软件建模与 UML

Matthew

UML 在软件生命周期的任何阶段都有应用

需求分析 Requirement Analysis
软件设计 Software Design
编码/编程 Coding/Programming
测试/调试 Testing/Debug
运行/维护 Running/Maintenance

软件过程模型 Software Process Model

瀑布模型 Waterfall Model
原始样机模型 Prototype model
增量模型 Incremental Model
螺旋模型 Spiral Model
螺旋模型分为四个阶段：规划、风险分析、工程和评估

关于 UML

1. UML 是一种建模语言，不是一个方法
2. UML 是独立于软件开发过程
3. 在 RUP 和 UML 中讨论了一个架构的五个（4+1）视图
4. 一些 UML 工具支持针对某些面向对象的编程语言的自动代码生成
5. 用于不同情况下使用的机制是不一样的体现在
   (1) 使用何种 UML 图
   (2) 建模粒度
   (3) 使用 UML 的方式
   ……

软件工程设计的方法和技术

结构化的方法 structured method
面向流程——重视数据流
功能分解

面向对象的方法 Object-oriented Method
UML 非常支持关键的面向对象概念
对象、类、继承、消息传递

软件过程 Software Process

流程定义了谁在什么时候做什么，以及如何达到某个目标

合理统一过程 Rational Unified Process(RUP)

像一个在线的指导者；它可以为所有方面和层次的程序开发提供指导方针.

RUP 使用 UML 作为建模软件密集型系统的标准语言.

RUP 的六种最佳实践：

迭代开发 Develop iteratively：

通过连续的改进来增加对问题的理解, 并在多次迭代中逐步增长一个有效的解决方案

管理要求 Manage requirements：

用例图

使用组件 Use components

构件图

模型可视化 Model visually

验证质量 Verify quality

控制更改 Control changes

核心工作流

核心流程工作流 Core Process Workflows

Business Modeling

Requirements

Analysis & Design

Implementation

Test

Deployment

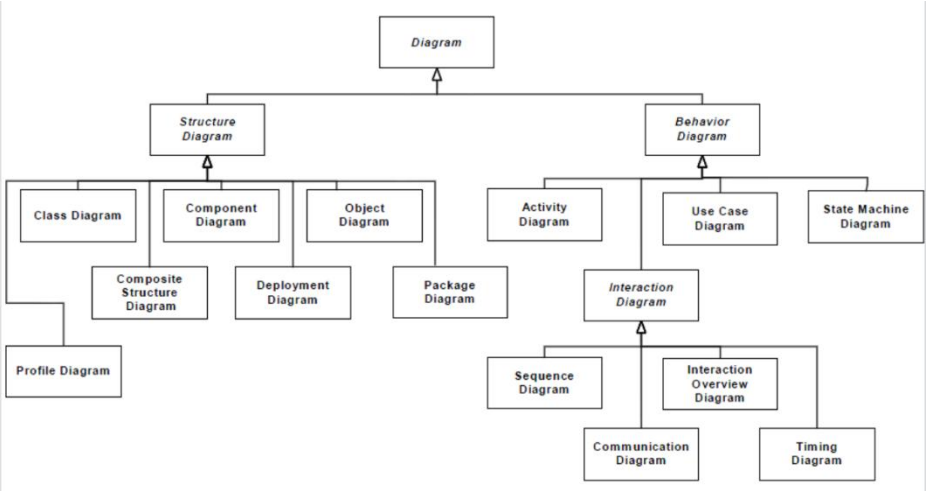
核心支持工作流 Core Supporting Workflows

Configuration & Change Management

Project Management

Environment

统一建模语言 Unified Modeling Language（UML）



行为图 Behavioral Diagrams

活动图、用例图、状态机图、交互图

结构图 Structure Diagram

类图、包图、对象图、部署图、构件图、复合结构图、断面图

UML 特征

独立于开发过程，并可应用于不同的软件开发过程

建模语言！独立于编程语言

UML 构建块 Building Blocks

事物 things

结构性事物：Structural things

行为型事物：Behavioral things

分组事物：Grouping things

注释性事物：Annotational things

关系 relationships

依赖、关联、实现、泛化

图 diagram

结构图、行为图

规则 rule

姓名、范围 Scope、可见性、完整性 integrity、执行 execution

UML 中的常见机制（Mechanisms）

规范 specifications

装饰 Adornments

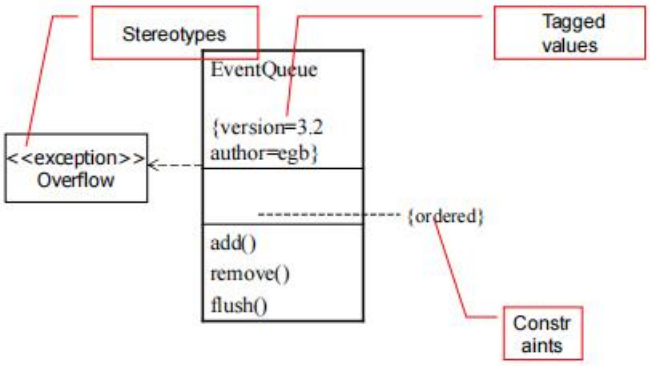
通用划分 Common divisions

可扩展性机制 Extensibility mechanisms

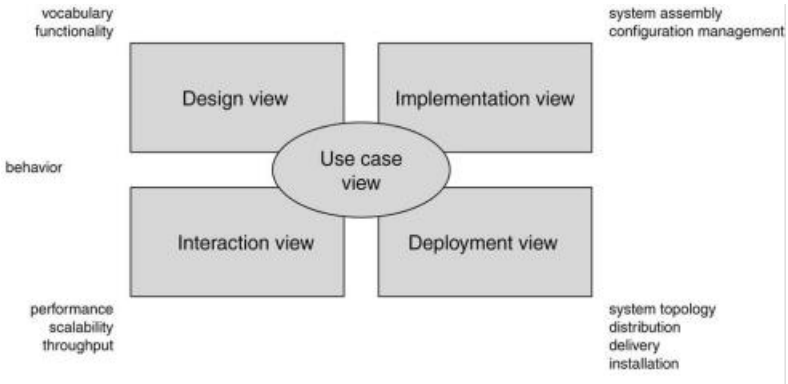
版型 stereotypes

标记值 Tagged values

约束 Constraints



4+1view



用例图 Use Case

用例图是 UML 的一部分，用例的确定通常是面向对象开发过程的第一步

用例：

定义：

用例是一种捕获系统需求的方法，即系统应该做什么。用例是对行为的一种规范

用例定义了外部参与者和系统之间的交互，以实现特定的目标

名称：

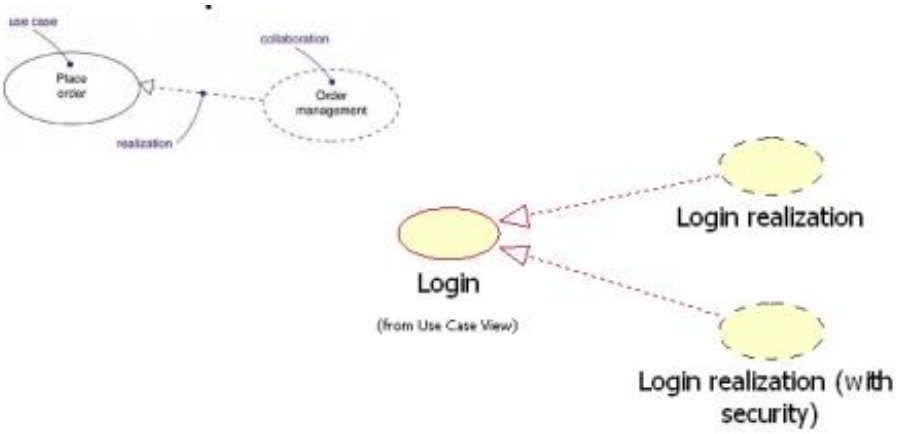
唯一，文本字符串，简短的动词短语

表示：

协作（collaboration）

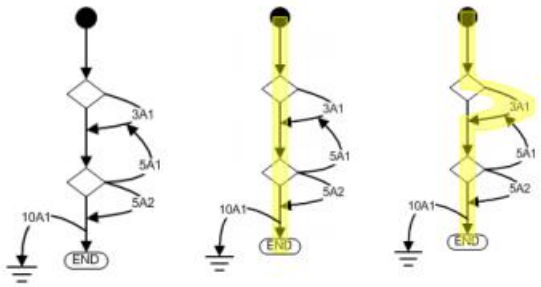
协作是一个由类、接口和其他元素组成的社会，它们共同工作，提供一些比其所有部分的总和更大的合作行为

在图形上，协作被呈现为带有虚线的椭圆



场景（Scenarios）

对于每个用例，您将找到主要场景（定义基本序列）和次要场景（定义备选序列）



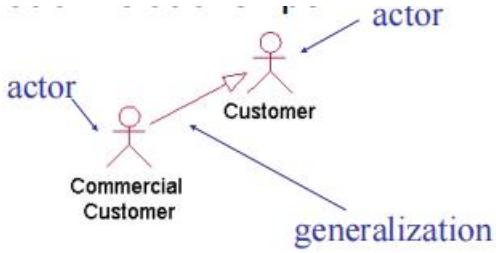
用例图：

参与者 Actor：

参与者表示用例的用户在与这些用例交互时所扮演的一组连贯的角色（可能是人、硬件设备、甚至是其他系统）

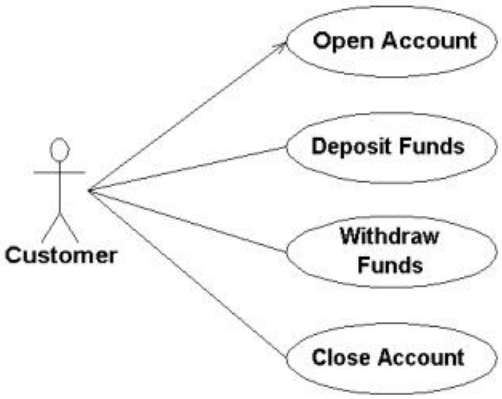
表示：

参与者间的泛化 generalization 关系：



参与者与用例间的关系：

只能通过关联连接到用例

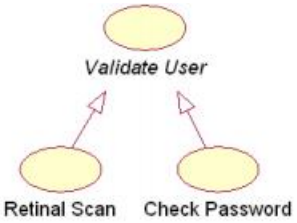


用例间的关系：

泛化（Generalization）

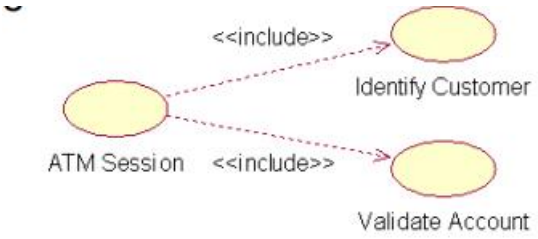
意味着子用例继承了父用例的行为和意义，子代可以添加或覆盖其父代的行为；子代可以被替换为父代出现的任何地方。

图形：带有三角箭头的实有向线



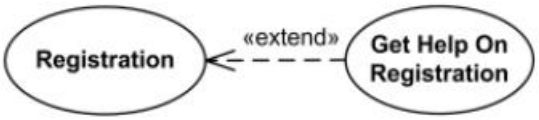
包含（Include）

基本用例显式地合并了在基本用例中指定的位置上的另一个用例的行为，被包含的用例是必须被执行的（从基本用例到所包含的用例）【呈现为依赖，但版型为include】



扩展（Extend）

基本用例可能是独立的，但在某些条件下，它的行为可以被另一个用例的行为所扩展，可以不执行（从扩展用例指向基本用例）【扩展关系被呈现为依赖关系，版型为扩展】



总结：

Relation	Description	Symbol
Association	Between actor and use case	———
Generalization	Between actors, between use cases	———▷
Include	Between use cases	<<include>>
Extend	Between use cases	<<extend>>

关联、依赖、泛化都是关系，包含和拓展是特殊的依赖

用例规范：

用例规范为用例提供了文本详细信息

用例名称 Use case name　　说明用例名称

简要描述 Brief description　描述用例的角色和目的

事件流 Flow of events　　介绍基本流和替代流

基本流 Basic flow　　描述系统的理想的主要行为

替代流 Alternative flows　描述与基本流的异常或偏差，例如当参与者输入错误的用户 ID 和用户身份验证失败时，系统会如何行为

特殊要求 Special requirements　一种特定于用例的，但在事件的用例流的文本中没有指定的非功能性要求

先决条件 Preconditions　在执行用例之前必须存在的系统状态

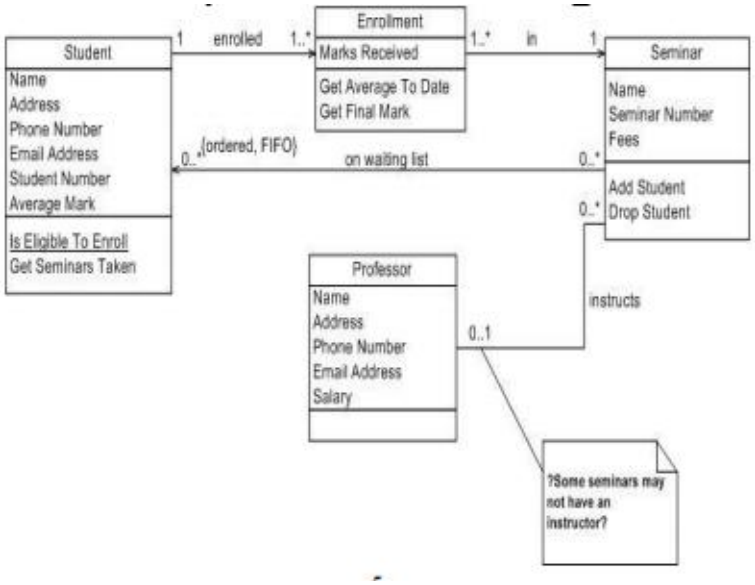
后置条件 Post-conditions　用例完成后立即显示系统的可能状态列表

扩展点 Extension points　在引用另一个用例的事件的用例流中的一个点

建模风格：

1. 用例名称：强动词短语
2. 在用例放置上，暗含时间考虑
3. 用单数的、与领域相关的名词来命名参与者
4. 将参与者放在图表的左上角
5. 不使用 <<uses>>, <<includes>>,or <<extends>>

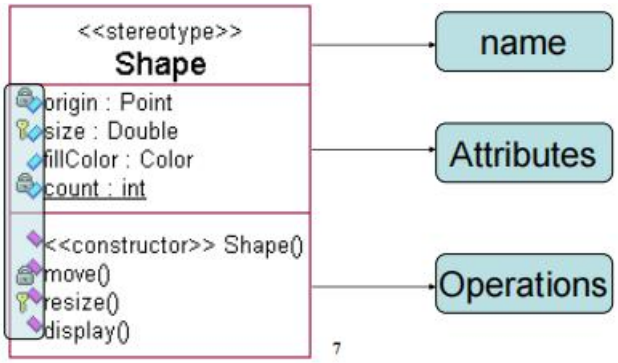
类图 Class Diagram



类 class

类是对一组共享相同属性、操作、关系和语义的对象的描述

表示：矩形



类名：唯一字符串（封闭包内），首字母通常大写【简单名称、限定名称】

属性 Attributes：

属性是一个类的一个命名属性，它描述了该属性的实例可能包含的一系列值

属性列在类名下面的隔间中

属性的形式：

[visibility] name [':' type] ['[' multiplicity ']'] ['=' initialvalue] [property-string {',' property-string}]

可见性[visibility]
- +:Public
- #:Protected
- -:Private
- ~:Package

多样性[Multiplicity]

0..1	0 or 1
1	1 exactly
0..*	0 1 or more
*	0 1 or more
1..*	1 or more
3	3 exactly
0..5	between 0 and 5
5..15	between 5 and 15

➢Name:String="COSE SEU."{readOnly}

```
public class ManagementSystem
{
private final String Name = "COSE SEU.";
}
```

属性约束[Property-string]

操作 Operations：

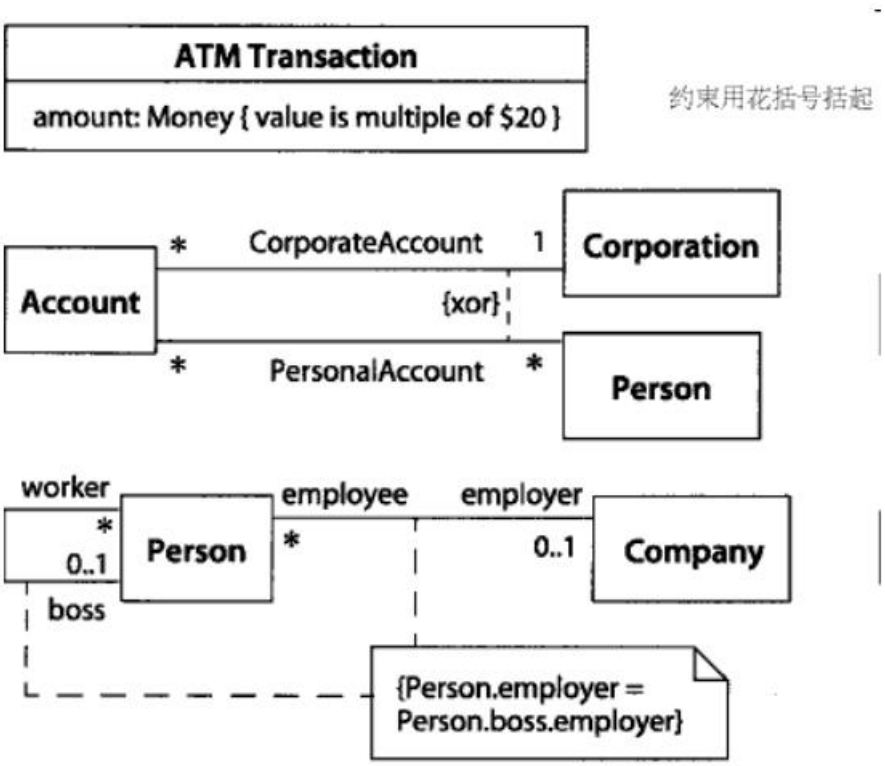
操作是可以从类的任何对象请求以影响行为的服务的实现，一个类可能有任意数量的操作，或者根本没有任何操作

操作被列出在类属性下方的一个矩形中

[visibility] name ['(' parameter-list ')'][':' return-type][property-string {',' property-string}]

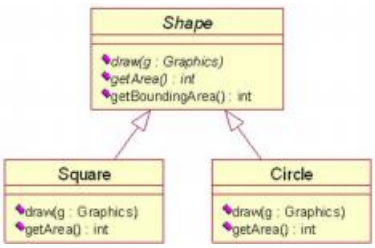
约束（Constraint）

有了约束条件，您可以添加新的语义或扩展现有的规则，运行时必须满足的条件



抽象类 Abstract Class：

不能直接实例化的类，在 UML 中，您可以通过用斜体书写类的名称来指定类是抽象的（也可以用版型）

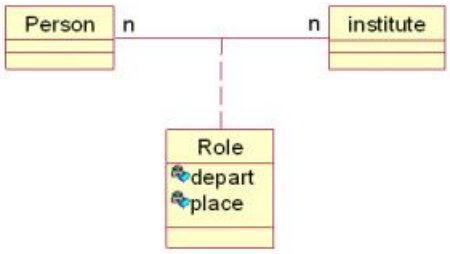


接口 Interface：

接口是用于指定类或组件的服务的操作的集合，从图形上看，接口可以呈现为一个定型的类，以公开其操作和其他属性

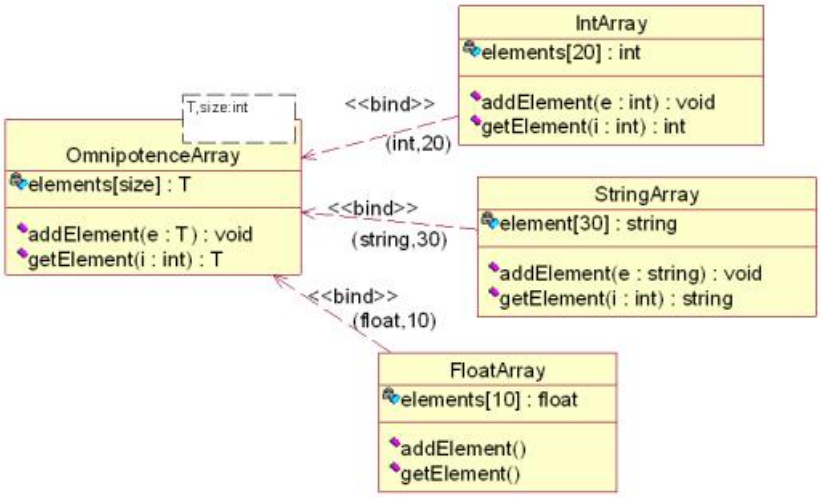
关联类 Association class：

一个同时具有关联属性和类属性的建模元素，可以将关联类作为由虚线附加到关联线的类符号呈现



模板类 Template class:

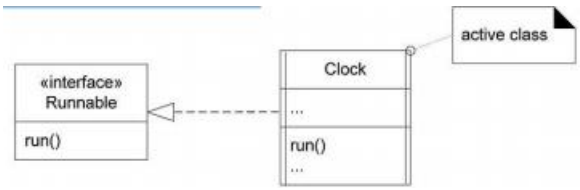
一个模板是一个参数化的元素，模板可以包括类、对象和值的插槽，并且这些插槽作为模板的参数



活动类 Active class：

活动对象是拥有进程或线程并可以启动控制活动的对象，活动类是指其实例是活动对象的类

图形上，活动类呈现为左右两侧双线的矩形。进程和线程被呈现为定型的活动类。



嵌套类 Nested class：

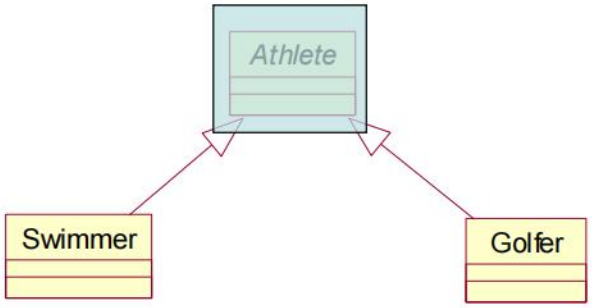
一个类可以在另一个类的范围内进行声明。这样的类被称为“嵌套类”。

嵌套类被认为是在封闭类的范围内，并且可在该范围内使用

类之间的关系

泛化 generalization：

泛化是指一般类型的东西（称为超类或父类）和更具体类型的东西（称为子类）之间的关系



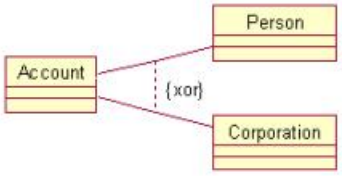
关联 association：

关联是一种结构性关系，它指定一个事物的对象连接到另一个事物的对象

一个关联至少有两个关联末端，每个关联端点都应该连接到一个类,可以限制单向关联

关联可以有一个名称，您可以使用该名称来描述关系的性质
如果有多个关联连接相同的类，则需要使用关联名称或关联结束名来区分它们
如果关联在同一个类上有多个端，则需要使用关联端名来区分端名

关联约束 Constraint of Association

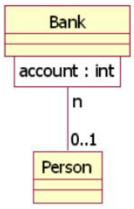


关联角色的多样性 Multiplicity of Association’s Role

- 0..1
- 0..n
- 1
- 1..*
- 0..*
- 7
- 3, 6..9
- 0

关联资格 Qualification of Association

您可以将限定符呈现为附加到关联结尾的小矩形，并将属性放置在矩形中

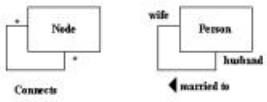


关联性 Arity of Association

每个关联都有特定的关联，因为它可以关联两个或多个项目

递归性或反射性关联 Recursive or Reflexive Association

递归关联是指一个类与它自身关联的地方（该类的实例 A 与实例 B 关联）



二向关联 Binary Association

恰好连接两个类的关联称为二向关联

N-ary 关联 N-ary Association

可以有连接两个以上类的关联；这些被称为 n-ary 关联

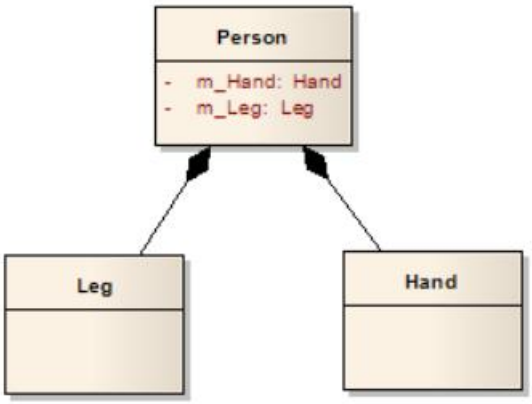
聚合 aggregation:

一种特殊的关联形式，指定集合（整体）和组件（部分）之间的全部分关系

两个类之间的普通关联表示对等体之间的结构性关系，这意味着两个类在概念上处于同一级别，没有一个比另一个更重要，表示一种“有”关系，意味着整体的对象有部分的对象。

组合 Composition

组合是一种聚合的形式，具有强大的所有权和一致的寿命作为整体的一部分



聚合和组合区别

1、依赖性区别

聚合中的两种类（或实体）是可以单独存在的，不会相互影响；被关联的一方可以独立于关联一方，依赖性不强。

相反，组合中的两个实体（或者类）是高度依赖于彼此的，它们之间会相互影响。

2、关系类型的区别

聚合代表了has-a关系，一种单向关系；组合代表了part-of关系。

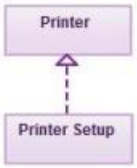
3、关联强度的不同

聚合是一种弱关联关系；组合是一种强关联关系。

4、生命周期的不同

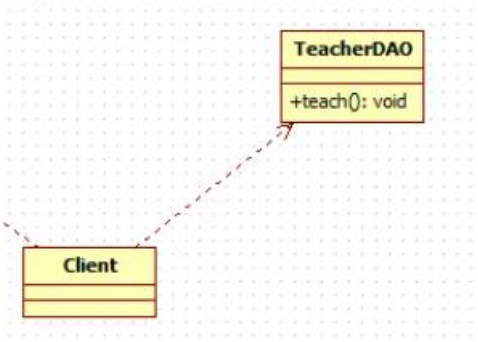
在聚合关系中的两个类（或实体）的生命周期是不同步；但，在组合关系中的两个类（或实体）的生命周期是同步的。

实现 Realization

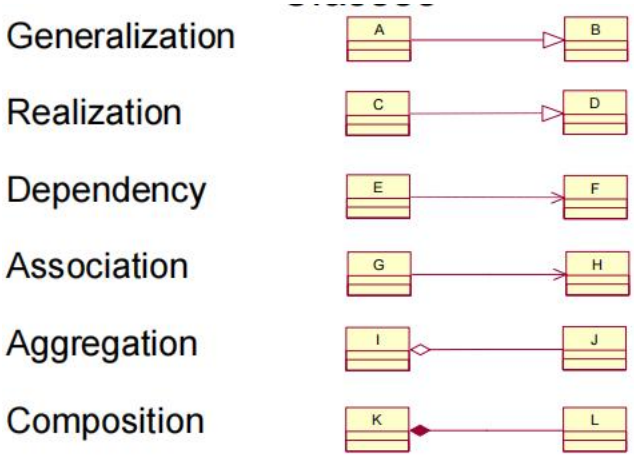


依赖 Dependency

依赖关系是一种关系，它声明一个事物（例如，类窗口）使用另一个东西的信息和服务（例如，类事件），但不一定是相反的



关系的强度：组合、聚合、关联、依赖



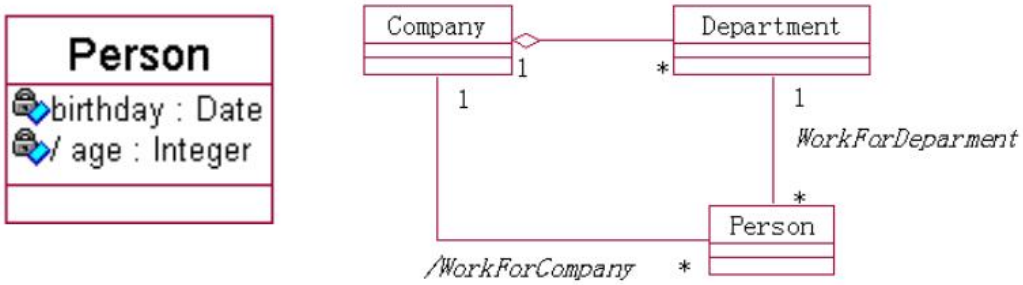
（依赖和实现是虚线）

派生关联和派生属性

派生的关联和派生的属性可以分别从类图上的其他关联和属性中计算出来

派生属性的名称将自动以斜杠（/）作为前缀

与派生属性一样，类图中派生关联的名称前面有一个斜杠（/）

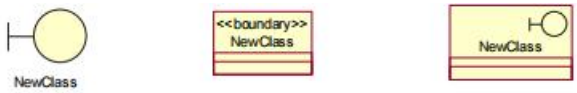


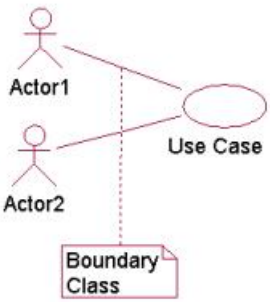
/age 和 /WorkForCompany 为派生属性和派生关联

版型：

边界类 Boundary class

边界类是用于模拟系统环境与其内部工作之间相互作用的类



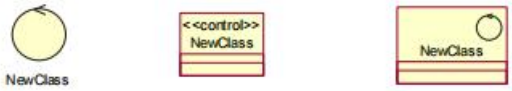


实体类 Entity Class

是一个用于为必须存储的信息和关联行为建模的类

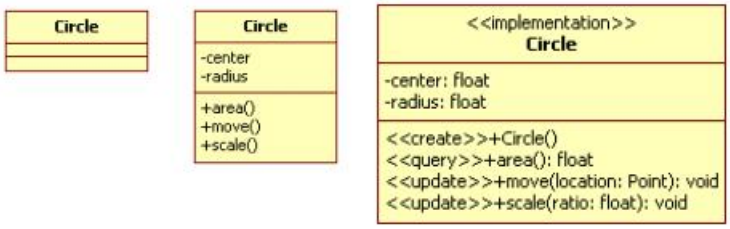
控制类 Control Class

控制类是一个用于为特定于一个或几个用例的控制行为建模的类



类图

类图是显示一组类、接口和协作及其关系的图。图形上，类图是顶点和弧的集合。



Conceptual Specification Implementation

面向对象类设计原则

开闭原则 Open Closed Principle (OCP)：

软件实体对于扩展是开放的，对于修改是封闭的，我们应该编写我们的模块，以便它们可以被扩展，而不需要修改它们。

Liskov 替换原则 Liskov Substitution Principle (LSP)：

子类型必须能够替换掉它们的基类型。

依赖倒置原则 Dependency Inversion Principle (DIP)：

依赖反转是一种依赖于接口或抽象函数和类，而不是具体函数和类的策略。

接口隔离原则 Interface Segregation Principle (ISP)：

许多特定于客户端的接口 优于一个通用接口。

建模风格：

1. 属性名称和类型保持一致
2. 不要命名具有关联类的关联
3. 按降低可见性的顺序列出操作/属性
4. 始终指示多重性
5. 避免使用“*”表示多重性
6. 不要对每个依赖项都进行建模
7. 将整体放在零件的左侧
8. 不要纠结组合还是聚合，不确定就关联

包图 Package Diagram

Basic building block → thing → grouping thing

包 Package：

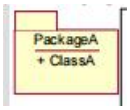
包是一种用于将建模元素组织成组的通用机制

用途：

1. 建模一组元素
   将相同的基本类型的元素分组
2. 建筑视图建模
   使用软件包来建模体系结构的视图

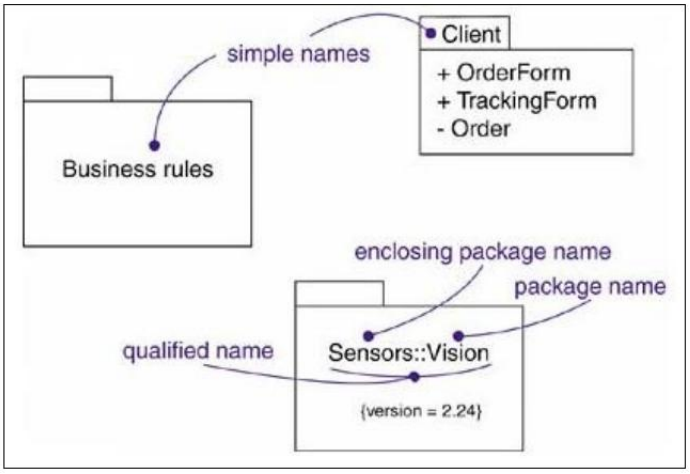
包的表示：

标签文件夹，名称会出现在文件夹中（若内容未显示），或选项卡中（显示了文件夹的内容）



包的名称（区别于别的包的名称）

字符串，简单名称 simple name、限定名称 qualified name



包含有的元素

类、接口、图、节点、用例……

包的可见性 visibility（见类的可见性）

- +: public
- #: protected
- -: private

扩展机制（版型）stereotype：

<<system>> <<subsystem>> <<facade>> <<stub>> <<framework>> ……

包之间的关系：

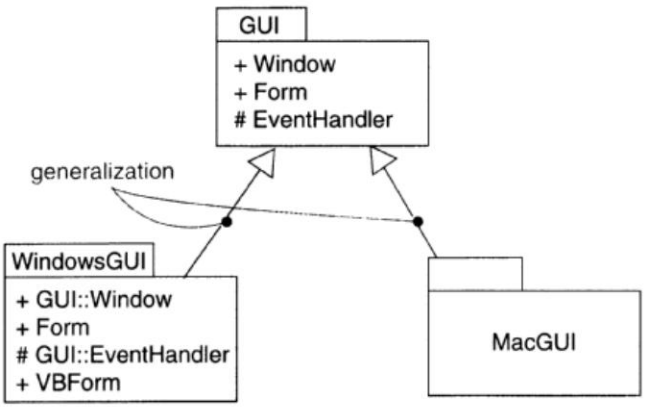
依赖 dependency：

<< import>> import 关系使命名空间合并（using,import)

<< access>> 客户包中的元素能访问提供者包中的所有公共元素但是命名空间 不合并，仅把提供者包的内容附加到客户包的私有命名空间

<< trace>> 表示一个包到另一个包的历史发展，例如不同版本的包。

泛化 generalization



包图：

包图将模型本身分解为组织单元及其依赖关系

包的设计原则：

重用释放等价原则 The Release Reuse Equivalency Principle（REP）

把类放入包中时，应考虑把包作为可重用的单元，即要求新版本的可重用类容易替换旧版本的可重用类

共同重用原则 The Common Reuse Principle（CRP）

通常，一个包中的元素（类）要么都可重用，要么都不可重用。不会一起使用的类，不要放在同一个包中

共同闭包原则 The Common Closure Principle（CCP）

把可能同时修改、同时维护的类放到一个包中，以便于维护和升级

非循环依赖原则 The Acyclic Dependencies Principle（ADP）

包之间不要形成循环依赖关系；循环依赖是由于分包不当造成的

包图的设计风格：

1. 合乎逻辑地组织设计

(1) 一个框架的类属于同一个包中

(2) 在同一继承层次结构中的类通常属于同一个包中
(3) 通过聚合或组合相互关联的类通常属于同一个包中
(4) 经常相互协作的类通常属于同一个包中

2. 在包上明确版型
3. 用例图中包括 actor，水平排布用例-包图

状态机图 State Machine Diagram

状态 State：

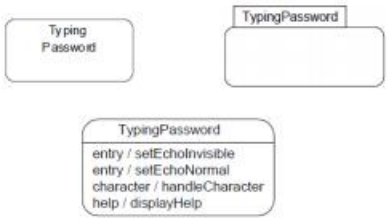
是指事物满足某个条件、执行某个活动或等待某个事件的一段时间

状态的组成：

名字、进入/离开时的影响、内部过度、子状态、延迟事件

状态的表示：

圆角矩形



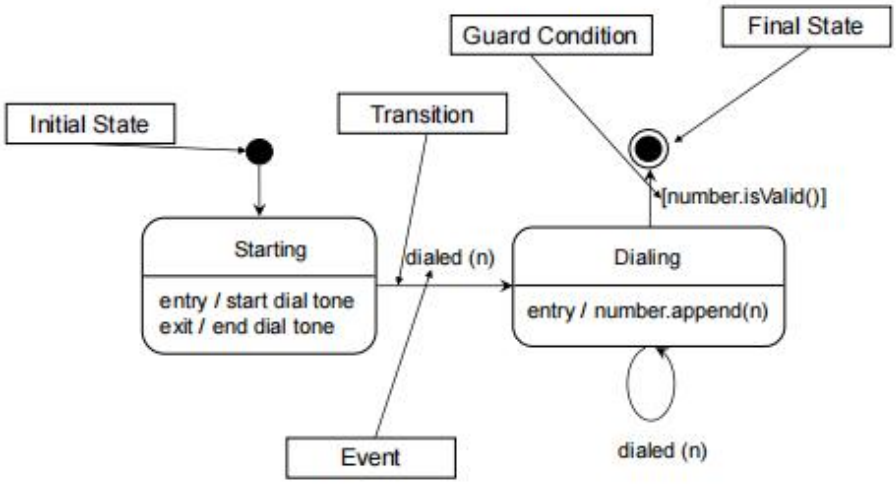
状态机 State Machine：

对单个对象的生命周期进行建模，无论它是类的实例、用例，甚至是整个系统

状态机图：

图形表示：

状态、转移、事件、效果



状态 state：

初态  和终态  （终态可以不表示出来）

原状态 Source state：受转移影响的状态

目标状态 Target state： 转移完成后的状态

转移 transition：

两个状态之间的关系，表明当指定事件发生并满足指定条件时，处于第一个状态的对象将执行某些操作并进入第二个状态

触发事件 Event trigger：

信号事件、调用事件（e.g.使用函数）、时间事件、改变事件

监控条件 Guard condition：

一种布尔表达式，在接收事件触发器触发转换时进行计算

true：触发

False：不触发

如果没有可以由同一事件触发的其他转换，则该事件将丢失

影响 effect：

内联计算 inline computation，操作调用 operation calls，创建、销毁对象，向一个物体发送信号

转移类型：

外部转换 External transition：包括进入/退出活动

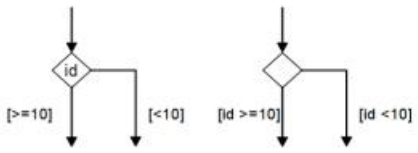
内部转换 Internal transition：不包括进入/退出活动

局部转换 Local Transition：（只存在与复合状态中）

在父状态过程中进入子状态然后子状态结束后继续刚才的父状态

选择伪状态 Choice Pseudo-State：

一个选择的伪状态显示为一个菱形，一个过渡到达，两个或多个过渡离开

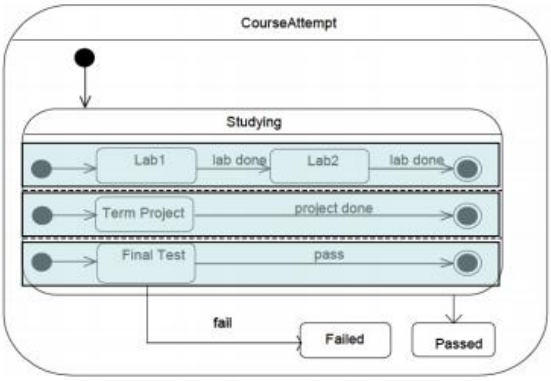


复合状态 Composite state：

具有子状态（嵌套状态）的状态称为复合状态

包含一个：concurrent 顺序子状态（studying）

包含多个：sequential 并发子状态（lab1，term project， final test）



历史状态 History State：

历史记录状态用于记住状态机被中断时的先前状态

建模风格：

监护条件不要重叠

不在初始过渡设置监护条件

仅在适用于所有进入（退出）过渡时，才指示进入（退出）操作

注意黑洞 “Black-Hole” States 和 “Miracle” States

对象图 Object Diagram

实例和对象 Instances and objects：

实例：

实例是一种抽象的具体表现，这个抽象可以应用到一组操作，并可能具有存储操作效果的状态

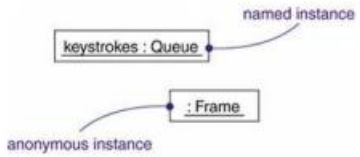
1）使用实例建模具体事物

2）抽象表示一个事物的理想本质

3）一个实例表示一种具体的表现形式

图形表示：

下划线标注名称（可视化命名实例与匿名实例）



UML 建模的大多实例都是类的实例，即对象

对象：

在一般意义上，一个物体是指在现实世界或概念世界中占据空间的东西，你可以对

它做一些事情

对象是一个具有身份、状态和行为的实体

对象的状态：

对象也有状态，在这个意义上，状态包含对象的所有属性加上这些属性的当前值

对象的操作：

可以对对象执行的操作在对象的抽象中声明

对象（实例）的命名：

一个实例有一个可以将其与上下文中其他实例区分开的名字

Example of Object

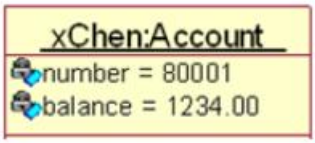
- Person: A class
- Individual: an object (each has its own individual characteristics)
  - State: height, weight, education, job, income .....
  - Operations: read, exercise, eat .....
  - Name

对象的表示：

名称：文本字符串

可以简单命名一个对象，并省略其抽象。也可以视情况呈现一个匿名对象。

属性：



对象之间的关系：

链接是对象之间的语义连接

1）通常，链接是关联的一个实例

2）如果一个类与另一个类有关联，则这两个类的实例之间可能存在链接

3）在两个对象之间有链接的地方，一个对象可以向另一个对象发送消息

对象图：

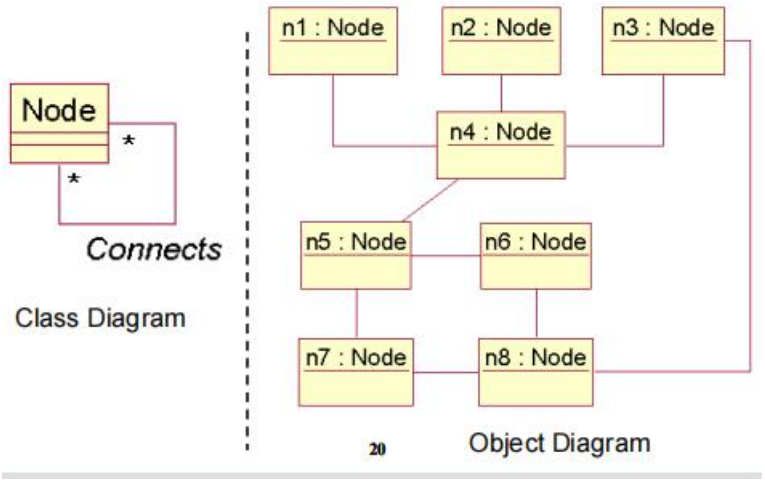
对象图是一种显示一组对象及其在一个时间点上的关系的图

内容：对象、链接

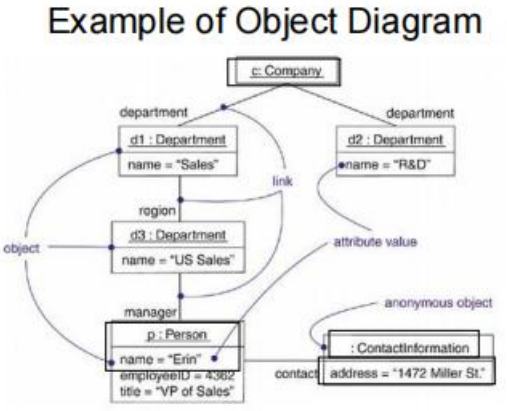
图形：顶点、圆弧

实例图与类图

类图描述了一般情况，而实例图则描述了从类图中派生出来的特定实例



使用对象图来建模系统的静态设计视图或静态流程视图，就像使用类图一样，但是从真实或原型实例的角度来看



对象图的应用：

1）建模复杂的数据结构
2）类模型的验证
3）分析并描述源代码

对象图的建模风格：

1）明确指出实例的属性值
2）对象名比属性更重要
3）指示用来区分不同关系的角色

活动图 Activity Diagram

显示了从活动到活动的流程

使用活动图来建模一个工作流或一个操作

组成：

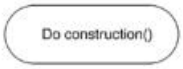
初始节点 Initial node：初始节点作为执行活动的起点 ●

结束节点 Final node：最后一个节点是活动中的流停止的控制节点

活动最终节点：

流最终节点：

活动节点 Activity Node：一个活动中的一个组织单位

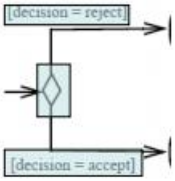


控制流 Control Flows：当一个操作或活动节点完成执行时，控制流将立即传递到下一个操作或活动节点

决策节点和监控条件 Decision Node and Guard Expression：

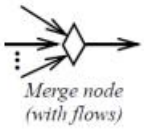
决策节点是一个在传出流之间进行选择的控制节点

监控条件是活动流向必须满足的条件



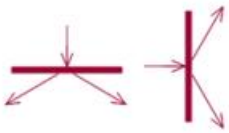
合并节点 Merge：

合并节点是一个控制节点，它将多个流聚集在一起而无需同步



分叉节点 Fork Node：

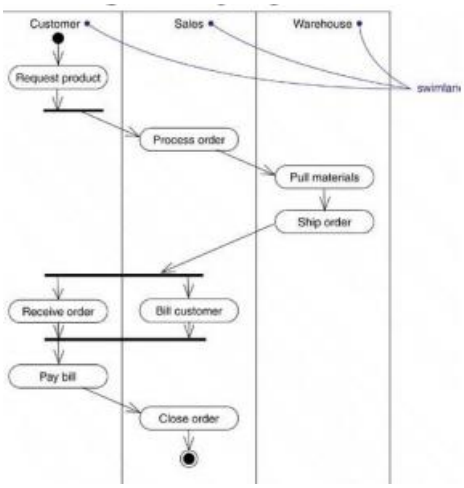
分叉表示将单个的控制流分割为两个或更多的并发的控制流

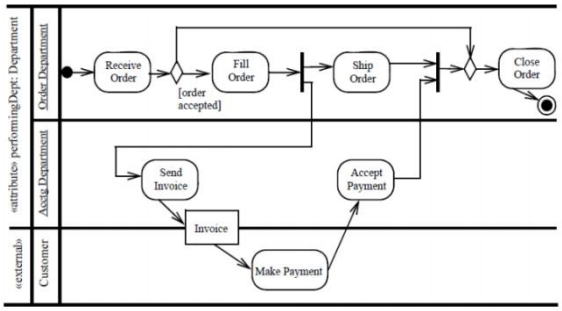


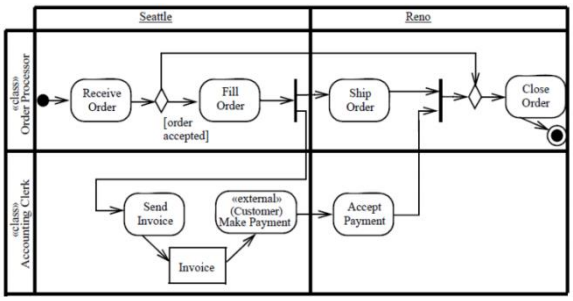
汇合节点 Join Node：

汇合表示两个或两个多个并发控制流的同步

泳道（Swim lane）：

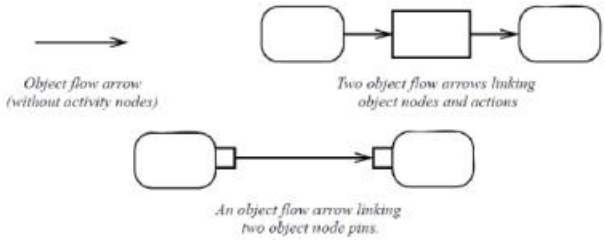






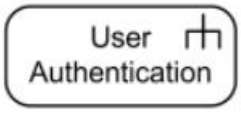
对象流（Object Flow）：

对象可能参与到与活动图相关联的控制流中

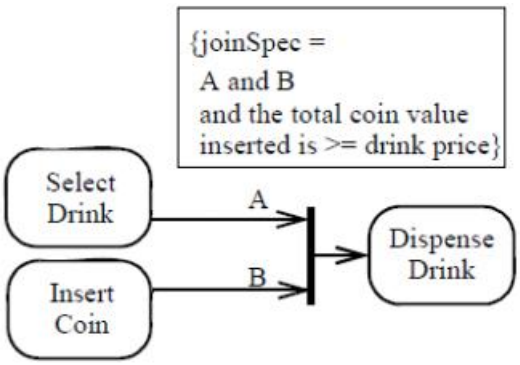


子活动（sub-activity）：

耙型表示有更详细的活动描述

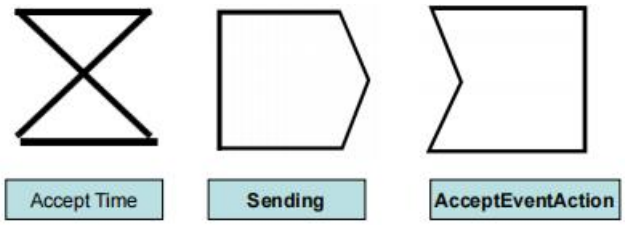


结合说明（Join Specification）：



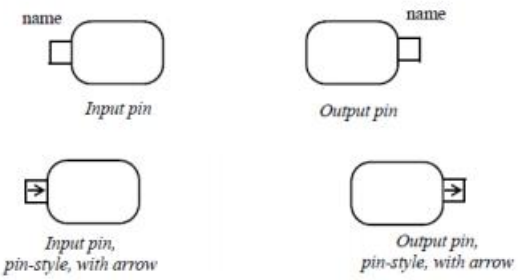
信号行为（Signal Action）：

信号是一种事件，它表示在实例之间通信的异步消息的规范



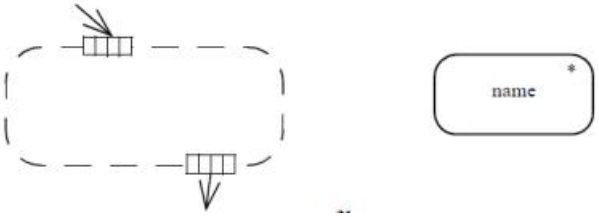
引脚（Pin）：

表示对操作的输入或对操作的输出



扩展区域（Expansion Region）：

扩展区域是一个结构化的活动节点，它多次执行其包含的元素，并对应于输入集合中的元素



活动图的建模风格

1) 确保离开决策点的每个活动边缘都有一个防护装置
2) 确保在决策点上的监控条件形成一个完整的集合
3) 不要使监控条件有交集
4) 确保分叉处有相应的连接处
5) 确保一个分叉只有一个条目
6) 确保联接只有一个出口
7) 游泳泳道少于 5 条
8) 模拟主要泳道中的关键活动

顺序图 Sequence Diagram

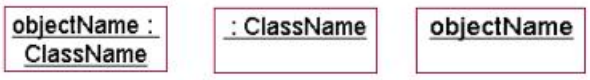
顺序图是一种强调消息的时间顺序的交互图

顺序图显示了一组角色以及这些角色的实例发送和接收的消息

建模元素：

沿 X 轴：对象　　沿 Y 轴：消息按时间递增

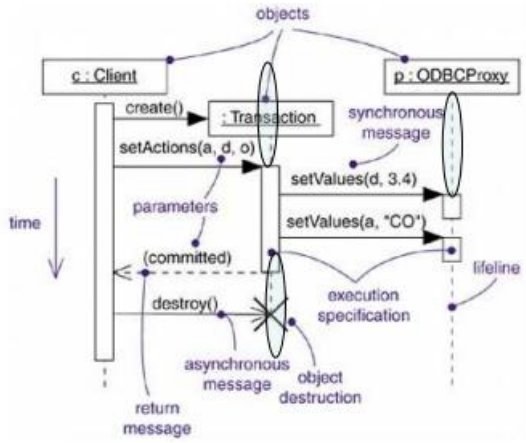
对象：将参与交互的对象或角色放置在图表的顶部



生命线 Lifeline：对象生命线是表示一段时间内对象存在的垂直虚线

一般同时开始同时结束

但也可以在交互中创建/销毁对象

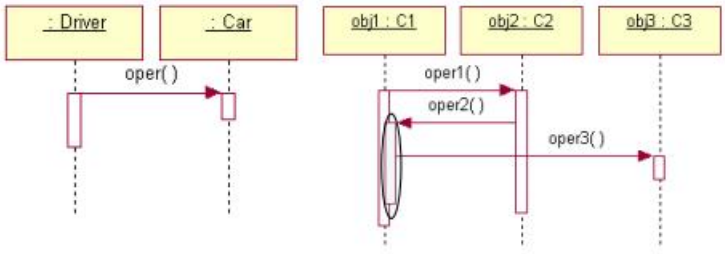


控制焦点（激活器）Focus of control：

控制焦点是一个又高又细的矩形，它显示了一个物体直接执行一个动作或通过一个下级程序来执行一个动作的时间段

矩形的顶部与动作的开始对齐；底部与完成对齐（可以用返回消息标记）

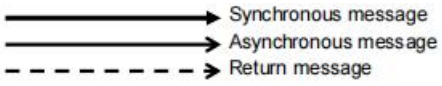
可以通过显示控制焦点的嵌套(由递归、对自操作的调用或来自另一个对象的回调引起)，稍微叠加到父焦点的右侧（可以达到任意的深度）



消息：

顺序图的主要内容

通过一个箭头从一条生命线到另一条，箭头指向接收器



同步、异步、返回消息

当你传输消息时，通常会有一个操作来响应返回，这个操作可能会造成目标对象及可访问对象的状态变化。

调用、返回、发送、创建、销毁

结构化控制 Structured Control：控制操作符

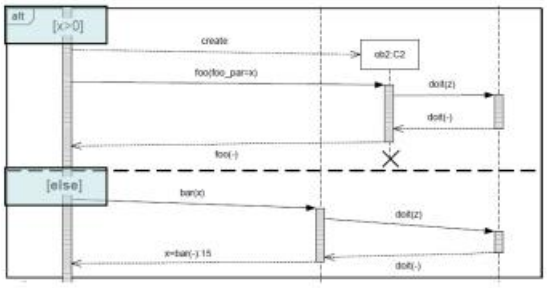
一个控制操作符在序列图中显示为一个矩形区域，左上角五边形说明种类

opt：

如果在输入操作符时，保护条件为真，则执行控制操作符的主体

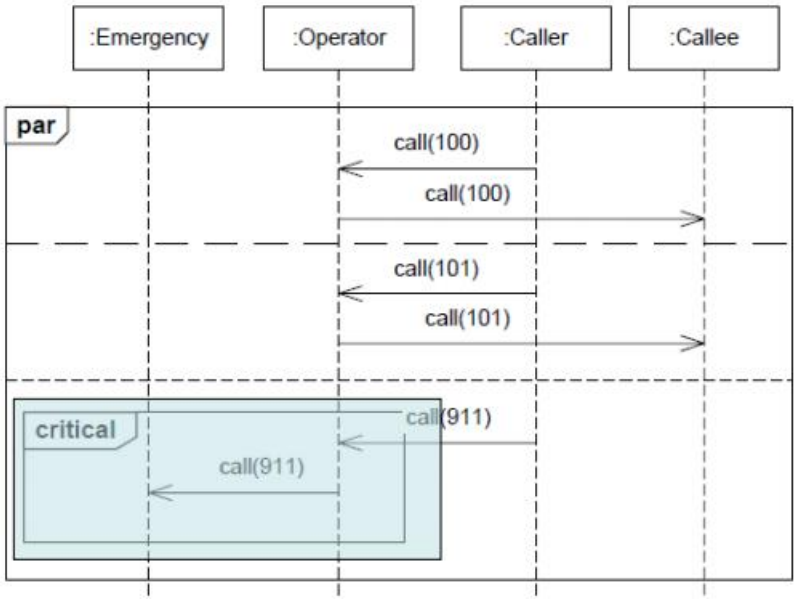
alt：

分成多个子区域，每个子区域对应一个条件区域的分支，每个子区域都有一个 guard condition，如果成真则执行，但最多执行一个



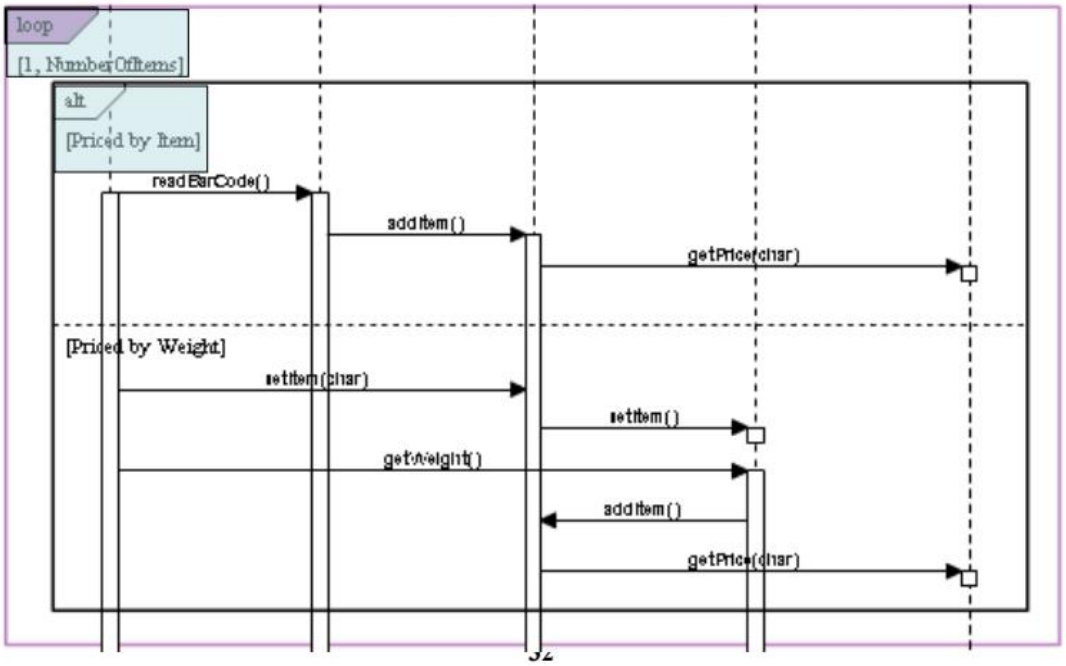
par：

划分多个子区域，每个子区域表示一个并行，输入控制操作符时所有子区域并发执行



loop：

只要 guard condition 为真，就重复执行



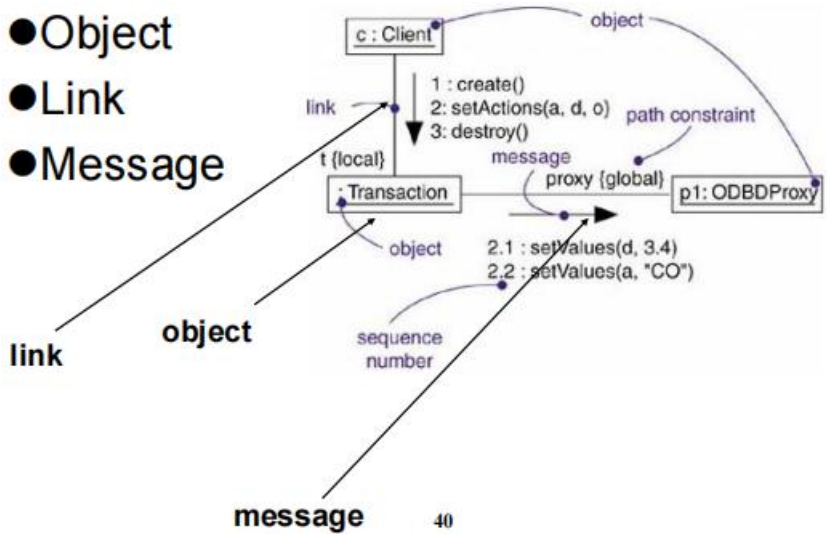
建模风格：

1) 努力实现从左到右的消息排序
2) 当存在多个相同类型时，命名对象
3) 专注于关键的交互

通信图（协作图）Communication Diagram

通信图是一种交互图，它强调了发送和接收消息的对象的结构组织

建模组成：



链接（link）

是对象之间的语义连接，通常来说，链接是关联的一个实例

消息（message）

要指示消息的时间顺序，要在消息前加编号，要显示嵌套，需用（1 是第一个消息，其中包含消息 1.1 和消息 1.2，以此类推）

建模风格：

1) 请不要使用通信图来建模工艺流程，如果要为流程或数据流建模，则应该考虑绘制活动图
2) 当顺序重要时创建顺序图

顺序图与通信图的比较

顺序图强调了消息的时间顺序。通信图强调了参与交互的对象的组织结构

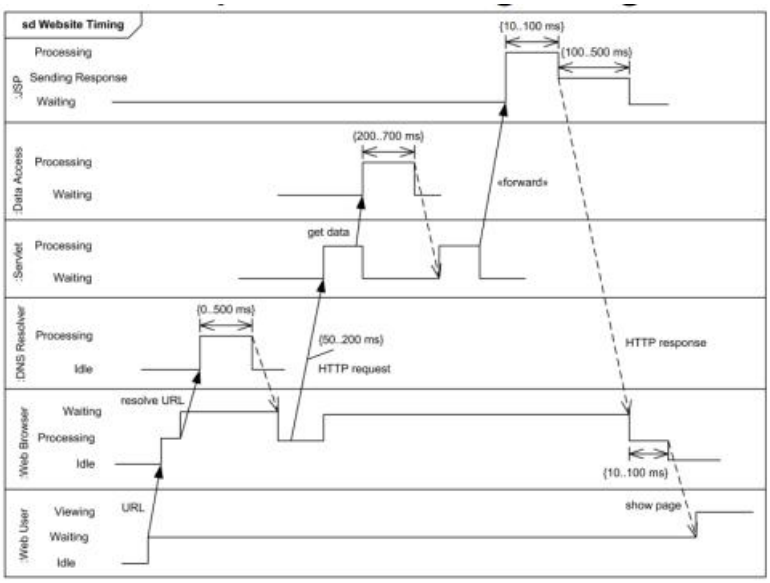
定时图 Timing Diagram

定时图关注的是沿着线性时间轴改变生命线内和生命线之间的条件

UML 定时图用于显示一个或多个元素随时间变化的状态或值的变化。它还可以显示定时事件与控制它们的时间和持续时间约束之间的交互作用

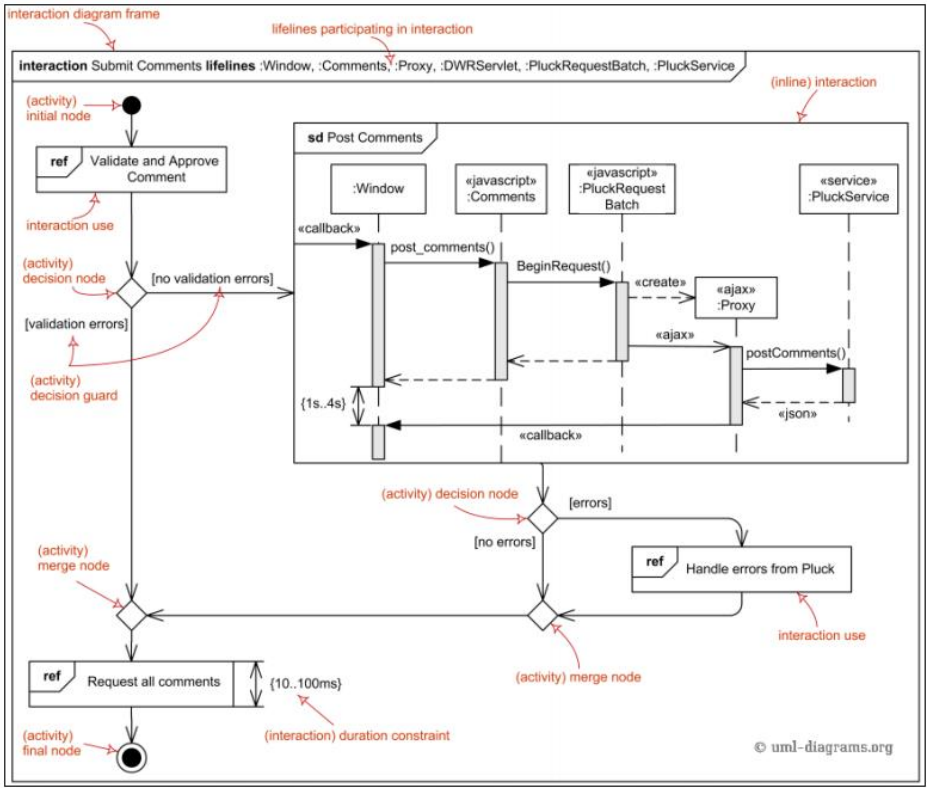
建模形式：

x 轴显示所选择的任何单位的所经过的时间，而 Y 轴被标记为给定的状态列表



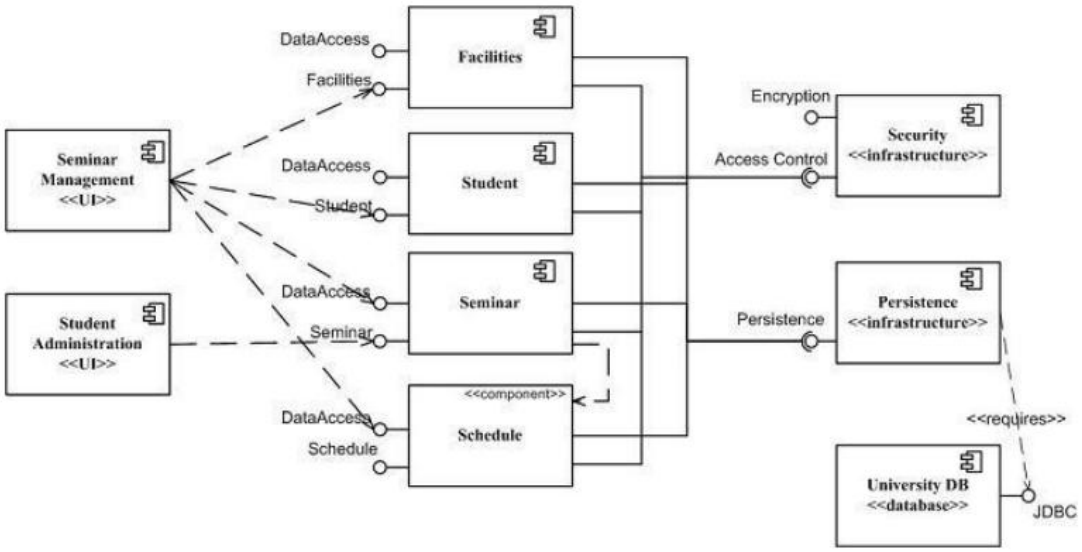
交互概述图 Interaction Overview Diagram

活动图的变体，顺序图的变体



构件图 Component Diagram

显示一组组件之间的组织关系和依赖关系的图表，处理了系统的静态实现视图



构件 Component：

构件是系统中可替换的部分，它符合并提供了一组接口的实现

构件图比类图具有更高的抽象级别。通常，一个组件是在运行时由一个或多个类（或对象）实现的。

构件的接口：

provided interface：提供服务给其他构件

required interface：需要从其他构件获取服务

构件的类别：

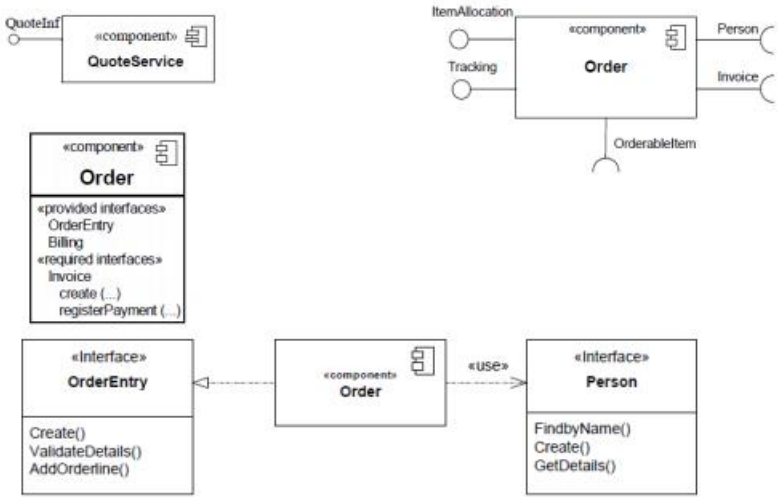
配置构件 Deployment：操作系统、可执行文件

工作产品构件 Work Product：源代码、数据文件

执行构件 Execution Component：系统运行时创建的构件

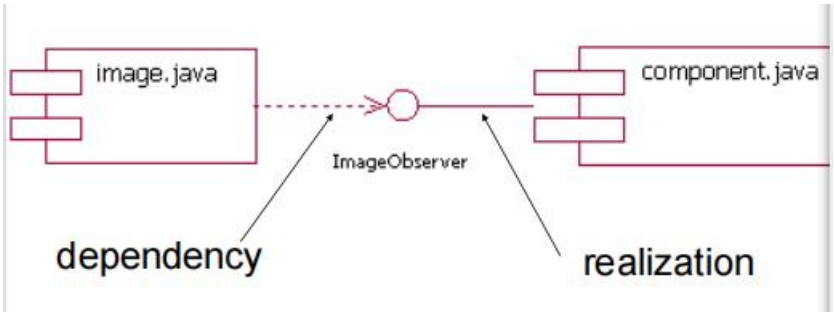
构件的表示：

一个构件显示为一个矩形，在其右上角有一个小的双叉图标，该组件的名称将显示在该矩形中，组件可以具有属性和操作，但这些通常在图表中省略



构件与接口的关系:

依赖/实现



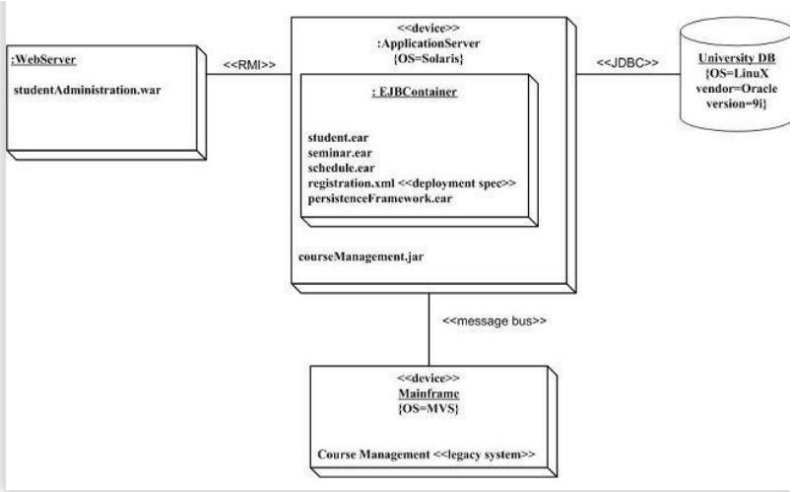
可以嵌套组件

建模风格:

1) 所有组件选择同一种表示方式
2) 仅描述适用于关系图的目标的接口
3) 组件只依赖于接口，而不是其他组件本身

部署图 Deployment Diagram

部署图是用于建模面向对象系统的物理方面的图
显示了运行时处理节点的配置及其上存在的工件



使用 UML，您可以使用部署图来可视化这些物理节点的静态方面及其关系，并指定它们的详细信息

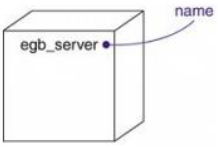
部署图：

包含元素：

节点 Node：

节点是在运行时存在的一种物理元素，它表示一种计算资源，通常至少具有一些内存，通常还具有处理能力

图形上，节点呈现为立方体



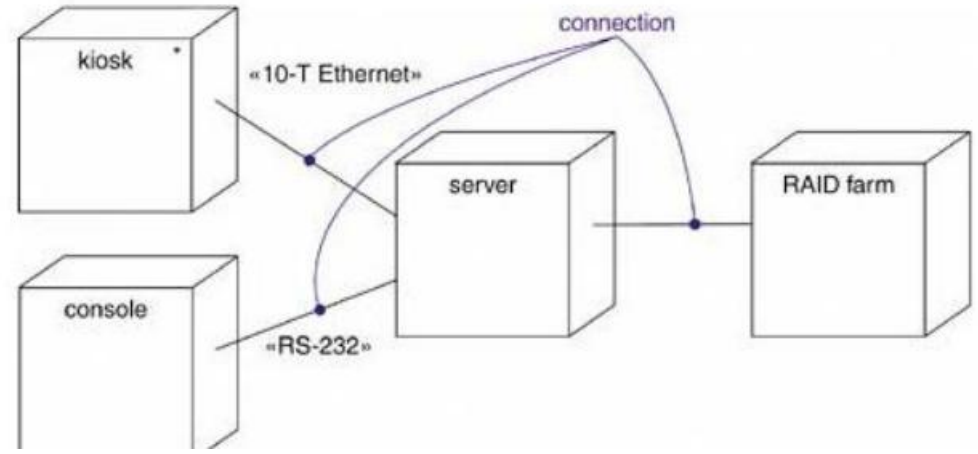
建模处理机和设备：

在对系统的拓扑进行建模时，通常对可视化或指定组成系统的处理器和设备之间的工件的物理分布很有用。

串口 Connection：

您将在节点之间使用的最常见的关系类型是关联

您甚至可以使用关联来建模间接串口，比如远程处理器之间的卫星链接



建模风格：

1) 使用描述性术语来命名节点
2) 只建模是非常重要的软件组件
3) 将视觉版型应用于节点
4) 通过版型来指示通信协议

双向工程 Forward and Reverse Engineering

正向工程 Forward Engineering：

给定 UML 模型，生成相应的代码

逆向工程 Reverse Engineering：

给定某些类的代码，生成相应的模型