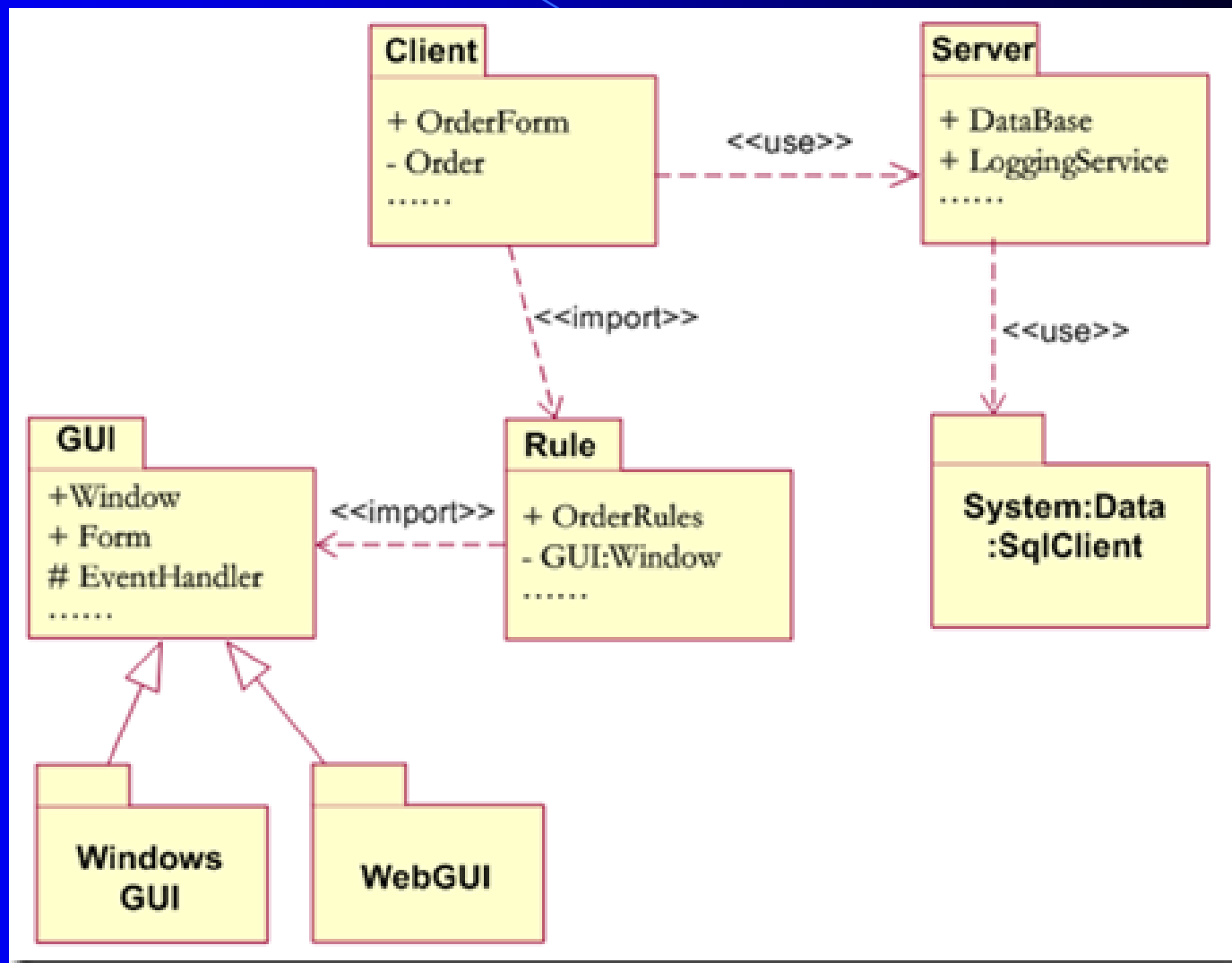
名称：每个包都必须有一个与其它包相区别的名称。

拥有的元素：在包中可以拥有各种其它元素，包括类、接口、构件、节点、协作、用例，甚至是其它包或图。如果包被撤销了，则包的元素都被撤销了。

可见性：包的可见性用“+”来表示“public”，用“#”来表示“protected”，用“-”来表示“private”。

4.5 软件体系结构与UML

11) 包图



4.5 软件体系结构与UML

12) 交互概览图

交互概览图是活动图和序列图的混合版，其主要结构像活动图，表示流程，但是参与流程的节点不是一般的动作(action)，取而代之的是“交互”(interaction)片段。

一个交互概览图是活动图的一种形式，它的节点代表交互图。交互图包含顺序图、通信图、交互概览图和时序图。大多数交互概览图标注与活动图一样。例如：起始、结束、判断、合并、分叉和结合节点是完全相同。

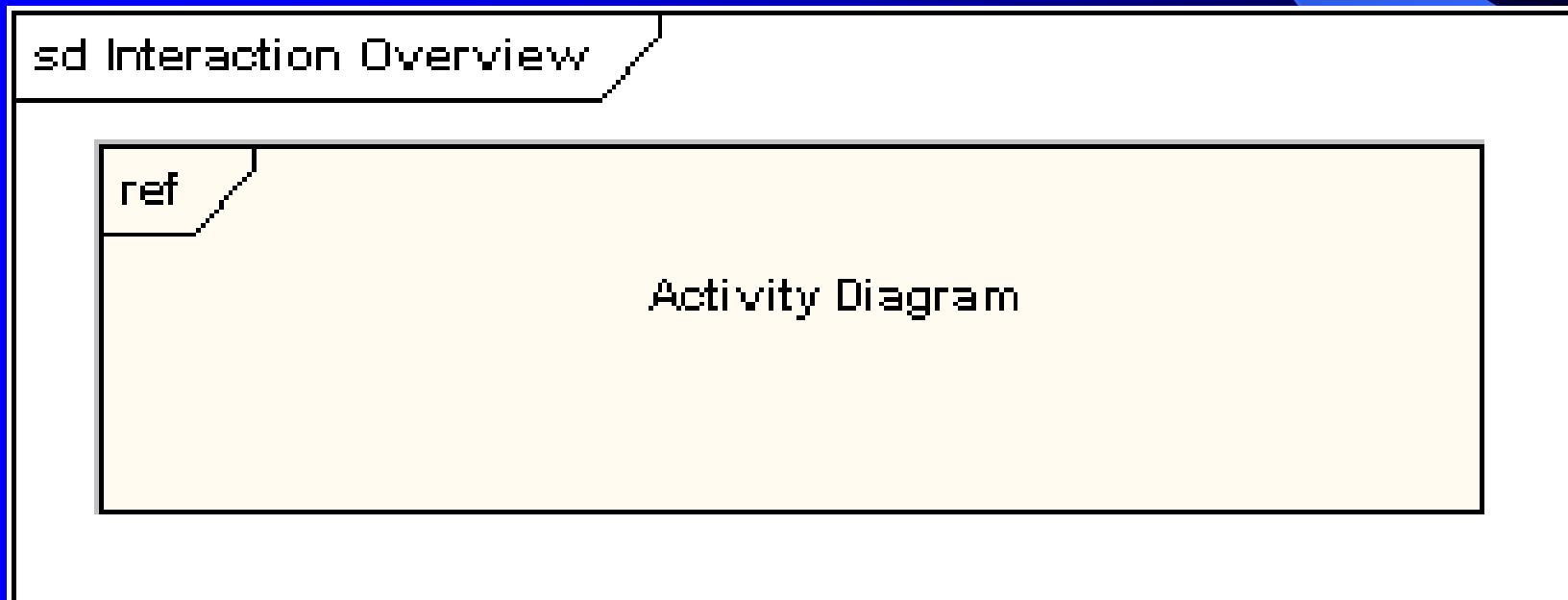
并且，交互概览图介绍了两种新的元素：交互发生和交互元素。

4.5 软件体系结构与UML

12) 交互概览图

交互发生引用现有的交互图。显示为一个引用框，左上角显示 "ref"。被引用的图名显示在框的中央。

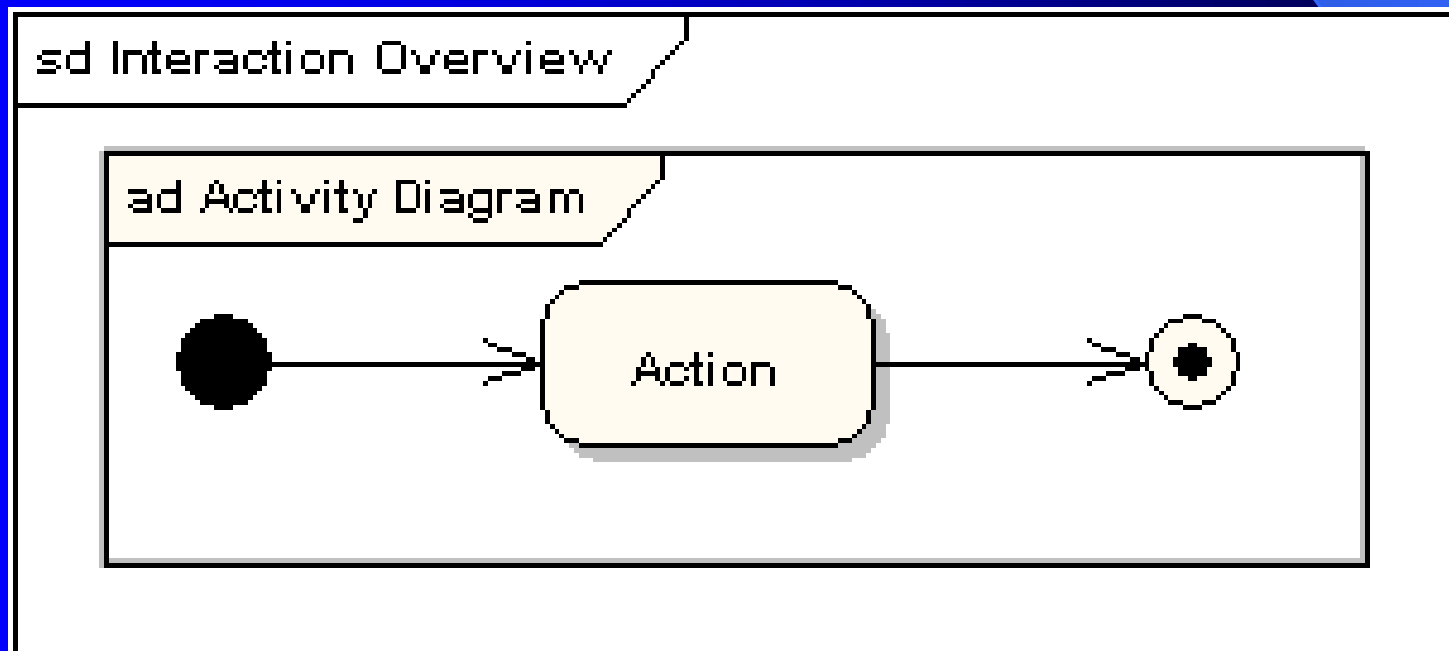
。



4.5 软件体系结构与UML

12) 交互概览图

交互元素与交互发生相似之处在于都是在一个矩形框中显示一个现有的交互图。不同之处在内部显示参考图的内容不同。



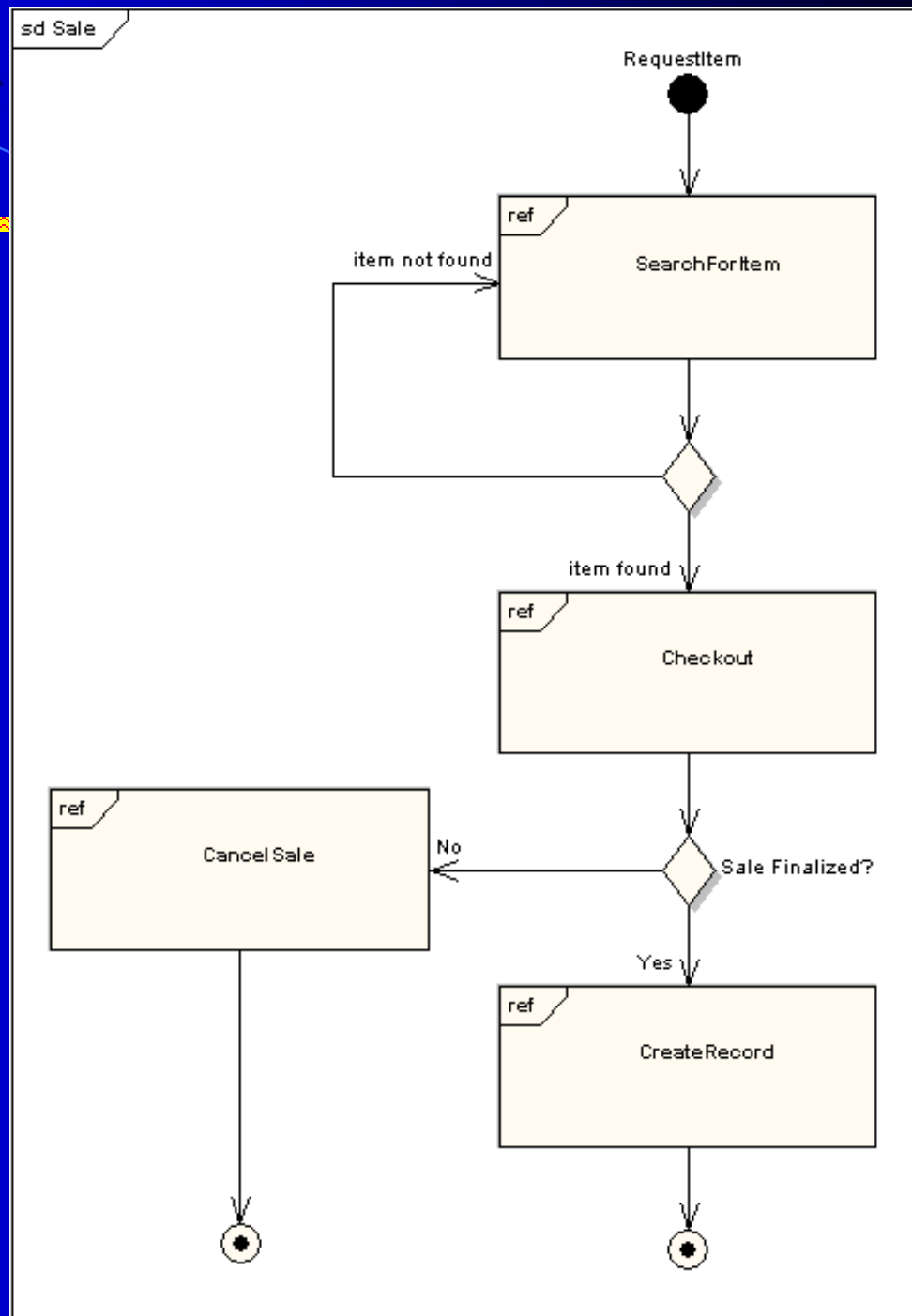
4.5 软件体系结构与UML

12) 交互概览图

所有的活动图控件，都可以相同地被使用于交互概览图，如：分叉，结合，合并等等。它把控制逻辑放入较低一级的图中。下面的例子就说明了一个典型的销售过程。

4.5 软件体系结

12) 交互概览图



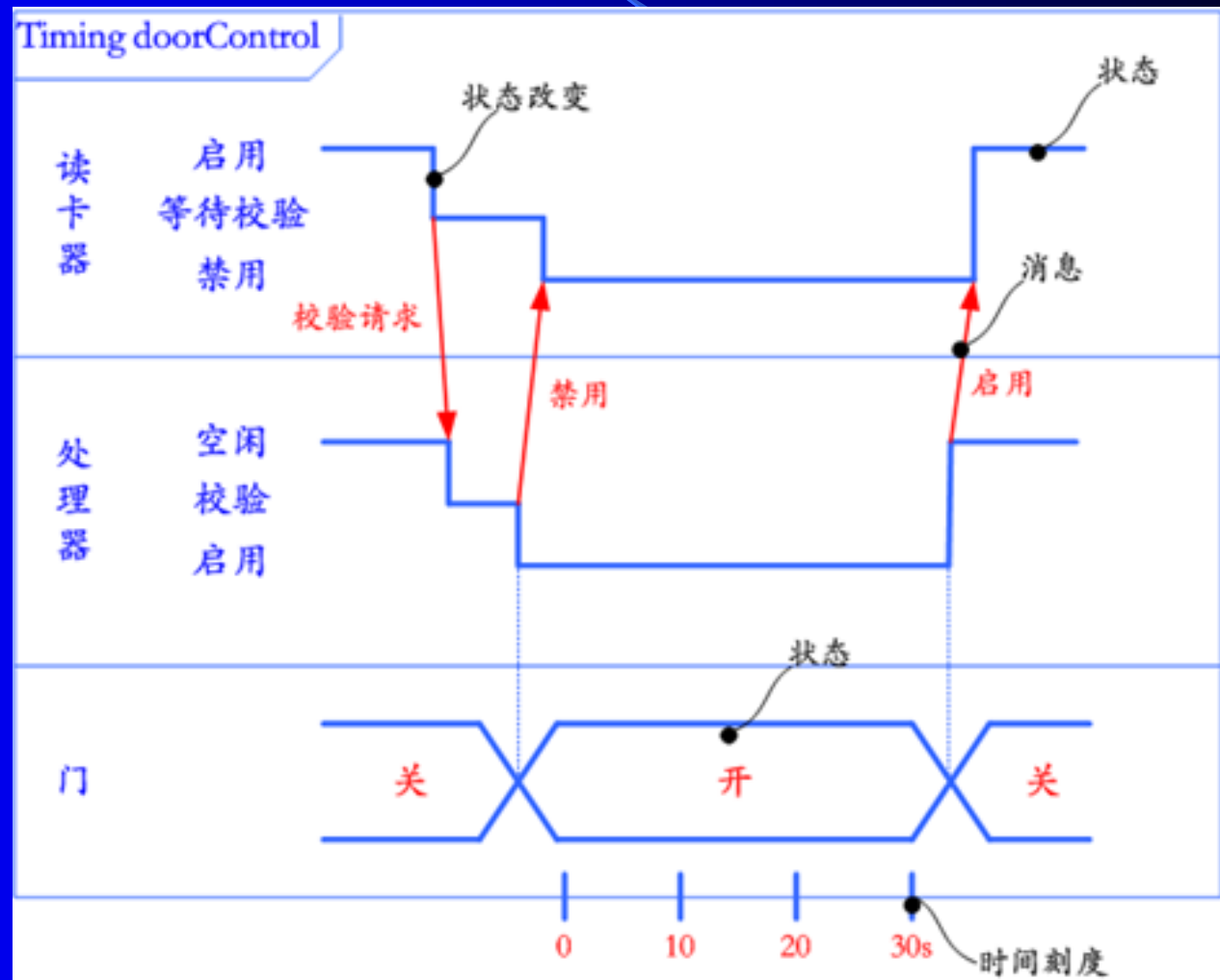
4.5 软件体系结构与UML

13) 时序图

与顺序图相比，坐标轴交换了位置，改为从左到右来表示时间的推移；用生命线的“凹下凸起”来表示状态的变化，每个水平位置代表一种不同的状态，状态的顺序可以有意义、也可以没有意义；生命线可以跟在一根线后面，在这根线上显示些不同的状态值；可显示一个度量时间值的标尺，用刻度表示时间间隔。

4.5 软件体系结构与UML

13) 时序图



4.5 软件体系结构与UML

(3) UML扩展

UML的创始人意识到总会存在这样的情况：初始的UML不足以获取一个特定领域或构架的相关语义。为了解决这个问题，他们确定了一种正式扩展机制，允许实践者扩展UML的语义。

4.5 软件体系结构与UML

(3) UML扩展

在UML中，扩充新元素可以通过附加新的语义到其它模型元素之上得到，将原有的建模元素特化成一种语义较特殊的新的变种，或者表示出它们的某些细节。这些新元素可以起到对已有的表示法进行扩充或细化的作用。

UML提供了三种扩充机制：构造型(stereotypes)、标记值(tagged value)和约束(constraint)。

4.5 软件体系结构与UML

(3) UML扩展

1)构造型是UML提供的最重要的扩充机制，用于在预定义的模型元素(称为基元素)基础上构造新的模型元素。它提供了一种在模型层中加入新的建模元素的方法。特殊用途的语义可以取决于用户的习惯，或特定的约束语言，或编程语言的解释。这样定制出来的建模元素可看作是原先建模元素的一个子类，它在属性、关系等方面与原先的元素形式相同，只是添加了新的语义，用途更为具体。

4.5 软件体系结构与UML

(3) UML扩展

构造型是在一个已定义的模型元素的基础上构造的一种新的模型元素。构造型的信息内容与已存在的基本模型元素相同，但是含义和使用不同。例如，商业建模领域的建模者希望将商业对象和商业过程作为特殊的建模元素区别开来，这些元素的使用在特定的开发过程中是不同的。它们可以被看作是特殊的类，但是他们与其他元素的关系上和它们的使用上有特殊的约束。

4.5 软件体系结构与UML

(3) UML扩展

构造型建立在已存在的模型元素基础上，构造型可以用标记值来存储不被基本模型所支持的附加属性，但是构造型元素的信息内容与已存在的模型元素基本相同，这样便可以允许工具以相同的方式存储和使用新元素和已存在的元素。构造型元素可以有它自己的区别符号，并且很容易被工具支持。例如，一个“商业组织”可以有一个看起来像一组人的图标。构造型也可以有适用于它的约束。例如，一个“商业组织”可能只能与另一个“商业组织”，而不能与任何其他类联合。不是所有的约束都能被多用途工具自动地确定，但是他们可以被手动执行或者被理解构造型地加入工具确定。

4.5 软件体系结构与UML

(3) UML扩展

构造型用双尖括号内的文字字符串表示，它可以放在表示基本模型元素的符号的里面或者旁边。建模者也可以为特殊的构造型创建一个符号，这个符号替代了原来的基本模型元素的符号。

定义stereotype必须满足以下规则：

- a) stereotype名不能与其基类重名；
- b) stereotype名不能与它所继承的stereotype重名；
- c) stereotype名不能与元类名空间冲突；
- d) stereotype所定义的tag名不能与其基类元素的元属性命名空间冲突，也不能与它所继承的tag名冲突。

4.5 软件体系结构与UML

2) 标记值

标记值是一对字符串：一个标记字符串和一个值字符串，值字符串存储着有关元素的一些信息。标记值可以与任何独立元素相关，包括模型元素和表达元素。标记是建模者想要记录的一些特性的名字，值是给定元素的特性值。例如，标记可以是employer，而值是对元素负责的人的名字，如Kramer。

4.5 软件体系结构与UML

2) 标记值

标记值可以用来存储元素的任意信息，对于存储项目管理信息尤为有用，如元素的创建日期、开发状态、截至日期和测试状态。除了内部元模型属性名称外，任何字符串可以作为标记值(这是因为标记和属性在一起会被认为是一个元素的属性并且可以被工具一起访问)，一些标记名称已经被预定义了。

4.5 软件体系结构与UML

2) 标记值

标记值还提供了一种方式将独立于实现的附加信息与元素连续起来。例如，代码生成器需要有关代码的附加信息以从模型中生成代码。通常，有几种方式可以用来正确的实现模型，标记可以用来告诉代码生成器使用那种实现方式。

标记值也可以用来存储有关构造型模型元素的信息。

4.5 软件体系结构与UML

2) 标记值

标记值用字符串表示，字符串有标记名称、等号和值。它们被规则的放置在大括弧内。在图标中标记值经常被省略，只显示在下拉表格中。

4.5 软件体系结构与UML

3)约束是模型元素之间的一种语义关系，是对建模元素语义上的限制。它说明了某种条件和某些必须保持为真的命题，否则系统所描述的模型无效，同时UML的语义无法解释其结果。其表示法是在大括号{}之间用一种工具能识别的语言(如UML提供的OCL)写出表示条件的正文串。约束可以附加在表元素、依赖关系或注释上。

4.5 软件体系结构与UML

约束是用文字表达式表示的语义限制。每个表达式有一种隐含的解释语义，这种语言可以是正式的数学符号，如集合论表示法；或者是一种基于计算机的约束语言，如OCL；或者是一种编程语言，如C++；或者是伪代码或非正式的自然语言。当然，如果这种语言是非正式的，那么它的解释也是非正式的，并且要有人来解释。即使约束由一种正式语言来表示，也不意味着它自动成为有效约束，必须有相应的解释工具来翻译约束的语法和语义以支持真正意义上的约束。

4.5 软件体系结构与UML

约束可以表示不能用UML表示法来表示的约束和关系。当描述全局条件或影响许多元素的条件时约束特别有用。

4.5 软件体系结构与UML

3、使用UML描述软件体系结构

UML是一种可视化的面向对象的建模语言，虽然与ADL有着相似的软件建模哲学，但二者有着不同的假设：UML倾向的用途并不是用于体系结构建模；UML中的一些假设在体系结构描述语言中则不存在。所以，UML并不是一种体系结构描述语言。尽管如此，UML仍具有较强的体系结构建模特性，在当前的体系结构描述实践中，若干研究者都提出了用UML描述体系结构的思路。

4.5 软件体系结构与UML

3、使用UML描述软件体系结构

UML作为一种定义良好、易于表达、功能强大且普遍适用的建模语言，可以用于软件体系结构的建模。软件设计人员为了降低软件体系结构建模的复杂度，从多个不同的视图来描述软件体系结构，通过某种视图来描述软件体系结构的某个侧面及不同的特点，然后将多个视图结合在一起则可较全面地反映软件体系结构的内容和特征。

4.5 软件体系结构与UML

3、使用UML描述软件体系结构

另外，由于软件体系结构所支持的构件及构件之间的语义非常丰富，仅用图和自然语言的方法很难描述清楚。UML的一个重要特点是引入了形式化定义(对象约束语言)，这有利于描述软件体系结构，同时，UML又具有很好的扩充机制，众多的工具支持，与具体程序设计语言和开发过程无关，因此，可选择UML作为描述软件体系结构的基础。

4.5 软件体系结构与UML

3、使用UML描述软件体系结构

UML的图本身就隐含了体系结构的元素。例如：用例图从概念上描述了系统的逻辑功能，类图反映了体系结构中的静态关系；顺序图在一定程度上反映了系统的同步与并行的信息；活动图表现了一定的并发行为的信息；构件图反映了系统的逻辑结构；部署图描述了物理资源的分布情况。

4.5 软件体系结构与UML

3、使用UML描述软件体系结构

UML的元素与体系结构的元素很相似：

- 1)UML结构元素中的用例、类、构件、节点和组织元素中的包、子系统相当于体系结构中的组件；
- 2)UML的关系支持体系结构的连接件；
- 3)UML建模元素中的接口支持体系结构的接口；
- 4)UML中的规则支持体系结构的约束；
- 5)体系结构的配置可以由UML的包图、构件图和配置图描述；
- 6)UML预定义及用户自己扩展的构造型，例如精化、复制等能够较好地表达体系结构的行为。

4.5 软件体系结构与UML

3、使用UML描述软件体系结构

表1 UML 的体系结构建模特性分析

体系结构元素	相应的 UML 元素
构件	类、构件、节点、用例、包、子系统
连接件	关系,包括泛化、关联、依赖
接口	接口
约束	规则,包括 OCL
配置	构件图、包图、配置图
其它属性	扩展机制

4.5 软件体系结构与UML

用UML描述体系结构的两种思路:

1)直接的UML体系结构描述: 根据某种体系结构描述模型, 如4+ 1模型, 然后以UML 中类似的模型去表示与实现。

2)间接的UML体系结构描述: 根据一种ADL的元素和描述思路, 采用UML模型元素来表示这些元素和描述, 这种体系结构的UML描述是间接的, 要考虑ADL向UML的转换。

4.5 软件体系结构与UML

三种UML实现体系结构建模方法及其比较：

1)无扩展方法(直接的UML)：按照原来的方式使用UML，并且直接比较UML和某个ADL，说明体系结构与设计之间的关系。因而在体系结构建模和软件建模之间存在一种映射关系，在UML描述的体系结构到以UML描述的设计之间实现转换。该方法描述的体系结构是可理解的，可以有多个标准的工具进行操作，但是可能违背体系结构约束；

4.5 软件体系结构与UML

“4+1”模型从逻辑、开发、过程、物理和场景5个不同的视角来描述软件体系结构的全部内容。

场景可以采用UML中的用例图来描述。

概念视图可以采用UML用例图来表示，从所有参与者的角度出发，通过用例来描述他们对系统概念的不同理解，每一个用例名相当于一个概念功能的名称。

过程视图用UML中的活动图来描述。

构件视图用UML中的构件图来表示。在构件视图中，组件就是程序模块，程序模块之间的关系是连接件。

物理视图用UML中的部署图来表示。

4.5 软件体系结构与UML

2)内嵌扩展方法(约束的UML): 应用UML对元类所支持的扩展机制, 选择在语义上与ADL构造子相近的元类, 定义一个构造型, 并将其应用到元类实例上, 但类的语义应该遵循ADL的约束, 允许一致性检查。该方法保证了体系结构的约束, 但需要完全的风格规格说明, 需要OCL兼容的工具;

4.5 软件体系结构与UML

3)定制扩展方法(扩充的UML): 扩展UML直接支持体系结构问题, 即直接增加体系结构描述的设计元素。该方法对体系结构提供原本的支持, 但是需要后向兼容工具支持, 会导致UML版本的不兼容。

4.6 可扩展标记语言

可扩展标记语言（eXtensible Markup Language，简称：XML），是一种标记语言。

标记指计算机所能理解的信息符号，通过此种标记，计算机之间可以处理包含各种信息的文章等。

如何定义这些标记，既可以选择国际通用的标记语言，比如HTML，也可以使用像XML这样由相关人士自由决定的标记语言，这就是语言的可扩展性。

XML是从标准通用标记语言（SGML）中简化修改出来的。它主要用到的有可扩展标记语言、可扩展样式语言（XSL）、XBRL和XPath等。

4.6 可扩展标记语言

1、发展

XML是从1995年开始有其雏形，并向W3C（万维网联盟）提案，而在1998年2月发布为W3C的标准（XML1.0）。XML的前身是SGML（The Standard Generalized Markup Language），是自IBM从1960年代就开始发展的GML（Generalized Markup Language）标准化后的名称。

GML的重要概念：

- 1) 文件中能够明确的将标示与内容分开
- 2) 所有文件的标示使用方法均一致

4.6 可扩展标记语言

1、发展

1978年，ANSI将GML加以整理规范，发布成为SGML，1986年起为ISO所采用（ISO 8879），并且被广泛地运用在各种大型的文件计划中，但是SGML是一种非常严谨的文件描述法，导致过于庞大复杂（标准手册就有500多页），难以理解和学习，进而影响其推广与应用。

4.6 可扩展标记语言

1、发展

同时W3C也发现到HTML的问题：

- 1) 不能解决所有解释数据的问题 – 例如影音档或化学公式、音乐符号等其他形态的内容。
- 2) 性能问题 - 需要下载整份文件，才能开始对文件做搜索。
- 3) 扩充性、弹性、易读性均不佳。

4.6 可扩展标记语言

1、发展

为了解决以上问题，专家们使用SGML精简制作，并依照HTML的发展经验，产生出一套使用上规则严谨，但是简单的描述数据语言：XML。XML是在一个这样的背景下诞生的--为了有一个更中立的方式，让消费端自行决定要如何消化、体现从服务端所提供的信息。

XML被广泛用来作为跨平台之间交互数据的形式，主要针对数据的内容，通过不同的格式化描述手段（XSLT，CSS等）可以完成最终的形式表达（生成对应的HTML，PDF或者其他文件格式）。

4.6 可扩展标记语言

2、用途

XML设计用来传送及携带数据信息，不用来表现或展示数据，HTML语言则用来表现数据，所以XML用途的焦点是它说明数据是什么，以及携带数据信息。

1) 丰富文件 (Rich Documents) - 自定文件描述并使其更丰富

属于文件为主的XML技术应用

标记是用来定义一份资料应该如何呈现

2) 元数据 (Metadata) - 描述其它文件或网络资讯

属于资料为主的XML技术应用

标记是用来说明一份资料的意义

3) 配置文档 (Configuration Files) - 描述软件设置的参数

4.6 可扩展标记语言

3、DTD与Schema

XML 文件的文件型别定义（Document Type Definition）可以看成是一个或者多个 XML 文件的模板，在这里可以定义 XML 文件中的元素、元素的属性、元素的排列方式、元素包含的内容等等。

4.6 可扩展标记语言

3、DTD与Schema

DTD (Document Type Definition) 概念缘于 SGML, 每一份 SGML 文件, 均应有相对应的 DTD。对 XML 文件而言, DTD 并非特别需要, well-formed XML 就不需要有 DTD。DTD 有四个组成如下:

- 1) 元素 (Elements)
- 2) 属性 (Attribute)
- 3) 实体 (Entities)
- 4) 注释 (Comments)

由于 DTD 限制较多, 使用时较不方便, 近来已渐被 XML Schema 所取代。

4.6 可扩展标记语言

3、DTD与Schema

XML Schema如W3C建议，发布于2001年5月，是许多XML纲要语言中的一支。它是首先与XML本身分离的纲要语言，故取得W3C的推荐地位。

XML Schema语言也被称为XML Schema Definition (XSD)，用于描述了XML文档的结构。

XML Schema如同DTD一样是负责定义和描述XML文档的结构和内容模式。它可以定义XML文档中存在哪些元素和元素之间的关系，并且可以定义元素和属性的数据类型。

XML Schema本身是一个XML文档，它符合XML语法结构。可以用通用的XML解析器解析它。

4.6 可扩展标记语言

3、DTD与Schema

DTD有着不少缺陷：

- 1) DTD是基于正则表达式的，描述能力有限；
- 2) DTD没有数据类型的支持，在大多数应用环境下能力不足；
- 3) DTD的约束定义能力不足，无法对XML实例文档作出更细致的语义限制；
- 4) DTD的结构不够结构化，重用的代价相对较高；
- 5) DTD并非使用XML作为描述手段，而DTD的构建和访问并没有标准的编程接口，无法使用标准的编程方式进行DTD维护。

4.6 可扩展标记语言

3、DTD与Schema

而XML Schema正是针对这些DTD的缺点而设计的，XML Schema的优点：

- 1) XML Schema基于XML，没有专门的语法
- 2) XML Schema可以象其他XML文件一样解析和处理
- 3) XML Schema支持一系列的数据类型(int、float、Boolean、date等)
- 4) XML Schema提供可扩充的数据模型。
- 5) XML Schema支持综合命名空间
- 6) XML Schema支持属性组。

4.6 可扩展标记语言

4、CSS和XSL

层叠样式表（Cascading Style Sheets，简写CSS），又称串样式列表，由W3C定义和维护的标准，一种用来为结构化文档（如HTML文档或XML应用）添加样式（字体、间距和颜色等）的计算机语言。

CSS的语法很简单，它使用一组英语词来表示不同的样式和特征。

4.6 可扩展标记语言

4、CSS和XSL

一个式样表由一组规则组成。每个规则由一个“选择器”（selector）和一个定义部分组成。每个定义部分包含一组由半角分号（;）分离的定义。这组定义放在一对大括号（{ }）之间。每个定义由一个特性，一个半角冒号（:）和一个值组成。

选择器（Selector）：通常为档中的元素（element），如HTML中的<body>，<p>，等标签，多个选择器可以半角逗号（,）隔开。

属性（property）：CSS1、CSS2、CSS3规定了许多属性，旨在控制选择器的样式。

值（value）：指属性接受的设置值，可由各种关键字（keyword）组成，多个关键字时大都以空格隔开。

4.6 可扩展标记语言

4、CSS和XSL

XSL是指可扩展样式表语言 (EXtensible Stylesheet Language), 是一种用于以可读格式呈现 XML 数据的语言。

XSL实际上包含两个部分: XSLT和XSL-FO。

可扩展样式表转换语言 (Extensible Stylesheet Language Transformations, 简称XSLT) 是一种对 XML文档进行转化的语言, XSLT中的T代表英语中的“转换” (transformation)。

4.6 可扩展标记语言

4、CSS和XSL

XSLT是把XML文档转化为另一文档的转换语言，即将源文档的所有数据或者部分数据，利用XPath进行选择，生成另外的XML文档或者其他可直接显示或打印的文件格式（例如HTML文件、RTF文件或者TeX文件）。XSLT语言是声明性的语言，即XSLT程序本身只是包含了一些转换规则的文档。而这些规则可以被递归地应用到转换过程中。XSLT处理程序会首先确定使用XSLT中的哪些规则，然后根据优先级作出相应的转换操作。

XSLT本身也是一份XML文档，所以它也必须遵守严格的XML规范。

4.6 可扩展标记语言

5、Xpath, Xpointer与Xlink

XML链接与HTML完全不同，它没有专门的链接元素，需要通过指定元素属性来表示链接，只要元素包含xlink:type属性，且取值为"simple"或"extended"，该元素就是链接元素，其中xlink是代表XLink命名空间的前缀，根据xlink:type属性的取值，可以将XML链接划分为简单XML链接和扩展XML链接。简单XML链接的xlink:type固定取值为"simple"，扩展XML链接的xlink:type固定取值为"extended"。

4.6 可扩展标记语言

5、Xpath, Xpointer与Xlink

简单XML链接与HTML链接非常相似，它在链接元素和目标资源间建立链接。需要强调的是，如果XML文件具有文件类型定义DTD，XLink的全局属性必须在DTD中定义。不过，并不是所有的XLink全局属性都必须在DTD中加以声明，可以根据需要进行裁剪，只声明使用到的属性即可。而且，将一个元素声明为链接元素并没有增加对元素的属性和内容的限制，只要元素实例符合DTD声明，仍然可以包括任意属性和内容。

4.6 可扩展标记语言

5、Xpath, Xpointer与Xlink

XPath 是一门在 XML 文档中查找信息的语言。XPath 用于在 XML 文档中通过元素和属性进行导航。

XPointer是在可扩展标志语言（XML）文件中定位数据的一种语言，其定位是根据数据在文件中位置、字符内容以及属性值等特性进行的。

4.6 可扩展标记语言

6、XML名字空间

XML命名空间（XML namespace）用于在一个XML文档中提供名字唯一的元素和属性。XML命名空间在W3C推荐规范《Namespaces in XML》中定义。XML命名空间于1999年1月14日成为W3C的推荐规范。

XML名字空间提供了一种避免元素名冲突的方法

。

4.6 可扩展标记语言

6、XML名字空间

因为XML文档中使用的元素不是固定的，那么两个不同的XML文档使用同一个名字来描述不同类型的元素的情况就可能发生。而这种情况又往往会导致命名冲突。如果每一个词汇表指派一个命名空间，那么相同名字的元素或属性之间的名称冲突就可以解决。

4.6 可扩展标记语言

7、XML查询语句

XML查询语言(XQL) 是用于定位和过滤XML文档中元素和文本的符号。它是XSL模式语法的自然扩展，为指向特定的元素或查找具有指定特征的节点提供了简明的可以理解的符号。XQL是最早由Microsoft、Texcel等公司提出的一种XML查询语言。

4.6 可扩展标记语言

7、XML查询语句

前面我们已经提到，XSL模式语言提供了一种描述一类将需要处理的结点的好方法，实际上是通过XPath来实现的，当然XSL是说明性的，而非过程性的。

但XSL也有许多不足之处，如不支持表达式，不能在结点间进行等值连接，对多个分散的XSL文档没有一个形式化的查询机制，没有支持聚集操作等。

XQL则在XSL基础上提供了筛选操作、布尔操作，对结点集进行索引，并为查询、定位等提供了单一的语法形式。因而在一定意义上可将XQL看作XSL的超集。

4.6 可扩展标记语言

7、XML查询语句

XQL主要针对解决以下四个问题域而提出：

- (1)在单个XML文档中进行查询。如在XML浏览器和编辑器中对所处理的XML文档进行查询。另外，Script也能利用这种性质给出一个非过程性的存取文档数据和结构的方法。
- (2)在XML文档集中进行查询。如在XML文档仓储(Repository)中进行查询。

4.6 可扩展标记语言

7、XML查询语句

XQL主要针对解决以下四个问题域而提出：

- (3)能在XML文档间对结点进行定位。在HTML文档中，常常用HyperLink来定位其它文档，而在已有的XML中，链接形式更多样化，有TEL Links, HyTime Links, XML Linking，这些都允许链接有更大的灵活性。XQL则主要是想通过给出一已知位置的相对路径，或通过一绝对路径来在文档中定位任一结点。
- (4)以字符串语法形式表达，使其能在URL中应用在XSL模板中及其它地方应用。

4.6 可扩展标记语言

7、XML查询语句

XML-QL是在查询语言UnQL和StruQL基础上设计的，它能对XML文档进行查询、构造、转换和集成。它集中了查询语言技术和XML语法格式，通过说明路径表达式和模式的方式，给出XML数据的提取条件（WHERE子句）。

同时，XML-QL中可以给出构造查询输出的XML数据的模板，其输出结果仍为XML文档（CONSTRUCT子句）。

当操作对象限定为关系类数据时，XML-QL与关系演算或关系代数有同样的表达能力，即XML-QL是关系完备的。

4.6 可扩展标记语言

7、XML查询语句

XML-QL有类似select-from-where的结构(where-in-construct)，与SQL很相似。但XML-QL有一些很重要的区别于基于结构化数据查询语言的特点。其WHERE子句由模式和关系表达式组成，这意味着被选出的数据项要满足两个条件：

- 1) 数据项的类型（或Schema）和值必须与指定的模式匹配；
- 2) 数据项的值要满足关系表达式。

在查询条件中加入模式匹配是XML-QL与半结构化查询语言和结构化查询语言最大的不同之处。

4.6 可扩展标记语言

7、XML查询语句

XQuery = XML Query，是W3C所制定的一套标准，用来从类XML文档中提取信息，类XML文档可以理解成一切符合XML数据模型和接口的实体，他们可能是文件或RDBMS。

XQuery有如下特点：

- 1) XQuery是查询XML的语言
- 2) XQuery类似RDBMS的SQL
- 3) XQuery建立在XPATH的基础之上
- 4) XQuery已经被现在主流的RDBMS所支持，如Oracle, DB2, SQLServer

4.6 可扩展标记语言

8、资源描述框架

资源描述框架（Resource Description Framework，缩写 RDF），是万维网联盟（W3C）提出的一组标记语言的技术标准，以便更为丰富地描述和表达网络资源的内容与结构。

RDF使用XML语法和RDF Schema（RDFS）来将元数据描述成为数据模型。

4.6 可扩展标记语言

8、资源描述框架

一个RDF文件包含多个资源描述，而一个资源描述是由多个语句构成，一个语句是由资源、属性类型、属性值构成的三元组，表示资源具有的一个属性。

通过RDF，人们可以使用自己的词汇表描述任何资源，但人们更乐意将它用于描述Web站点和页面，由于使用的是结构化的XML数据，搜索引擎可以理解元数据的精确含义，使得搜索变得更为智能和准确，完全可以避免当前搜索引擎经常返回无关数据的情况。

4.6 可扩展标记语言

9、DOM、SAX和XML解析器

XML DOM (Document Object Model) 定义了所有 XML 元素的对象和属性, 以及访问它们的方法 (接口), 换句话说, XML DOM 是用于获取、更改、添加或删除 XML 元素的标准。

DOM 是处理 XML 数据的传统方法。使用 DOM 时, 数据以树状结构的形式被加载到内存中。

4.6 可扩展标记语言

9、DOM、SAX和XML解析器

DOM 以及广义的基于树的处理具有几个优点。

由于树在内存中是持久的，因此可以修改它以便应用程序能对数据和结构作出更改；它还可以在任何时候在树中上下导航，而不是像 SAX 那样是一次性的处理；DOM 使用起来也要简单得多。

不足之处：

在内存中构造这样的树涉及大量的开销，大型文件完全占用系统内存容量的情况并不鲜见；此外，创建一棵 DOM 树可能是一个缓慢的过程。

4.6 可扩展标记语言

9、DOM、SAX和XML解析器

相比于DOM，SAX（Simple API for XML）是一种速度更快，更有效的方法。它逐行扫描文档，一边扫描一边解析。

SAX处理涉及以下步骤：

1. 创建一个事件处理程序。
2. 创建 SAX 解析器。
3. 向解析器分配事件处理程序。
4. 解析文档，同时向事件处理程序发送每个事件。

4.6 可扩展标记语言

9、DOM、SAX和XML解析器

这种处理的优点非常类似于流媒体的优点。分析能够立即开始，而不是等待所有的数据被处理。而且，由于应用程序只是在读取数据时检查数据，因此不需要将数据存储在内存中。这对于大型文档来说是个巨大的优点。事实上，应用程序甚至不必解析整个文档；它可以在某个条件得到满足时停止解析。

4.7 基于XML的软件体系结构描述语言

1、XADL 2.0

xADL是美国加州大学Irvine分校的软件研究所提出的一种基于XML的ADL。

他们利用XML和XML schemas提供的扩展机制，将ADL特征封装在模块中，并可以将这些模块组合起来形成了xADL。

xADL2.0 定义了一个通用的，可重用的ADL模块集，其本身作为一种ADL，同时还可被扩展用来支持新的应用和新的领域。

在xADL2.0 中有若干的XML schemas，它们分别支持描述运行时体系结构、设计时体系结构以及实现映射等。

4.7 基于XML的软件体系结构描述语言

1、XADL 2.0

xADL2.0以xArch为基础的基于XML的ADL。除了xArch的核心元素，xADL2.0还提供了对系统运行时刻和设计时刻元素的建模支持，类似版本、选项和变量等更高级的配置管理观念，以及对产品家族体系结构的建模支持。并且利用XML的可扩展性简化了设计新的ADL和开发相应工具的过程。

4.7 基于XML的软件体系结构描述语言

1、XADL 2.0

xADL2.0通用的特征体现在：分离了软件系统的运行时和设计时的模型；实现映射能将体系结构的ADL规约映射到可执行代码；具有对体系结构演化和产品线体系结构的各方面进行建模的能力。

4.7 基于XML的软件体系结构描述语言

2、XBA

作为一种可扩展的ADL，XBA包括了构架描述中的三种基本抽象元素：构件、连接器和配置。

4.7 基于XML的软件体系结构描述语言

2、XBA

图1给出了一个简单的客户机—服务器结构，Client和Server是两个构件，Rpc为连接器，构件和连接器的实例构成了这个C-S 系统的Configuration。

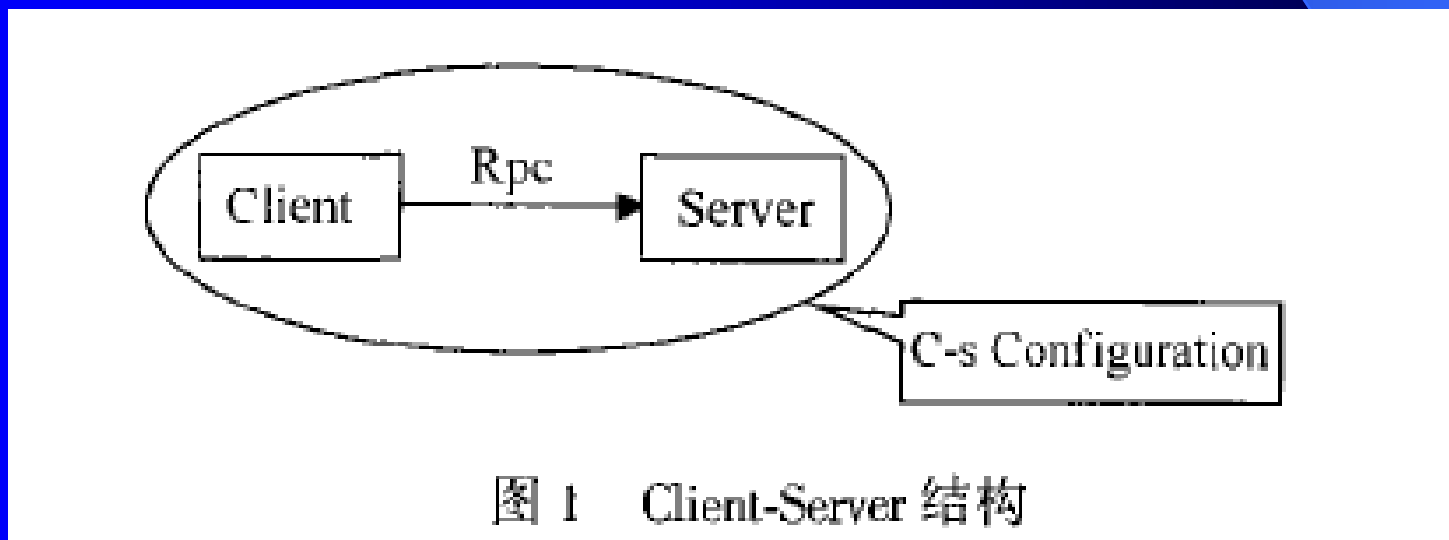


图 1 Client-Server 结构

4.7 基于XML的软件体系结构描述语言

2、XBA

1) 构件描述

一个构件描述了一个局部的、独立的计算。在XBA里，对一个构件的描述有两个重要的部分：接口(interface)和计算(computation)。

一个接口由一组端口(port)组成，每一个端口代表这个构件可能参与的交互。

计算部分描述了这个构件实际所做的动作，计算实现了端口所描述的交互并且显示了它们是怎样被连接在一起构成一个一致的整体。

4.7 基于XML的软件体系结构描述语言

2、XBA

1) 构件描述

端口定义了一个构件的接口，它指出了构件的两个方面的特征。首先，它指出了构件对外界提供的服务；其次，构件对它所交互的系统的要求。在XBA中，并不显式区分服务端口和请求端口，如果要显式说明这两种端口，可以通过对Schema 扩展，加入一个属性或元素来说明端口的类型。

4.7 基于XML的软件体系结构描述语言

2、XBA

```
<complexType name="componentType">
  <Sequence>
    <element name="Port" type="portType" minOccurs="0"
      maxOccurs="unbounded"/>
    <element name="Computation" type="computationType" minOccurs="0"/>
  </Sequence>
  <attribute name="Name" type="string"/>
</complexType>
<complexType name="portType">
  <element name="Description" type="string" minOccurs="0"/>
  <attribute name="Name" type="string"/>
</complexType>
<complexType name="computationType">
  <element name="Description" type="string" minOccurs="0"/>
  <attribute name="Name" type="string"/>
</complexType>
```


4.7 基于XML的软件体系结构描述语言

2、XBA

2) 连接器描述

一个连接器代表了一组构件间的交互。使用连接器的一个重要的好处是它通过结构化一个构件与系统其它部分交互的方式，增加了构件的独立性。

一个连接器实际上提供了构件必须满足的一系列要求和一个信息隐藏的边界，这个边界阐明了构件对外部环境的要求。

构件说明只需指明构件将要做什么，而连接器说明描述了在一个实际的语境中这个构件怎样与其它部分合作。

4.7 基于XML的软件体系结构描述语言

2、XBA

一个连接器由一组角色(Roe)和胶合(Glue)组成。每一个角色说明了一个交互中的一个参与者的行为。

在连接器里Glue描述了参与者怎样一起合作来构成一个交互。Glue说明了各个构件的计算怎样合起来组成一个规模更大的计算，就构件中的计算部分一样，Glue表示了连接器的动作。

4.7 基于XML的软件体系结构描述语言

2、XBA

```
<connector name="Rpc">
  <Role name="Source">
    <Description>
      Get request from client
    </Description>
  </Role>
  <Role name="Sink">
    <Description>
      Get request from server
    </Description>
  </Role>
  <Glue>
    Get request from client and pass it to server through some protocol
  </Glue>
</connector>
```

4.7 基于XML的软件体系结构描述语言

3)配置描述

为了描述整个系统构架，构件和连接器必须合并为一个configuration。一个configuration就是通过连接器连接起来的一组构件实例(Instance)。

4.7 基于XML的软件体系结构描述语言

3) 配置描述

因为在一个系统中对一个给定的构件或连接器可能会使用多次，所以我们把前面在介绍构件和连接器时所举的例子称为是构件或连接器的类型，也就是说，它们代表了构件或连接器的属性，而不是使用中的实例。为了区分一个configuration中出现的每一个构件和连接器的不同实例，XBA要求每一个实例都明确的命名，并且这个命名应唯一。

4.7 基于XML的软件体系结构描述语言

3) 配置描述

Attachments 通过描述哪些构件参与哪些交互定义了一个configuration的布局或称为拓扑，这是通过把构件的端口和连接器的角色联系在一起完成的。构件实现了计算，而端口说明了一个特别的交互。端口与一个角色联系在一起，那个角色说明了为了成为连接器所描述的交互的合法参与者，那个端口所必须遵守的规则。如果每一个构件的端口都遵守角色所描述的规则，那么连接器的Glue 定义了各个构件的计算怎样合并为一个单一的更大规模的计算。

4.7 基于XML的软件体系结构描述语言

3) 配置描述

```
<configuration name="Client-Server">
  <component name="Client">
    <component name="Server">
      <component name="Rpc">
        <Instance>
          <ComponentInstance>
            <ComponentName>MyClient</ComponentName>
          </ComponentInstance>
        </ComponentInstance>
      </ComponentInstance>
    </ComponentInstance>
  </ComponentInstance>
  <!-- ect -->
  <ConnectInstance>
    <ComponentName>MyRpc</ComponentName>
  </ComponentName>
  <ComponentType>Rpc</ComponentType>
</Instance>
  <Attachments>
    <Attachment>
      <From>Myclient.request</From>
      <To>MyRpc.Source</To>
    </Attachment>
    <!-- ect -->
  </Attachments>
</configuration>
```

4.7 基于XML的软件体系结构描述语言

4) 层次结构的描述

XBA支持层次结构描述。一个构件可以由一个嵌套的子系统构架构成。这样，这个构件就相当于这个子系统构架的包装器，它的内部是一个完整的构架，而在外界看来，它相当于一个构件。一个子系统构架就像前面讲过的那样描述为一个configuration。

例如，在系统的最高层有client和server两个构件，而server是由一个子系统构架实现的，这个子系统构架由ApplicationServer和DatabaseServer两个构件组成。这样我们必须修改上述component的XML Schema定义，应该增加一个元素subconfiguration，类型为configurationType。

4.7 基于XML的软件体系结构描述语言

5) XBA 的可扩展性

通过利用XML Schema的扩展性机制，可以方便的通过XBA的核心XML Schema定义来获得新的可以描述其它ADL的XML schema。

4.7 基于XML的软件体系结构描述语言

6) XBA 的优点

(1)XBA具有开放式的语义结构，继承了XML的基于Schema的可扩展性机制，在使用了适当的扩展机制之后，XBA可以表示多种构架风格。而且可以利用XML Schema的include和import等机制来复用已经定义好的XML schema，实现ADL的模块化定义。

(2)利用XML的链接机制(XLink)，可以让我们实现构架的协作开发。我们可以先把构架的开发分解，由不同的开发者分别开发，然后再利用XML的链接机制把它们集成起来。

(3)易于实现不同ADL开发环境之间的模型共享。可能会存在多种基于XML的ADL开发工具，尽管它们所使用的对ADL的XML描述会有不同，但通过XSLT技术，可以很方便的在对同一构架的不同XML描述之间进行转换。

谢谢大家！

再见！

