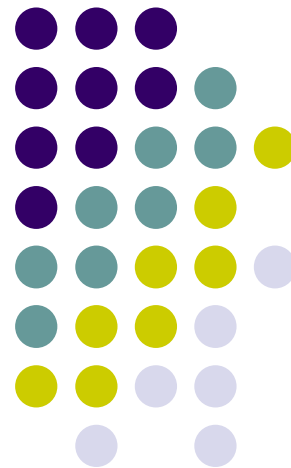


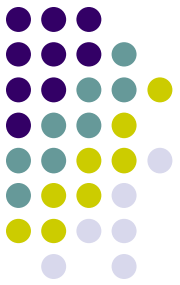
软件系统分析与设计

第五章 静态建模

Class and Object modeling

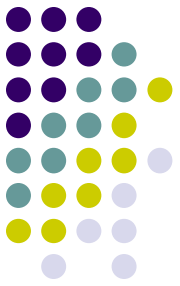
by Dr Y.Yang
Associate Professor
School of Computer Science & Technology
Soochow University
yyang@suda.edu.cn





Review: use case modeling

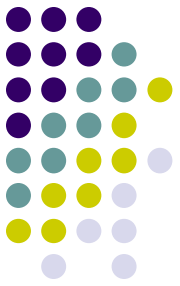
- 如何确定一个系统的范围和边界
- 如何通过客户需求了解、发现和确定用例
- 如何通过客户需求了解、发现和确定行为者
- 如何绘制用例图及其正文描述
- 如何描述用例之间的关联关系和层次



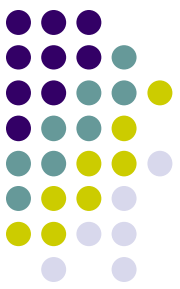
introduction

- 类和对象建模用于描述一个系统的静态结构。
- 类和对象模型有若干对象类图描述。
- 类图有若干类的图形符号及表示之间关系的图形符号组成。
- 类和对象图组成的可视化模型能有效地描述一个软件系统，具有强大的模型描述表达能力。

Today's topics



- 了解面向对象方法的分析和设计之间的关系
- 掌握**UML**中类图的描述方法
- 掌握类之间的关系
- 掌握**UML**中对象图的描述方法
- 掌握接口和包图的描述方法



5.1 面向对象分析和设计的关系

- 复习:

OOA的步骤: **OOA**是一个重复迭代渐增的过程。

(1) 需求分析: 标识场景**scenario**、用例**use case**, 构造需求模型;

(2) 构造系统对象静态模型;

(3) 建造系统对象的动态模型;

(4) 建造系统对象的功能模型;

以上(1) — (4) 反复迭代后利用用例/场景来复审分析模型。

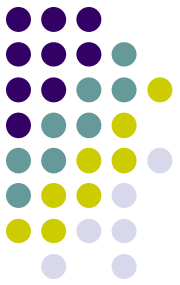
OOD分为两个阶段: 系统对象设计阶段和系统体系结构设计阶段:

系统对象设计: 建立系统整体结构并确认接口

(1) 对象接口设计 (2) 设计算法和数据结构 (3) 确认子系统 (4) 子系统之间的通信

系统体系结构设计: 系统环境选择与体系结构设计

(1) 任务管理设计 (2) 数据管理设计 (3) 人机界面设计



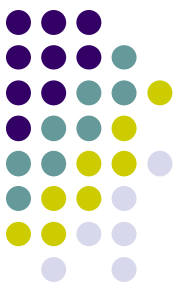
面向对象分析和设计的关系

- 面向对象分析与面向对象设计（OOD）不可截然分开，静态建模既是OOA的部分也是OOD的部分。
- 两者追求的目标不同但采用一致的概念、原则和表示法。
- OOD是以OOA模型为基础，对OOA产生的结构增添实际的计算机系统中所需的细节，如人机交互、任务管理和数据管理的细节。
- 从OOA到OOD是一个模型扩充过程。

类 类图

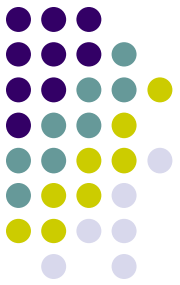


- 类是面向对象系统中最重要单位，它是一组具有相同属性、操作、关系和语义对象的描述。
- 一个类可以实现一个或多个接口。
- 可以用类来捕获正在开发的系统中的词汇。这些类可以是作为问题域一部分的抽象，也可能构成实现。可以用类描述软件事物和硬件事物，甚至也可以用类描述纯粹概念性的事物。
- 结构良好的类具有清晰的边界，并形成了整个系统的职责均衡分布的一部分。



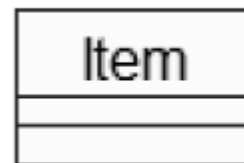
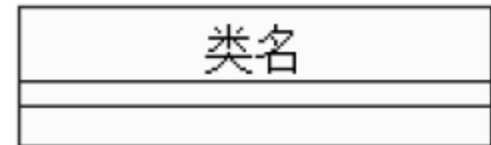
5.2 类图

- 类图（**Class Diagram**）是描述类、接口以及它们之间关系的图，用来定义系统中各个类的静态结构。
- 类图通过系统中的类以及各个类之间的关系描述系统的静态视图。
- 尽管类图与数据模型有相似之处，但是类图不仅显示了系统内信息的结构，也描述了系统内信息的行为。类图中的类可以直接在某些面向对象编程语言中被实现。

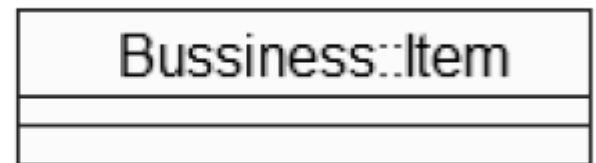


类的表示方法

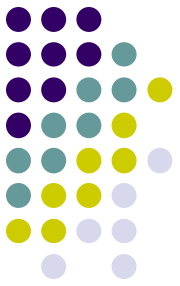
- 在**UML**中，类用一个矩形符号来表示。
- 该矩形被水平线划分为**3**个部分，分别用来定义类的
 - 名称（**Name**）
 - 属性（**Attribute**）
 - 操作（**Operation**）



简单名

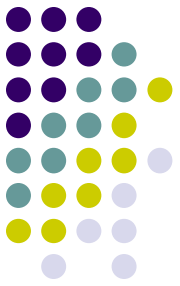


路径名



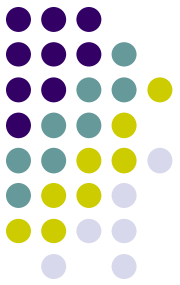
辨析：类和对象的区别

- 名称：
 - 类：类名是一个有意义的标识符
 - 对象：对象名加下划线，对象名后面可以接冒号和类名
- 属性
 - 类：定义属性的类型和属性名
 - 对象：与类的属性名相同，但是有具体值
- 操作（方法）
 - 类：定义操作（函数）——返回类型、操作名、传递参数。
 - 对象：与类中的操作定义相同

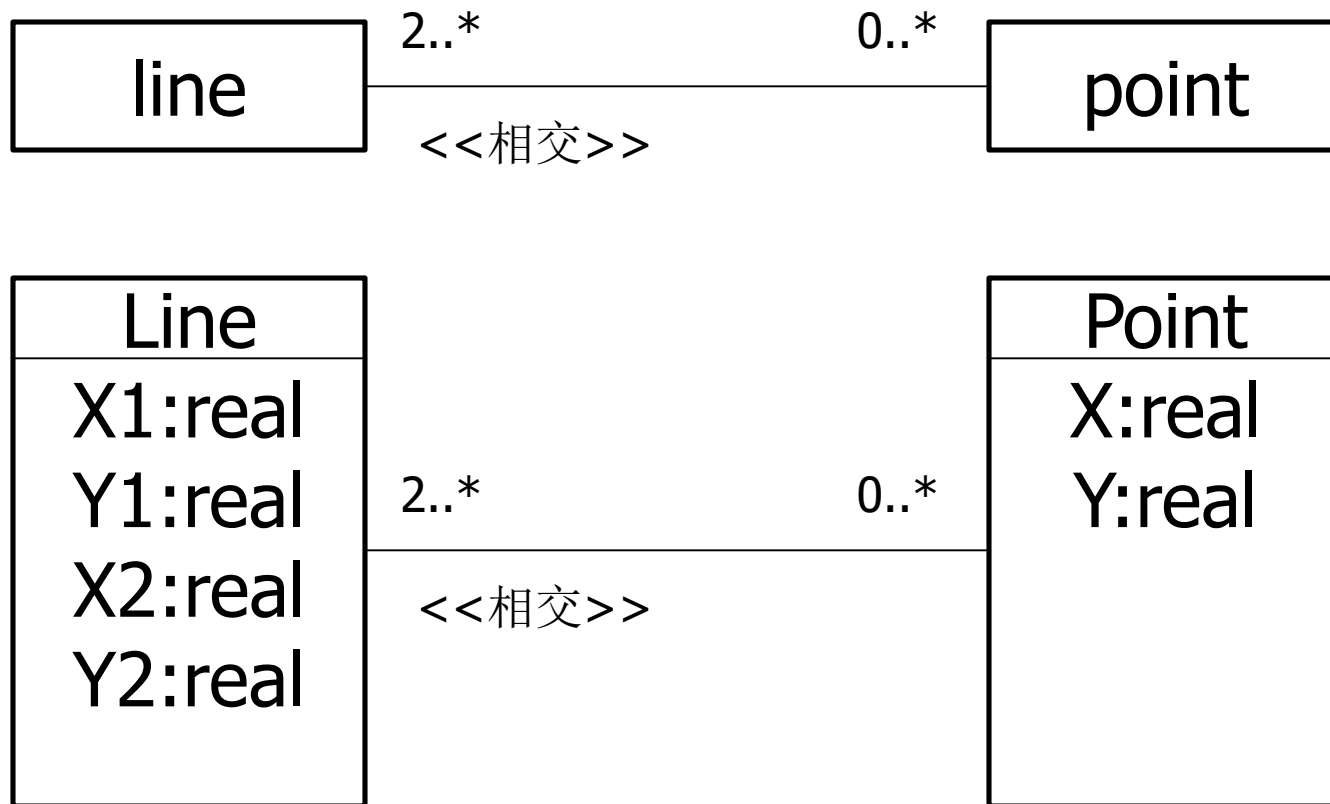


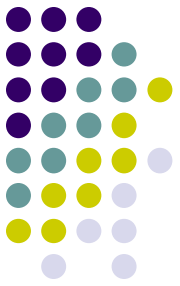
类图和对象图的区别和联系

- 一个系统的静态模型可以包含多个类图：一个类可以出现在几张类图中
- 类的图形符号表示分为：长式和短式
- 对象图是类图的一个实例，它描述了类图中类的特定实例以及某一时刻这些实例之间的特定连接



类图还包括：类与类之间的联系





定义类的属性




- 属性用来描述类的特征。
- 语法格式：

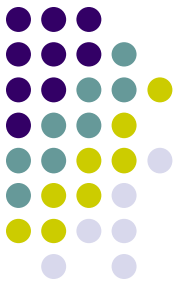
可见性 属性名[多重性]： 类型[=初值]



(1) 可见性

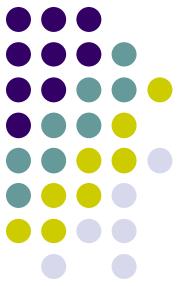
- 描述该属性在哪些范围内可见

可见性	说明	UML 图注	Rose 图注
Public	表示其为公有成员，其他类可以访问（可见）。	+	
Protected	表示其为保护成员，一般用于集成，只能被本类及派生类使用。	#	
Private	表示其为私有成员，不能被其他类访问（不可见），可缺省。如果没有特别说明，属性都应该是私有的。	-	



(2) 多重性

- 任选项，用多值表达式表示，格式为：低值...高值
 - 低值、高值为正整数
 - 0..* 表示0到无限多个
 - 可缺省，表示1..1, 只有1个



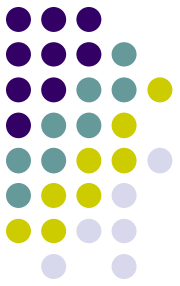
(3) 类型：属性的类型，如整型、字符串、对象等。

(4) =初始值：任选项，初值可作为创建该类对象时这个属性的默认值。

(5) 类属性：用下划线标识，该类的所有对象之间共享该属性。

定义属性时，属性名和类名是必须的，其他部分都是任选的。

属性名可以是像类名那样的文本文字。在实际应用中，属性名是描述属性所在类的一些特性的简单名词或名词短语。通常要将属性名中除第一个词之外的每个词的第一个字母大写，例如**familyName**。

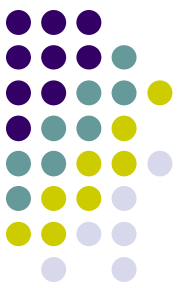


定义类的操作（方法）

- 操作是类的行为特征或动态特征。

- 语法格式：

可见性 操作名（[参数表]）：返回列表[{特征描述}]



(1) 可见性

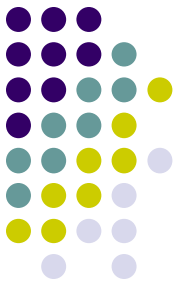
(2) 参数表：用逗号分隔的形式参数序列。

每个参数的语法：参数名：类型[=初值]。

当操作的调用者未提供参数时，该参数就使用默认值。

(3) 返回列表：回送调用对象消息的类型，格式：返回类型或返回值=类型，.....。

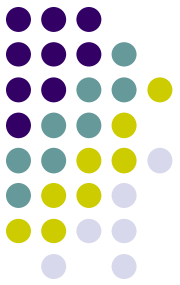
- 返回类型：向调用对象回送一个类型的返回值。
- 返回名=类型，.....：向调用对象回送多个返回类型的值。



(4) [{特征描述}]: 任选项，描述该操作的特征，通常不直接展示在类图中。包括：

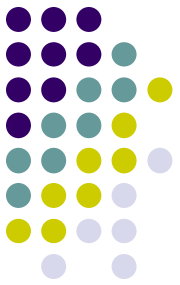
- 前置条件：满足该条件（为真）调用本操作。
- 后置条件：执行本操作后该条件为真。
- 某算法指定执行该操作。
- 用特征（操作名、回送型、参数表）来指定该操作。

(5) 类操作：用下划线标识的操作，在没有对象实例的情况下被调用，只允许访问本类属性，通常把一些通用的操作定义为类操作，如创建对象。


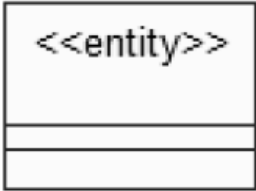

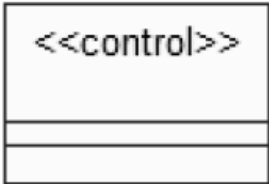
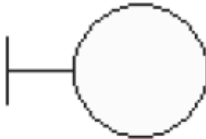
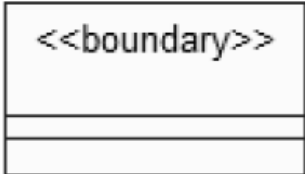


注意点

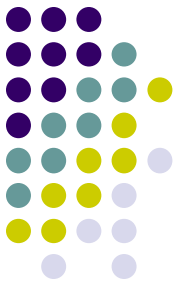
- 定义操作时，操作名（）和返回类型是必须的，其他部分是任选的。
- 操作名是描述它所在类的一些行为的短动词或动词短语。通常要将操作名除第一个词之外的每个词的第一个字母大写，如**move**或**isEmpty**。



类的分类（MVC架构思想）

类型	Icon（图符型）	Label（标签型）
实体类		
控制类		
边界类		

实体类 entity

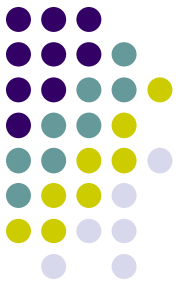


- 实体类是用于对必须存储的信息和相关行为建模的类。它主要是作为数据管理和业务逻辑处理层面上存在的类别。
- 实体类的主要职责是存储和管理系统内部的信息，它也可以有行为。实体对象经常是被动和永久性的。
- 我们经常是从词汇表（在需求阶段制定）和业务领域模型（如果进行了业务建模，则在业务建模阶段中建立）中找寻到实体类的。

控制类 controller

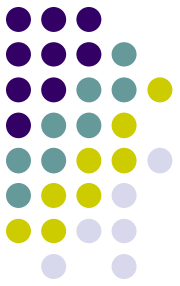


- 控制类用于对一个或几个用例所具有的事件流的控制行为进行建模，控制一个用例中的事件顺序。
- 控制类用于在系统中协调行为。简单情况下，系统可以在没有控制对象的情况下执行某些用例（仅使用实体对象和边界对象）。但是较复杂的用例一般都需要一个或多个控制类来协调系统中其他对象的行为。
- 控制类有效地将边界对象与实体对象分开，让系统更能适应其边界内发生的变更。
- 控制类还将用例所特有的行为与实体对象分开，使实体对象在用例和系统中具有更高的复用性。



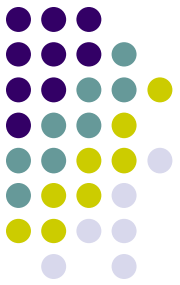
边界类 boundary

- 边界类是对系统外部环境与其内部运作之间的交互进行建模。这种交互包括转换事件，并记录系统表示方式（例如接口）中的变更。边界类描述外部参与者与系统之间的交互，识别边界类可以帮助开发人员识别出用户对界面的需求。
- 边界对象将系统与其外部环境的变更（与其他系统的接口的变更、用户需求的变更等）分隔开，使这些变更不会对系统的其他部分造成影响。
- 一个系统可能会有多种边界类比如：用户界面类、系统接口类、设备接口类等。



5.3 类之间的关系

- 关联
- 聚集
- 继承
- 依赖
-（可扩展）



关联

关联是类之间的语义联系，代表类的对象（实例）之间的一组连接（链）。

- 常规关联

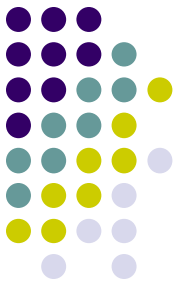
- 在两个类之间有一条直线连接。
- 直线上有关联名
- 重数：数值范围，表示该类有多少个对象可与对方一个对象连接。



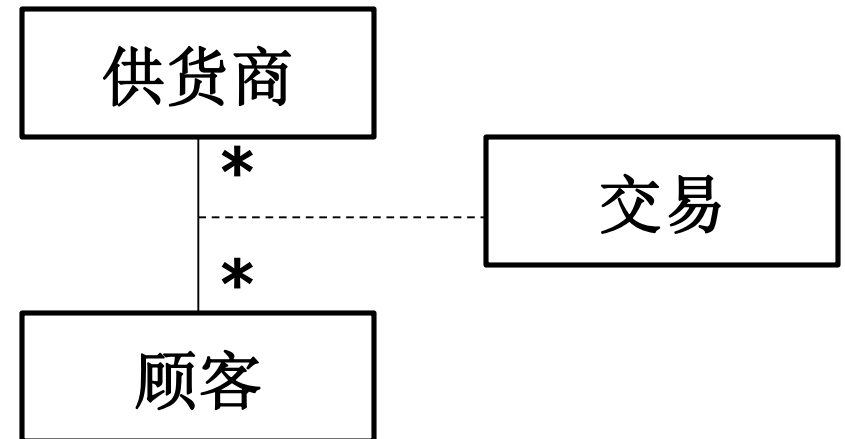
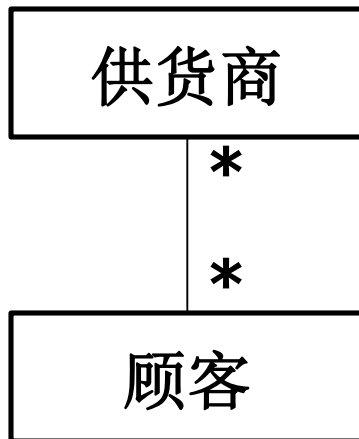
关联类

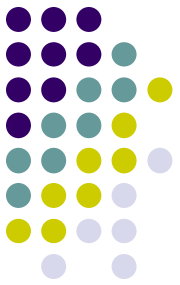


- 在**UML**中把关联定义成类，称为关联类。
- 关联的每个链都是这个关联类的实例
- 关联类也有属性、操作并与其它类关联。



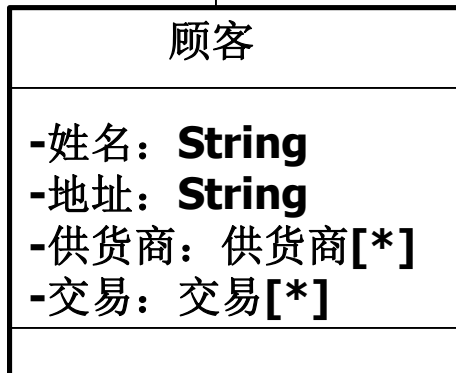
- 关联类的表示





*

*



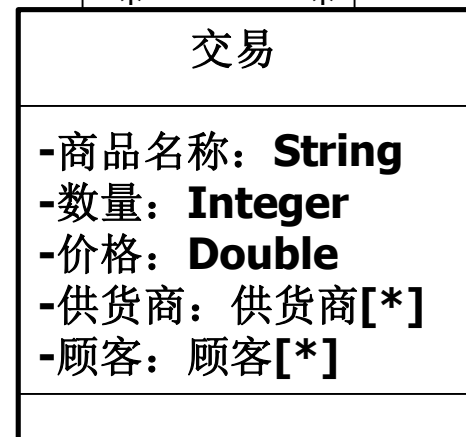
1

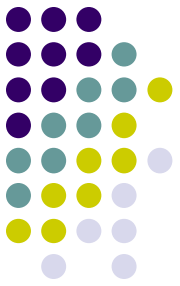
*



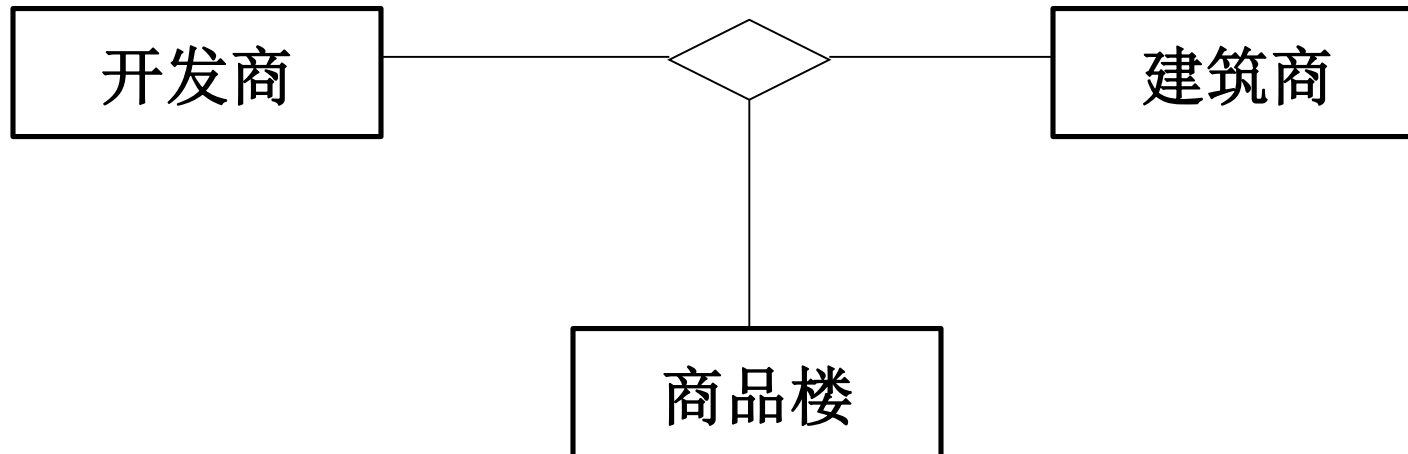
1

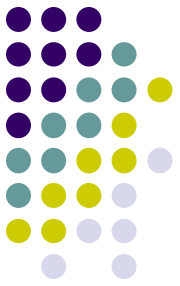
*





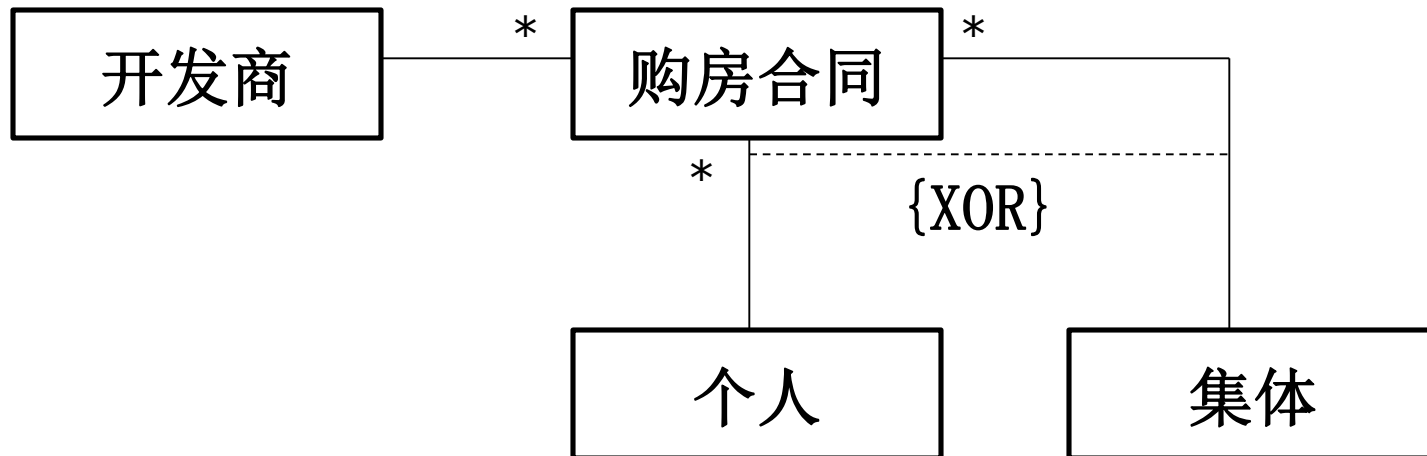
- 多重关联

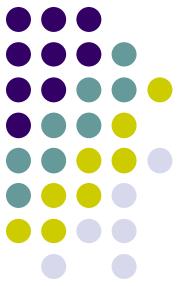




- 关联的约束

UML规定在一些关联上添加约束，约束在{ }内用字符串描述。





关联的约束:

Ordered: 有序的对象

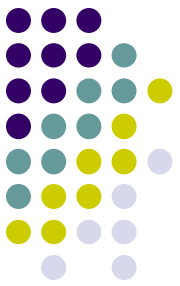
Implicit: 概念性关联

Changeable: 关联对象连接是可变的

Add only: 只能增加

Frozen: 不能改变

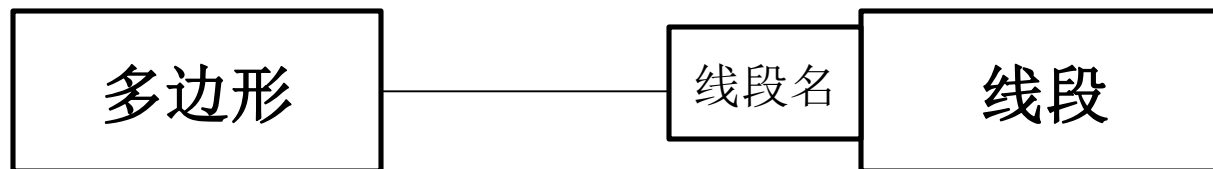
Xor: 互斥的关联



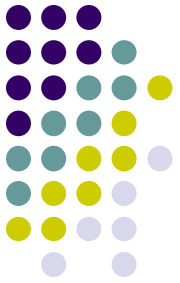
- 受限关联



用于一对多或多对多的关联，限制符号画一个小方快，内标限制内容。

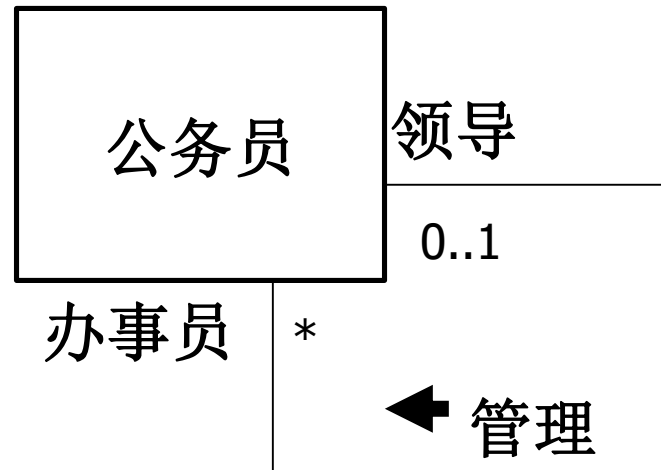


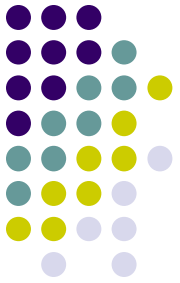
可用于简化一对多关系为一对一



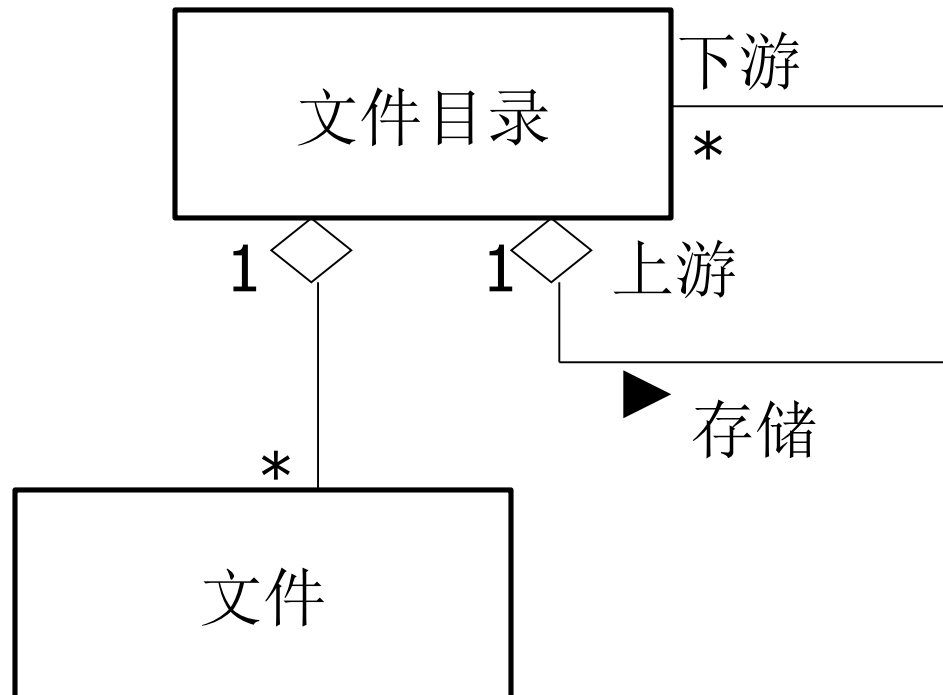
- 递归关联

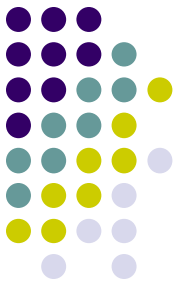
UML允许自己与自己关联。



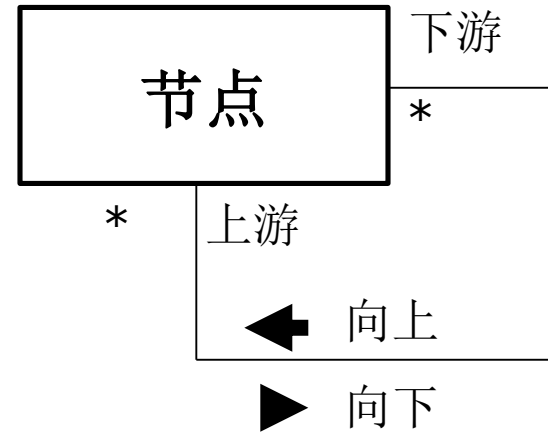
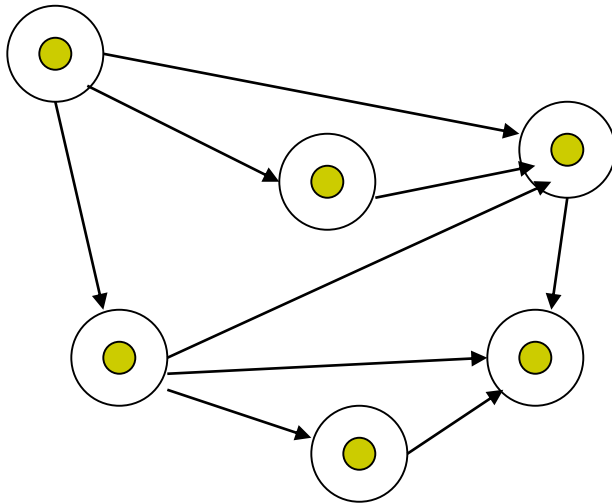


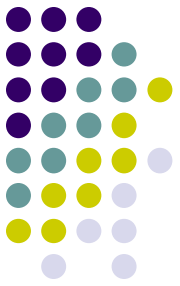
递归关联（自容类）





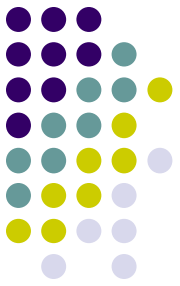
● 节点的网络结构





聚集 (Aggregation)

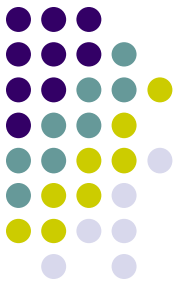
- 聚集是一种特殊的二元关联，它指出类间的“整体-部分”关系。
- 聚集是关联的特例，它可以有重数、角色、限制符号等。



共享聚集（聚集）

- 共享聚集的“部分”对象可以是任意“整体”对象的一部分，表示事物的**整体一部分关系较弱**。

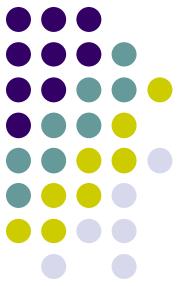




组合聚集（组成）

- 组合聚集中“整体”拥有它的“部分”，它具有强的物主身份，表示事物的**整体一部分关系较强**的情况





● 三种关联的比较

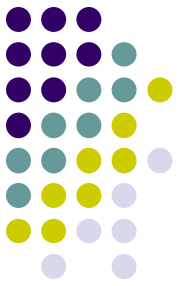
特征	正常关联	共享聚集	组合聚集
UML标记	实线	加空心菱形	加黑色菱形
拥有关系	无	弱	强
多重性	任意	任意	必为1
传递性	无	有	有
传递方向	无	整体到部分	整体到部分

继承



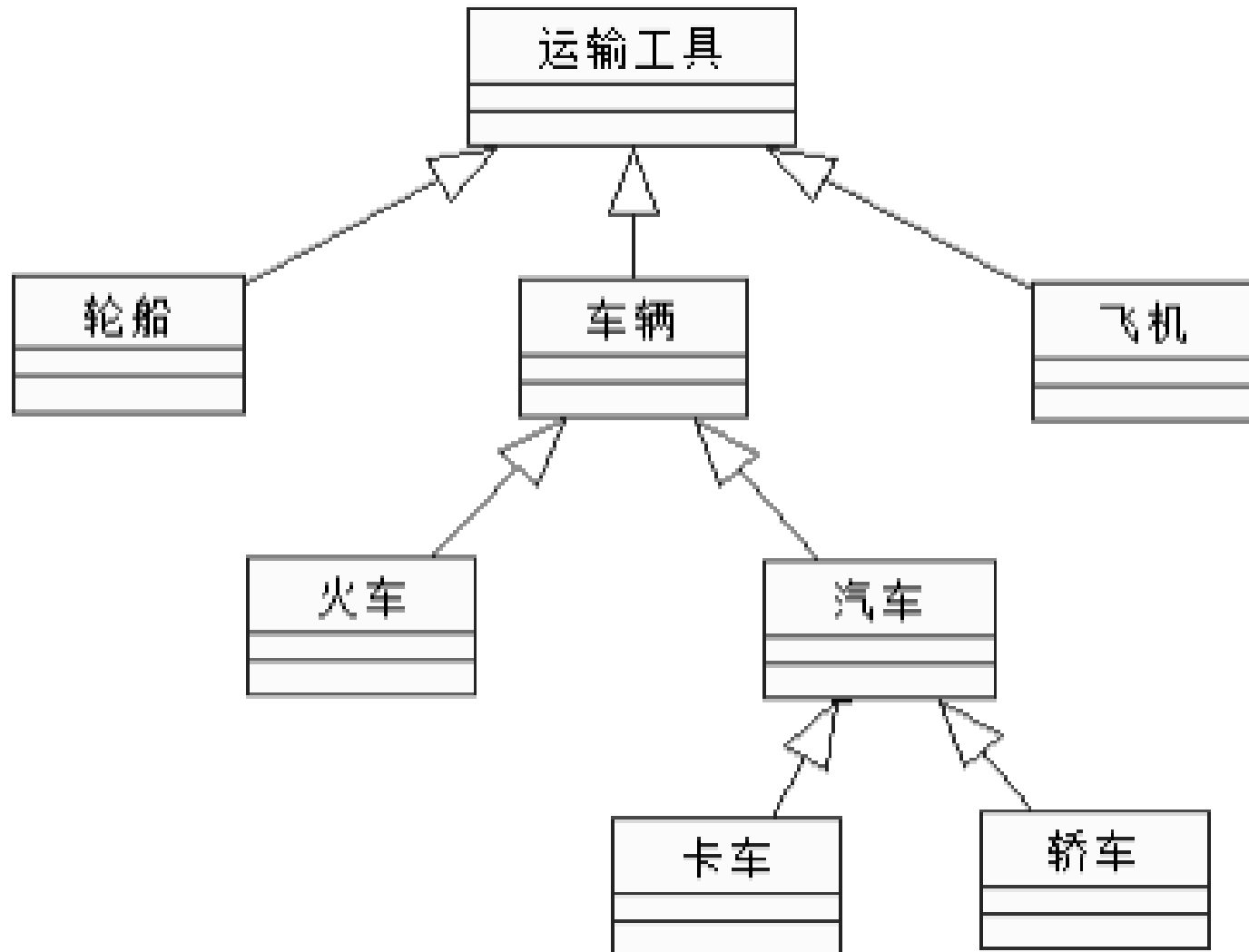
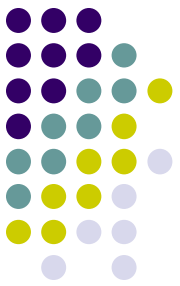
- 继承指出类间的“一般-特殊”
- 将一些类的特殊情况或扩展部分分离出去，把这些公共属性抽象成为一个一般类
- 对一般类，可以扩展一些属性和操作，成为一个特殊类。
- 一个类可以是另一个更一般的类的特殊类，同时它又是另一些更特殊类的一般类，这便形成了类的层次结构。

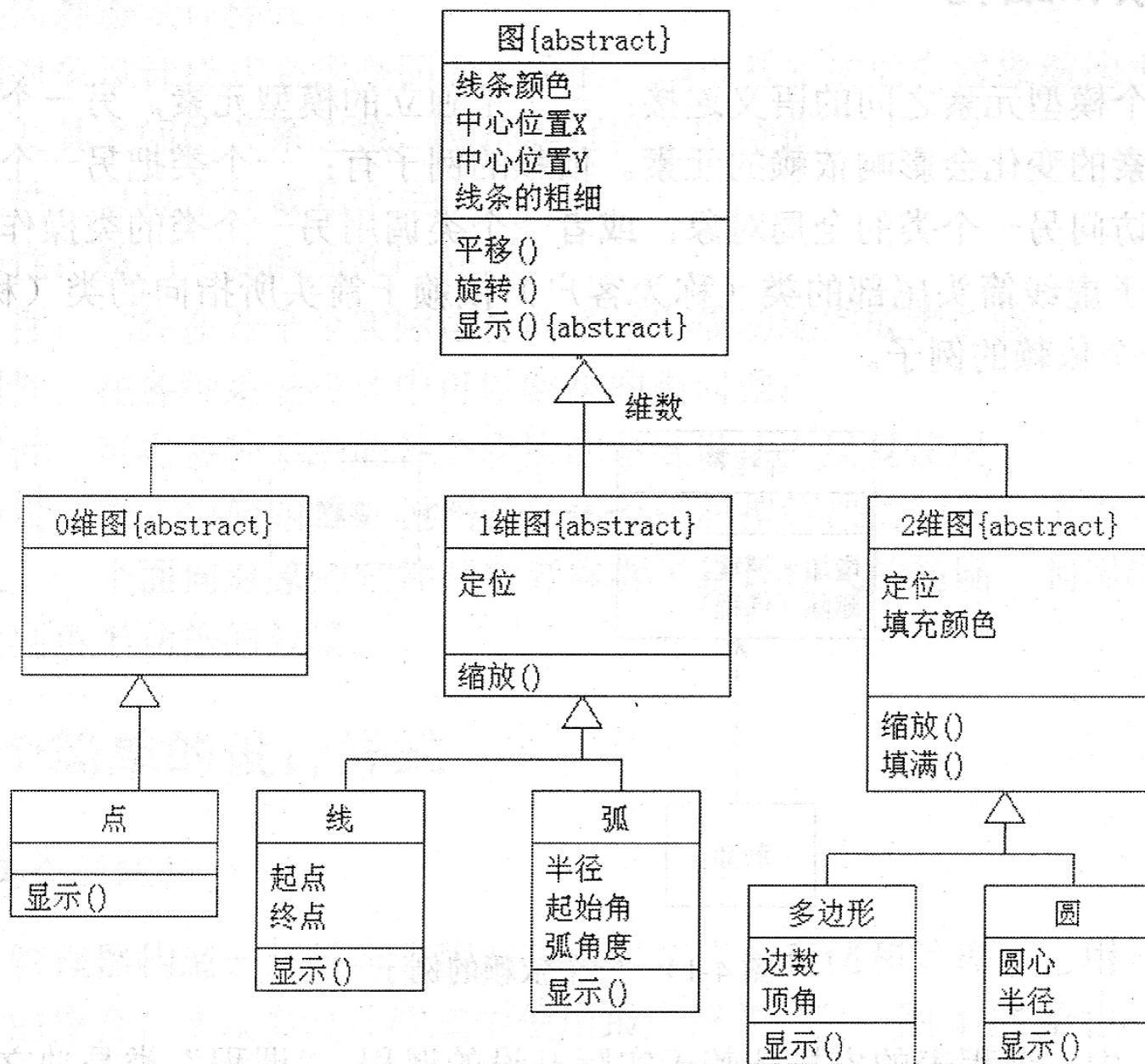
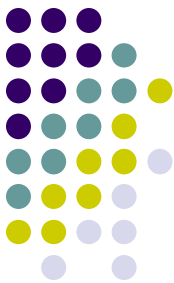
继承



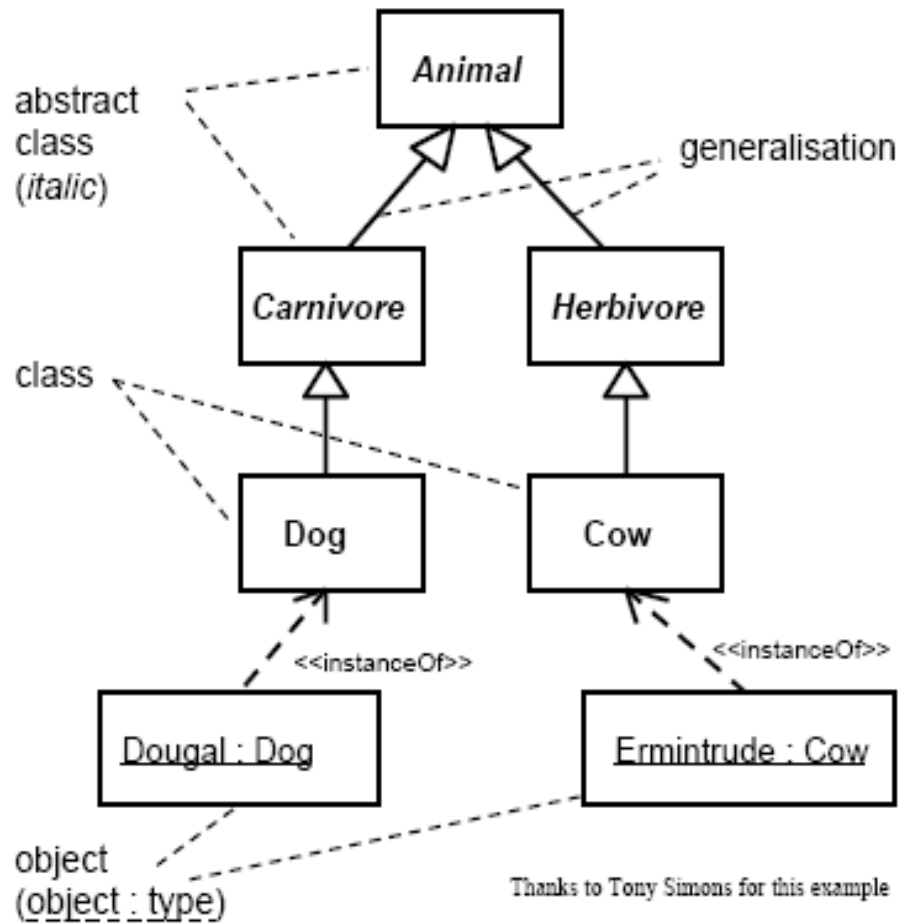
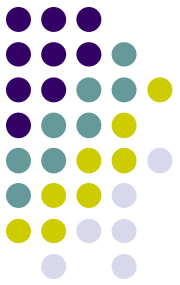
一般类是特殊类的父类，特殊类是一般类的子类

- 子类可以继承父类的属性、操作和关联
- 子类不可访问父类中具有私有可见性的属性和操作
- 父类中具有受保护的属性和操作可被父类及它的子类访问，但不能被其它类访问

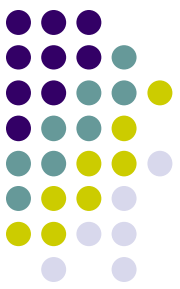




继承举例（3）

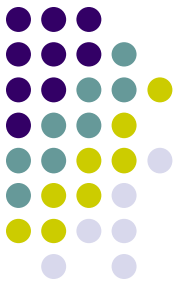


- Dougal is an *instance-of* Dog, which is a *kind-of* carnivore, which is a *kind-of* animal.
- Carnivores and herbivores both possess some of the attributes of animal

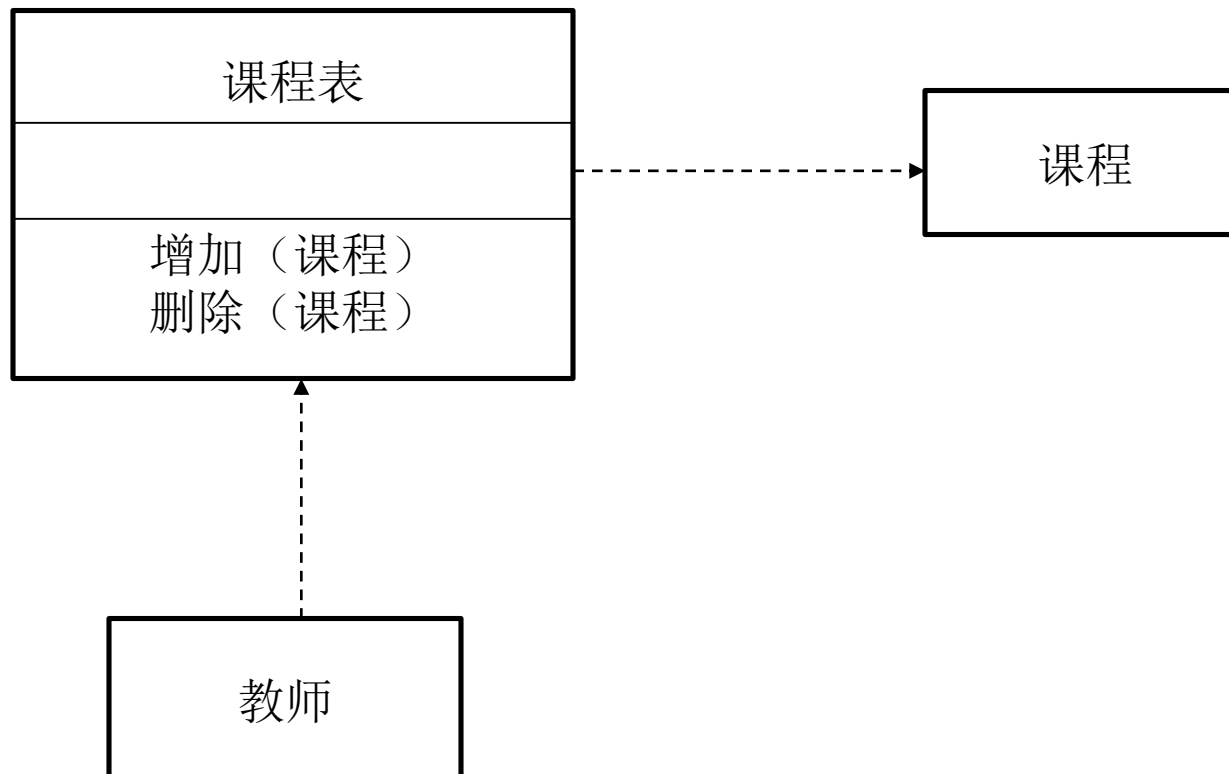


依赖

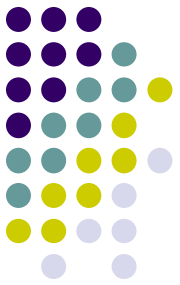
- 依赖是两个模型元素之间的语义连接：一个是独立的模型元素；一个是依赖的模型元素
- 一个类把另一个类的对象作为参数
- 一个类访问另一个类的全局对象
- 一个类调用另一个类的类操作



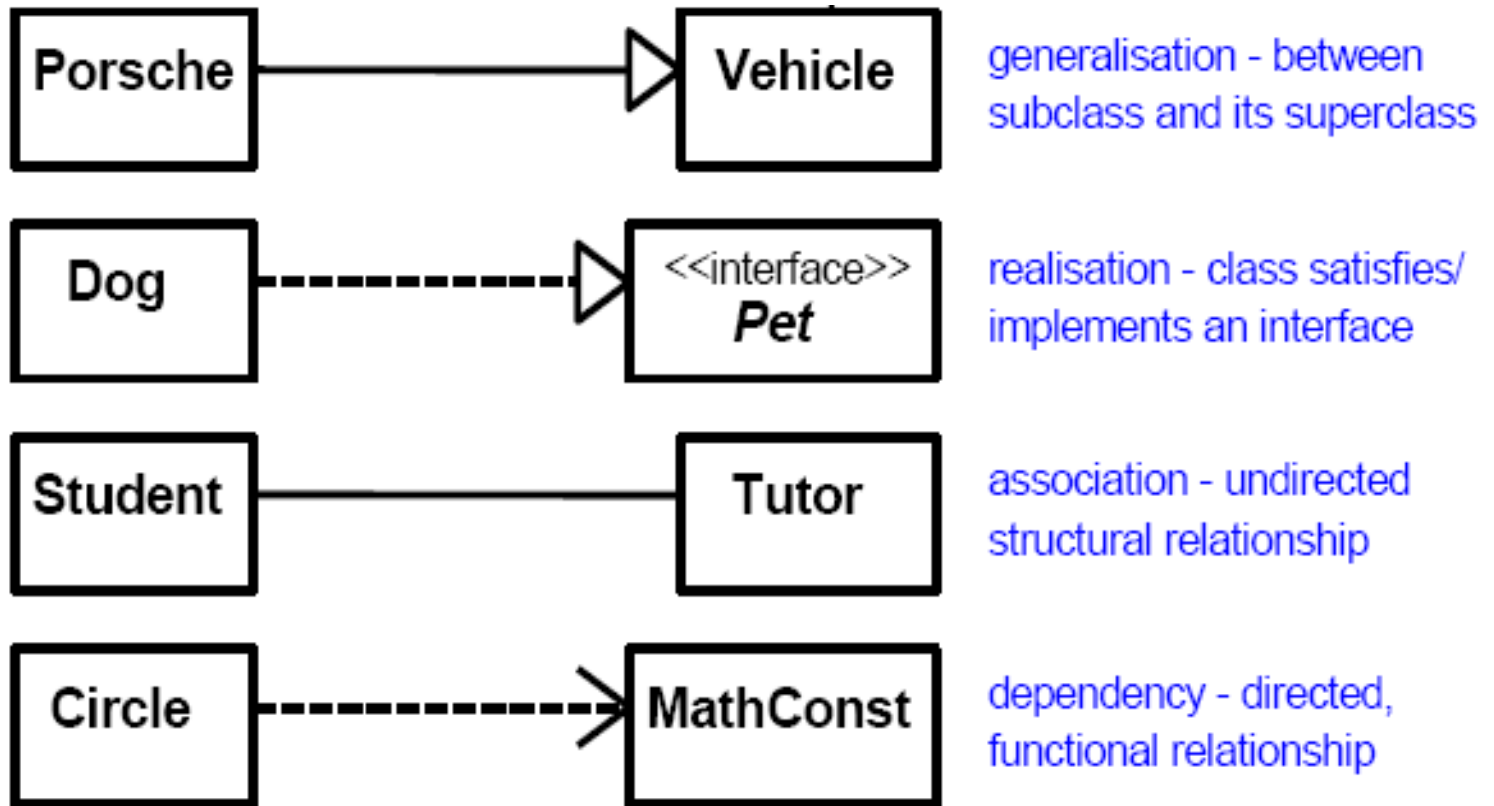
- 图形上，把依赖画成一条有向的虚线，指向被依赖的事物。
- 课程表依赖课程，因为类“课程表”使用类“课程”作为其操作的参数类型。这意味着如果课程发生变化，则会影响到课程计划。在执行操作时，类“课程表”的对象与“课程”对象建立联系。

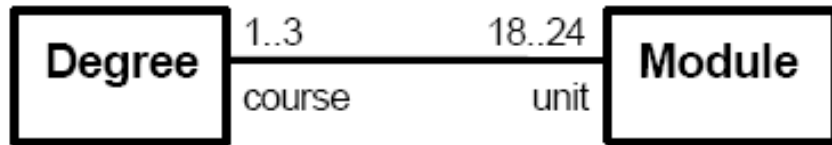
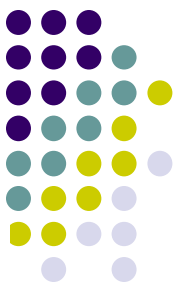


更多类的关系举例:

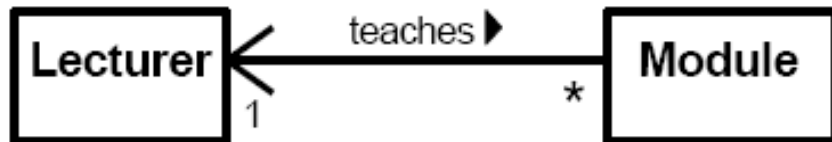


Four main structural relationships:

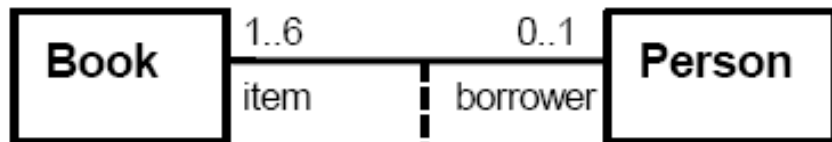




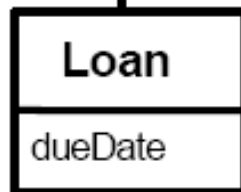
association with end roles and multiplicities



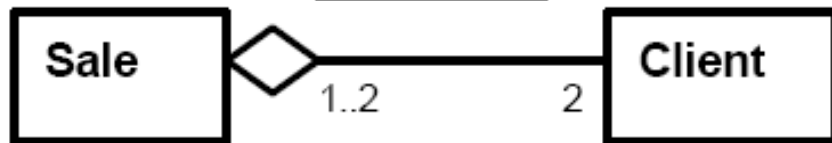
named association, also with navigable direction



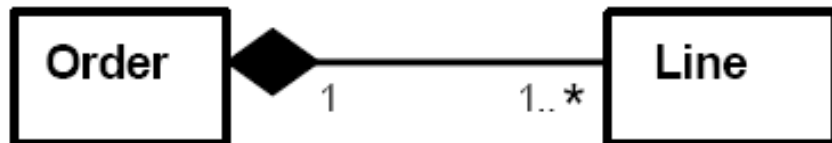
association class - stores attributes of association



Composition is a stronger version of aggregation. In composition each part lives and dies with the whole – cascading deletion.



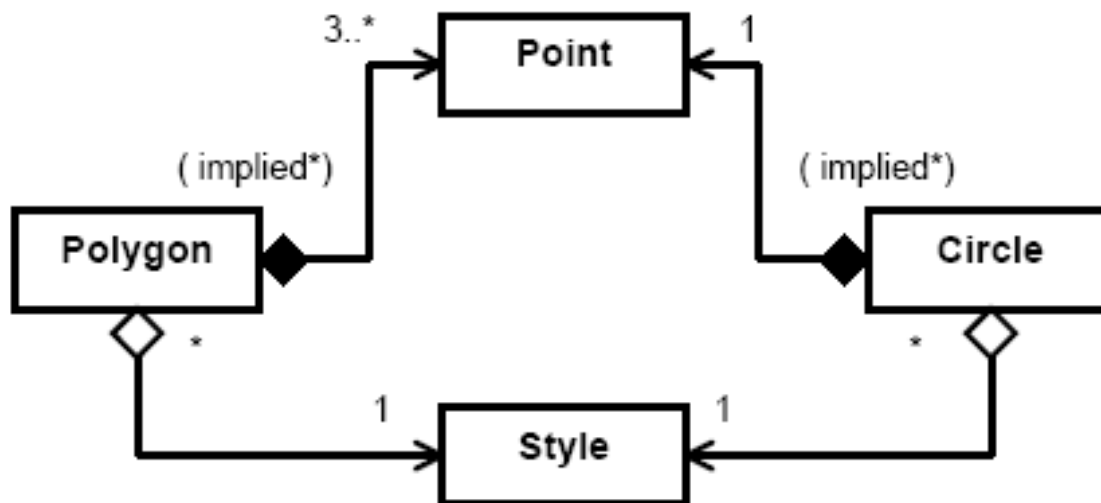
aggregation - whole/part structural association



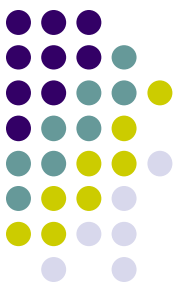
composition - whole/part exclusive association



综合举例

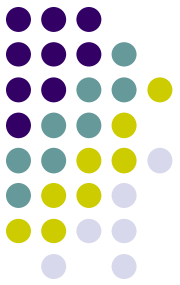


- Any instance of Point may be part of a Polygon or a Circle, but not both.
- A Style instance may be shared by many Polygons or Circles.
- Deleting a Polygon implies deletion of associated Points but not Styles.
- The composition constraint cannot be expressed by multiplicities alone.



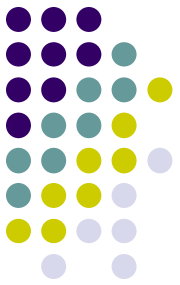
设计样式简述

- 在软件开发项目的系统设计中，总有一些只包含少数几个对象的设计模块在设计图中反复出现在各种更大更复杂的系统设计中，我们把这些模块称为“设计样式”。
- 一个设计样式有广为人知的外在表现，可以反复使用以解决同类问题。
- 在系统分析设计中不是采用单个类，而是有效反复采用样式进行模型的设计，高效率、高质量地进行软件开发。



5.4 对象图

- 对象是类的一个具体实例，而对象图（**Object Diagram**），也常被认为是类图的一个具体实例。
- 类只是一个抽象的概念，而对象是具体的，并且随着系统动态行为而不断变化的。因此，对象图反映的只能是具体某一个时间点一组对象的状态和它们之间的关系。
- 对象图是动态建模的一个基础。



对象图中可以包含类

对象图一般包括：

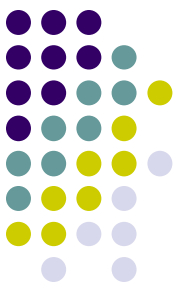
- 对象
- 链（关系）

像所有的其他图一样，对象图可包含注解和约束。有时也要把类放在对象图中，特别是要把各实例背后的类可视化时，就更是如此。



对象图和类图的比较

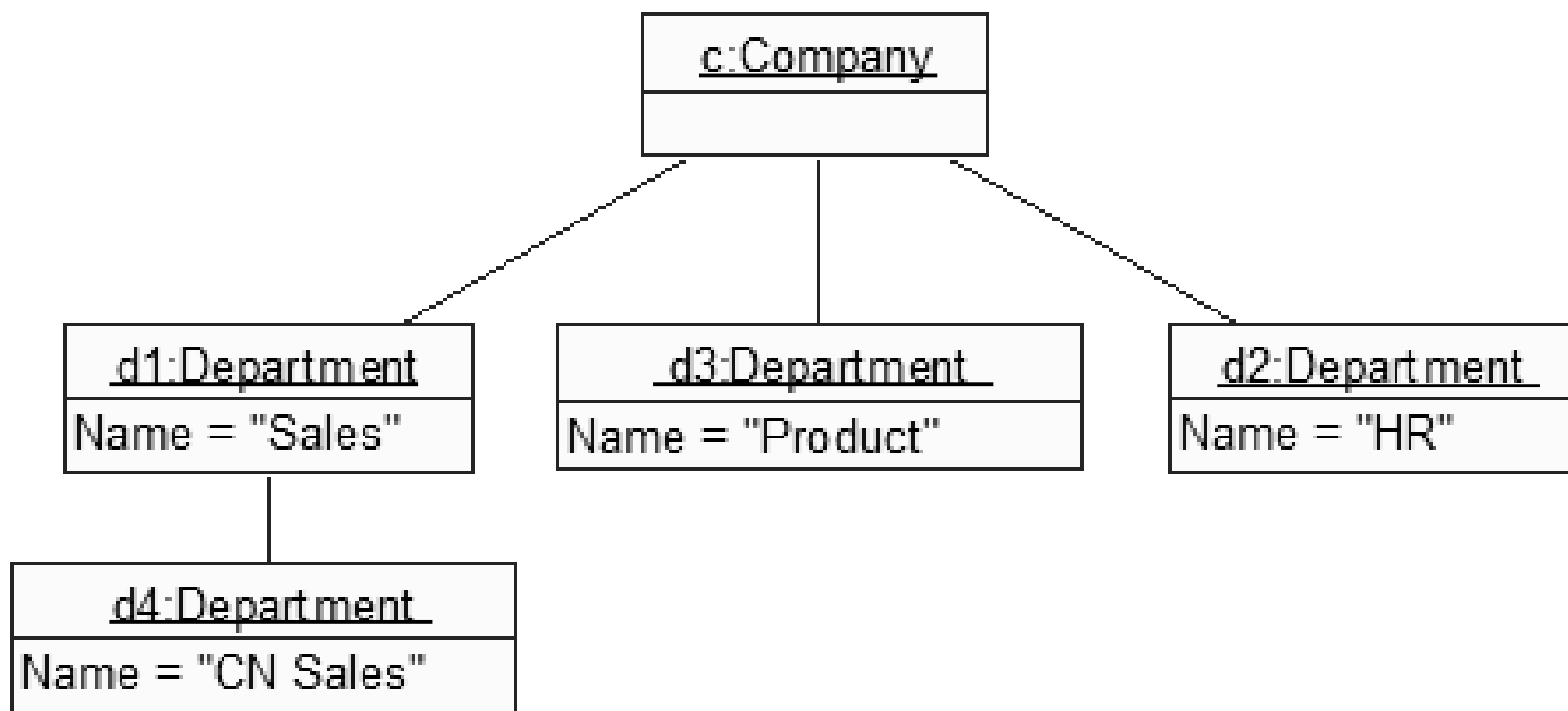
类图	对象图
类具有三个分栏：名称、属性和操作。	对象要定义的有两个分栏：名称和属性。操作与类完全一样。
在类的名称分栏中只有类名（可带路径名）。类名用英文表示时通常首字母大写。	对象的名称形式为“对象名：类名”，匿名对象的名称形式为“：类名”。对象名用英文表示时首字母小写。需要与类名区分时，对象名加下划线。
类的属性分栏定义了所有属性的特征。	对象必须定义属性的当前值，以便用于测试用例或例子中。
类使用关联连接，关联使用名称、角色、多重性及约束等特征定义。必须说明可参与关联的对象数目。	对象使用链连接，链拥有名称、角色，但是没有多重性。对象代表单独的实体，所有的链都是一对一的。

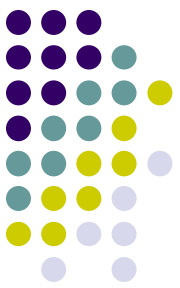


对象图的建模过程

- (1) 确定参与交互的各对象的类，可以参照相应的类图和交互图；
- (2) 确定类和类间的关系，如依赖、泛化、关联和实现；
- (3) 针对交互在某特定时刻各对象的状态，使用对象图为此些对象建模；
- (4) 建模时，系统分析师要根据建模的目标，绘制对象的关键状态和关键对象之间的连接关系。

举例





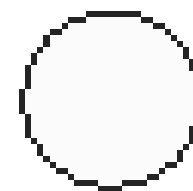
5.5 接口

- 对于一个经常被其他类访问的类，为了保证系统的安全性、通用性和限制软件构件的对外运算，一般不允许其他类直接与其通信，通常对外只开设一个窗口供其他类进行间接访问。
- 接口也可以认为是一个类，定义了一组提供给外界的操作。接口是一个特殊的抽象类，没有属性
- 接口不仅对分离类或构件的规约和实现是重要的，而且当系统较大时，还可以用接口详述包或子系统的外部视图。

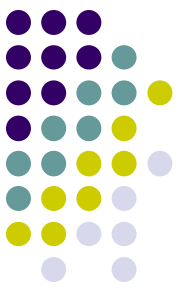
UML中接口的表示方法



- 接口是一组操作的集合，是在没有给出对象的实现和状态的情况下对对象行为的描述。接口包含操作但是不包含属性，且没有对外界可见的关联。一个类可实现一个或多个接口，一个或多个类也可使用一个或多个接口。
- 在**UML**中，接口有两种表示法：标签型和图符型

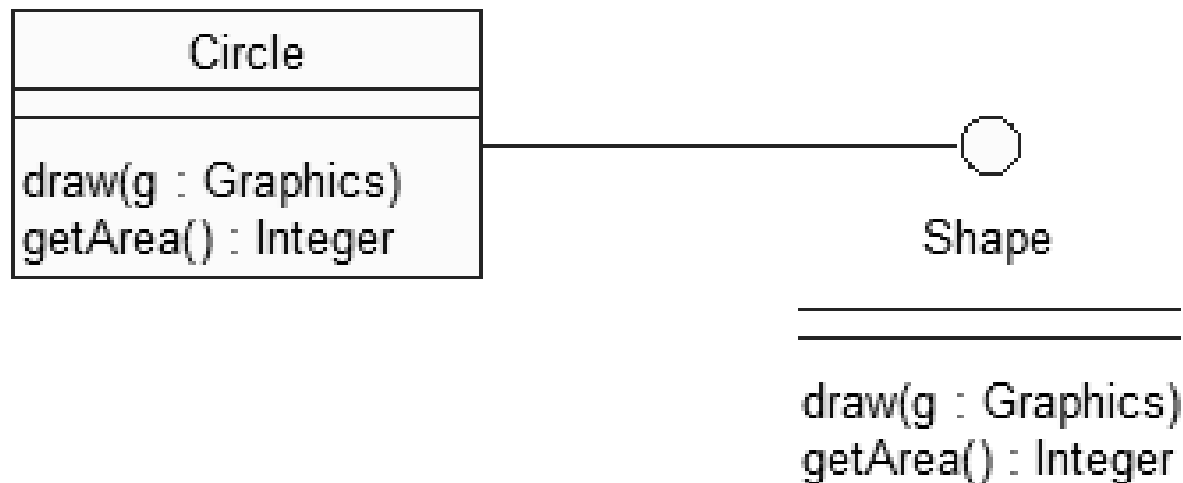
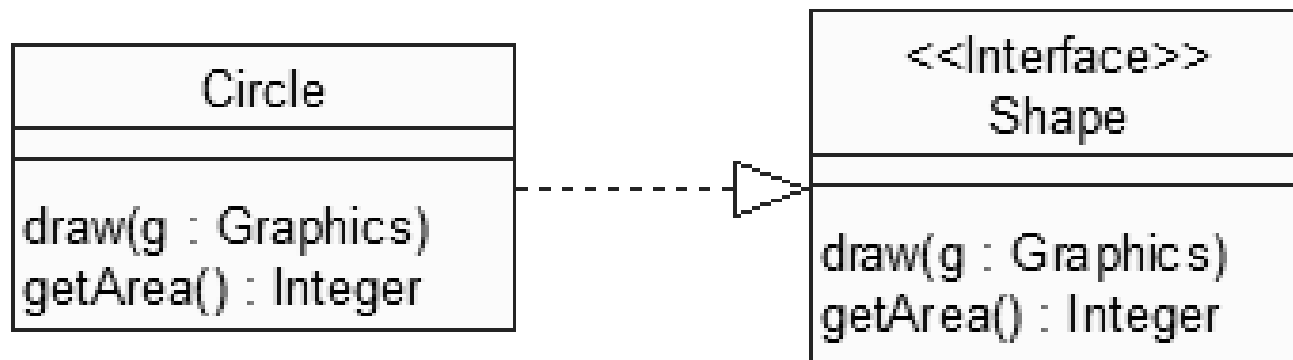
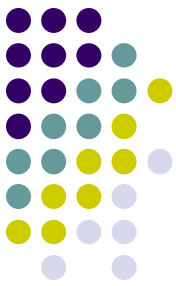


接口2



接口的关系

- 和类一样，接口也可以参与泛化、关联和依赖关系。此外，接口也可以参与实现关系。
- 实现关系在类图中就是接口和类的关系，一般是指一个类实现了一个接口定义的方法。实现是两个类之间的语义关系，其中一个类描述了另外一个类保证实现的合约。
- **UML**实现关系有两种表示方法，一种是在类图中使用空心三角形的虚线箭头表示，虚线段一端连接实现类，箭头从实现类指向接口。另一种表示方法为，用圆圈表示接口，用一条实线将其与一个类连接。



示例



选课

浏览所有课程()

注册()

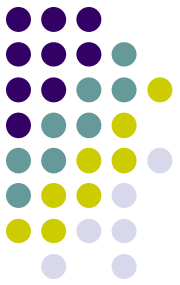
取消注册()

implement

选课实施

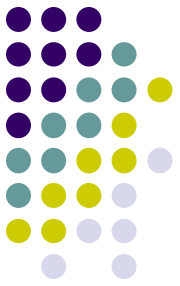
课程名 : String
课程编号 : String
学号 : String
姓名 : String
学分 : int

选课()



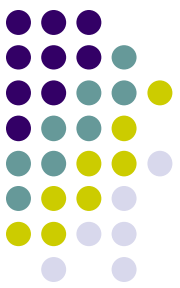
理解接口

- 接口是一组操作，描述了类（或构件）的服务。更深些，会看到这些操作的全部特征标记，连同它们的各项具体特性，如可见性、范围和并发语义等。
- 但是对于复杂接口来讲，这些特性还不足以帮助理解所描述的服务的语义。



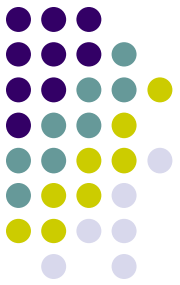
在**UML**中，为了使接口易于理解和处理，可以为接口提供更多的信息：

- 可以为各个操作附上前置和后置条件。通过这样做，就能理解接口做什么以及如何使用它，而不必深究其实现。
- 其次，给接口附上一个状态机。用状态机详述接口操作的合法的局部命令。
- 最后，可以为接口附上协作。通过一系列的交互图，可以用协作详述接口的预期行为。



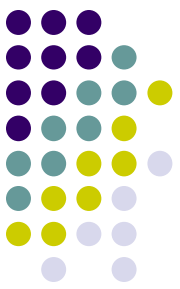
5.6 包与子系统

- 一个复杂结构有很多个类图，类图之间又有很多关联，形成一个复杂的网络。为了清晰、简洁地描述一个复杂的系统，通常把它们分解成若干较小的系统（子系统）。
- 在**UML**中使用了“包”的机制，一个包相当于一个子系统。



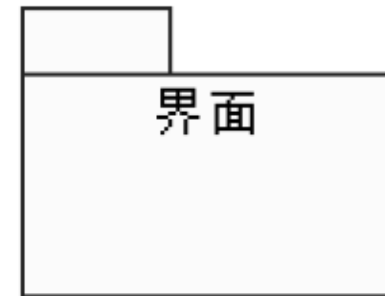
包的定义

- 包是**UML**的模型元素之一，包可以包含其他包和类。包之间可以有各种关系，如依赖等。
- 包是一种分组机制。
- 包的实例是没有意义的。因此包仅建模时有用，而不需要转换成可执行的系统。

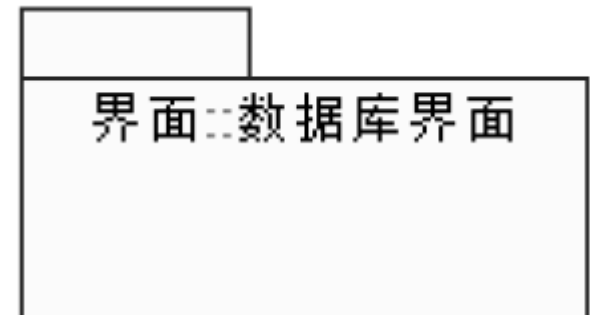


包的图形表示

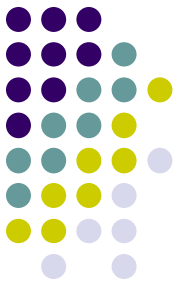
- **UML**中把包画成带标签的文件夹。
- 包的名字放在文件夹中（如果没有展示它的内容）或放在标签上（如果在文件夹里展示内容）。
- 每个包都必须有一个有别于其他包的名称。名称是一个文字串。单独的名称叫做简单名，而限定名是以包所位于的外围包的名称作为前缀的包名，用双冒号（::）分隔包名，也称带路径包名。



a.简单包名

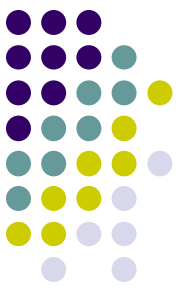


b.带路径包名



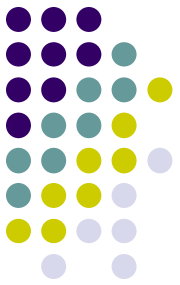
包的构造型

- UML中对包预定义的构造型有：
 - <<system>>: 系统模型
 - <<subsystem>>: 子系统模型
 - <<facade>>: 仅仅是其他包的某一个视图，如业务模型中有关产品的信息
 - <<stub>>: 另一个包公共内容的服务代理
 - <<framework>>: 模型的体系结构



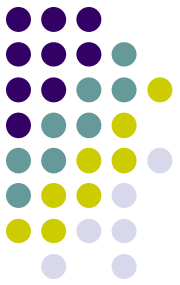
包的元素

- 包里面可以“拥有”其他元素，**这些元素可以是类、接口、构件、结点、协作、用例和图，也可以是其他包。**
- 这种“拥有”就是一种组成关系。
- 包形成了一个命名空间，这意味着在**一个包的语境中同一种元素的名称必须是唯一的**。不同种类的元素可以有相同的名称。可能的话，在不同的包中也应避免重复的名字，以避免造成混乱。
- 拥有关系的语义使包成为一种按规模来处理问题的重要机制。没有包，最后将得到一个庞大的、平铺的模型，其中的所有元素的名称都要唯一，这种情况很难管理。

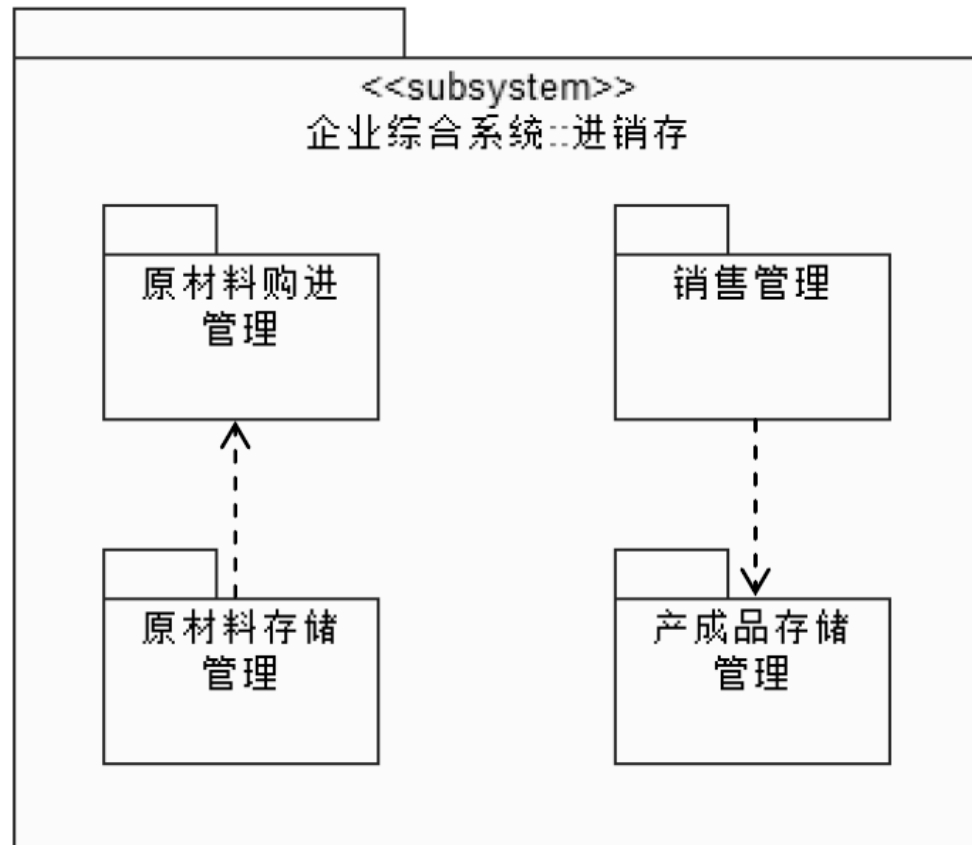


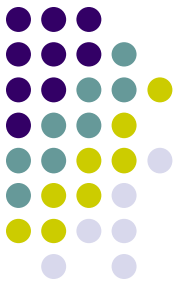
包的嵌套

- 如果包的图符中又含有包，称为包的嵌套。
- **UML**允许包的嵌套。包的嵌套有两种表示方法：内嵌式表示法和树形层次结构表示法。
- 在实际使用中，最好避免过深地嵌套包，两、三层的嵌套差不多是可管理的极限。

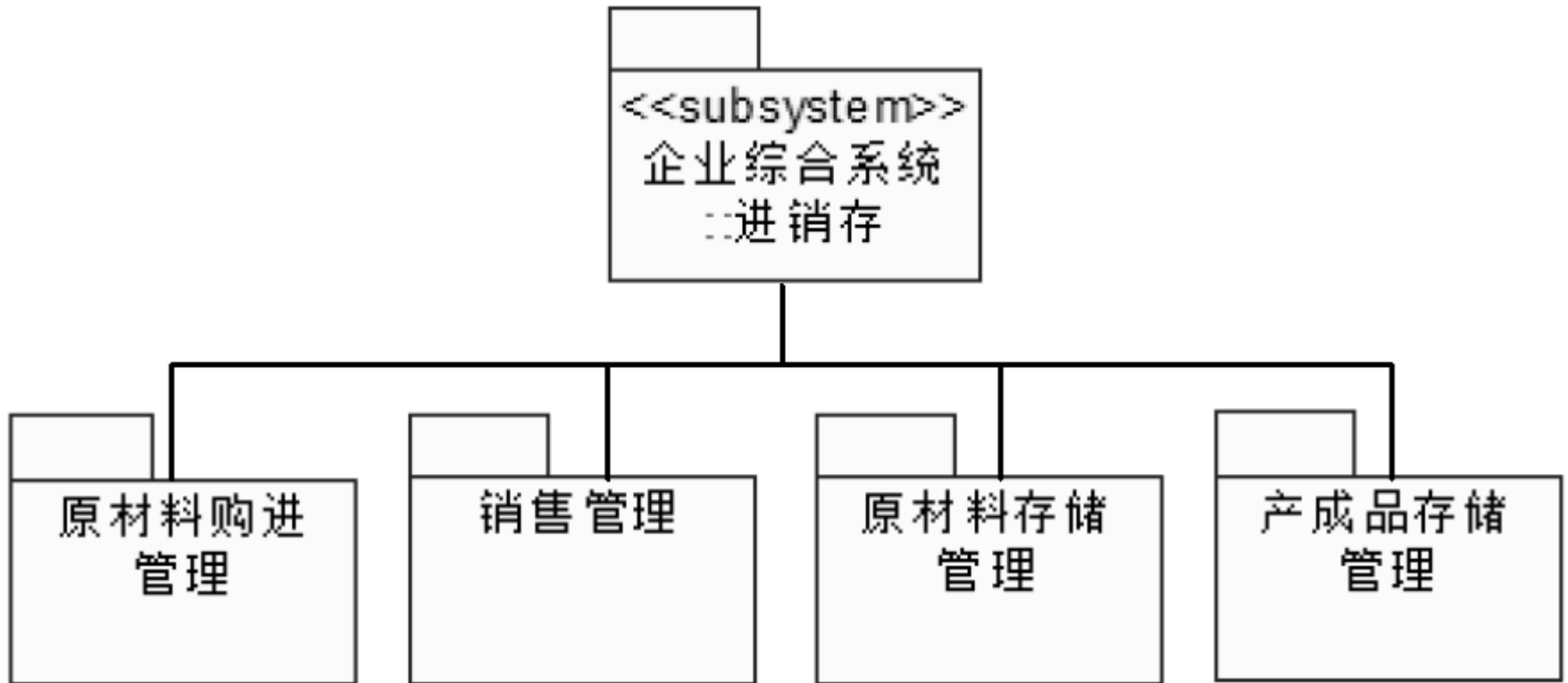


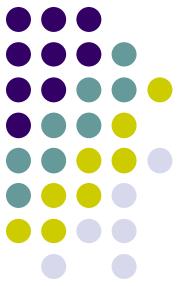
内嵌式表示法





树型层次结构

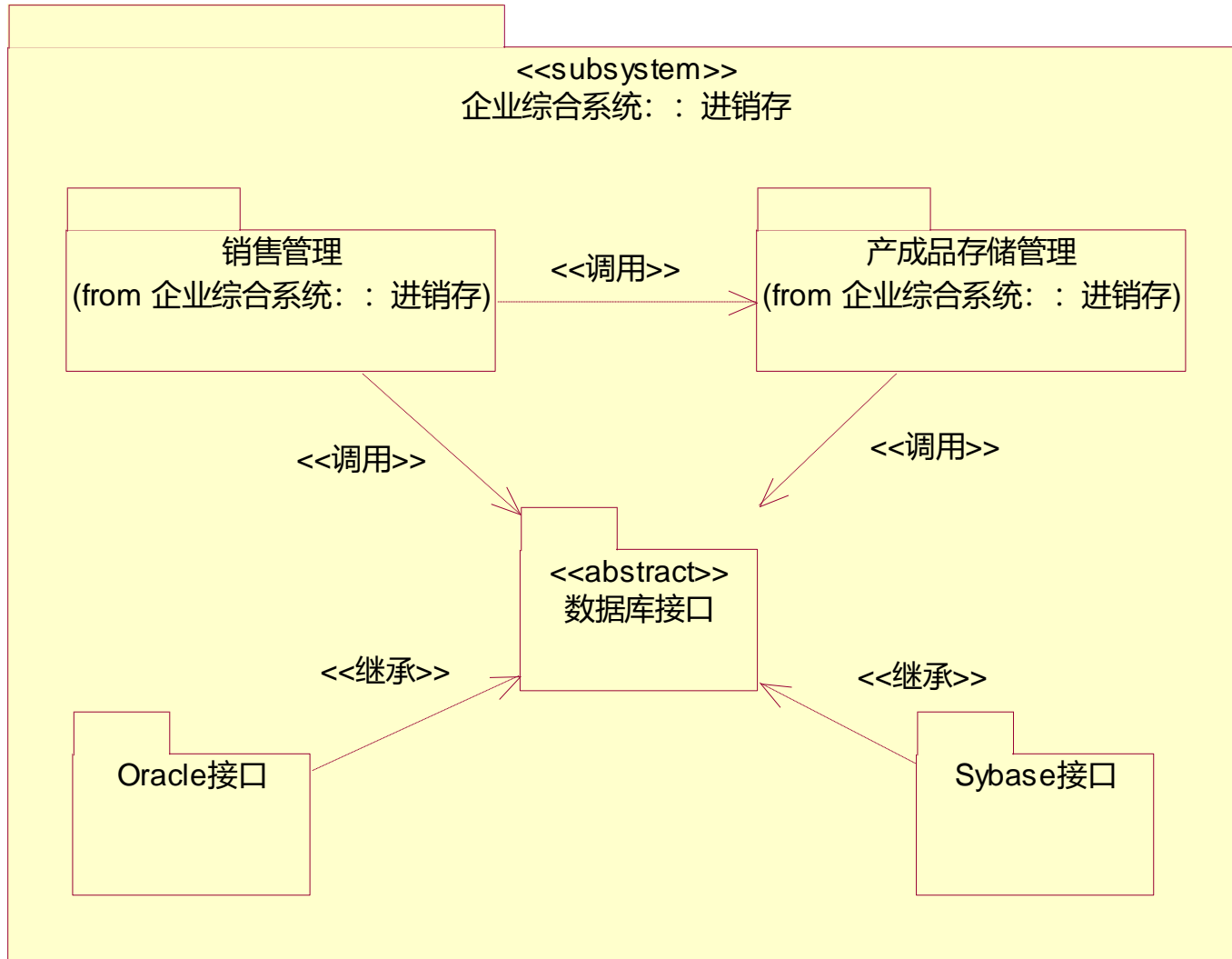




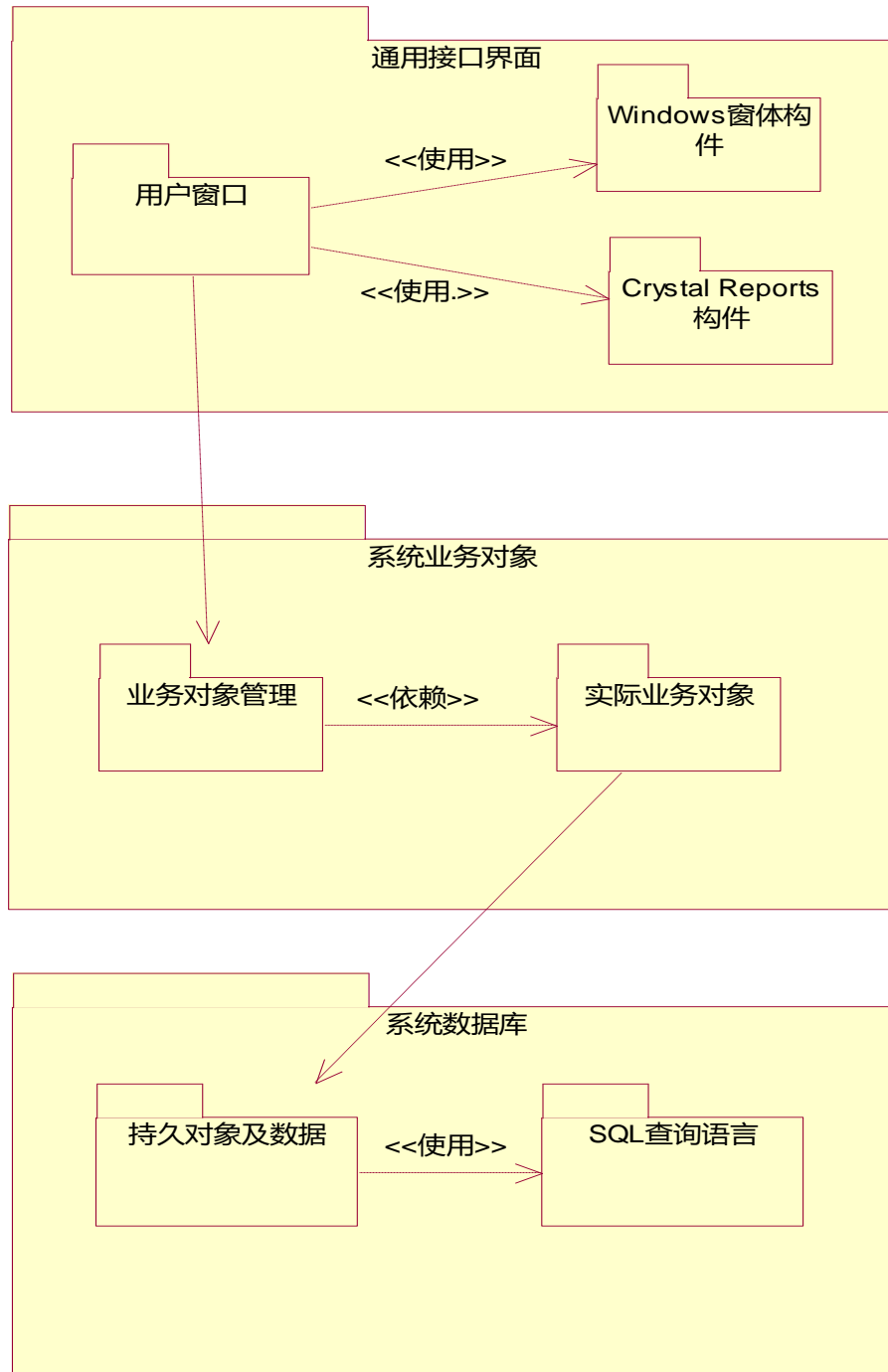
包之间的关系

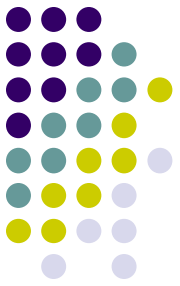
- 依赖：两个包之中的元素（例如类）之间有依赖，则两个包有依赖
- 继承：包之间的继承关系描述了系统的接口

举例1



举例2





Additional suggestions

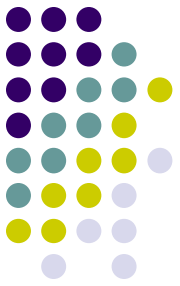
类图几乎是所有OO方法的支柱。采用标准建模语言**UML**进行建模时，必须对**UML**类图引入的各种要素有清晰的理解。以下对使用类图进行建模提出几点建议：

- 不要试图使用所有的符号。从简单的开始，例如，类、关联、属性和继承等概念。在**UML**中，有些符号仅用于特殊的场合和方法中，只有当需要时才去使用。

more



- 根据项目开发的不同阶段，用正确的观点来画类图。如果处于分析阶段，应画概念层类图;当开始着手软件设计时，应画说明层类图;当考察某个特定的实现技术时，则应画实现层类图。
- 不要为每个事物都画一个模型，应该把精力放在关键的领域。最好只画几张较为关键的图，经常使用并不断更新修改。使用类图的最大危险是过早地陷入实现细节。为了避免这一危险，应该将重点放在概念层和说明层。如果已经遇到了一些麻烦，可以从以下几个方面来反思你的模型。

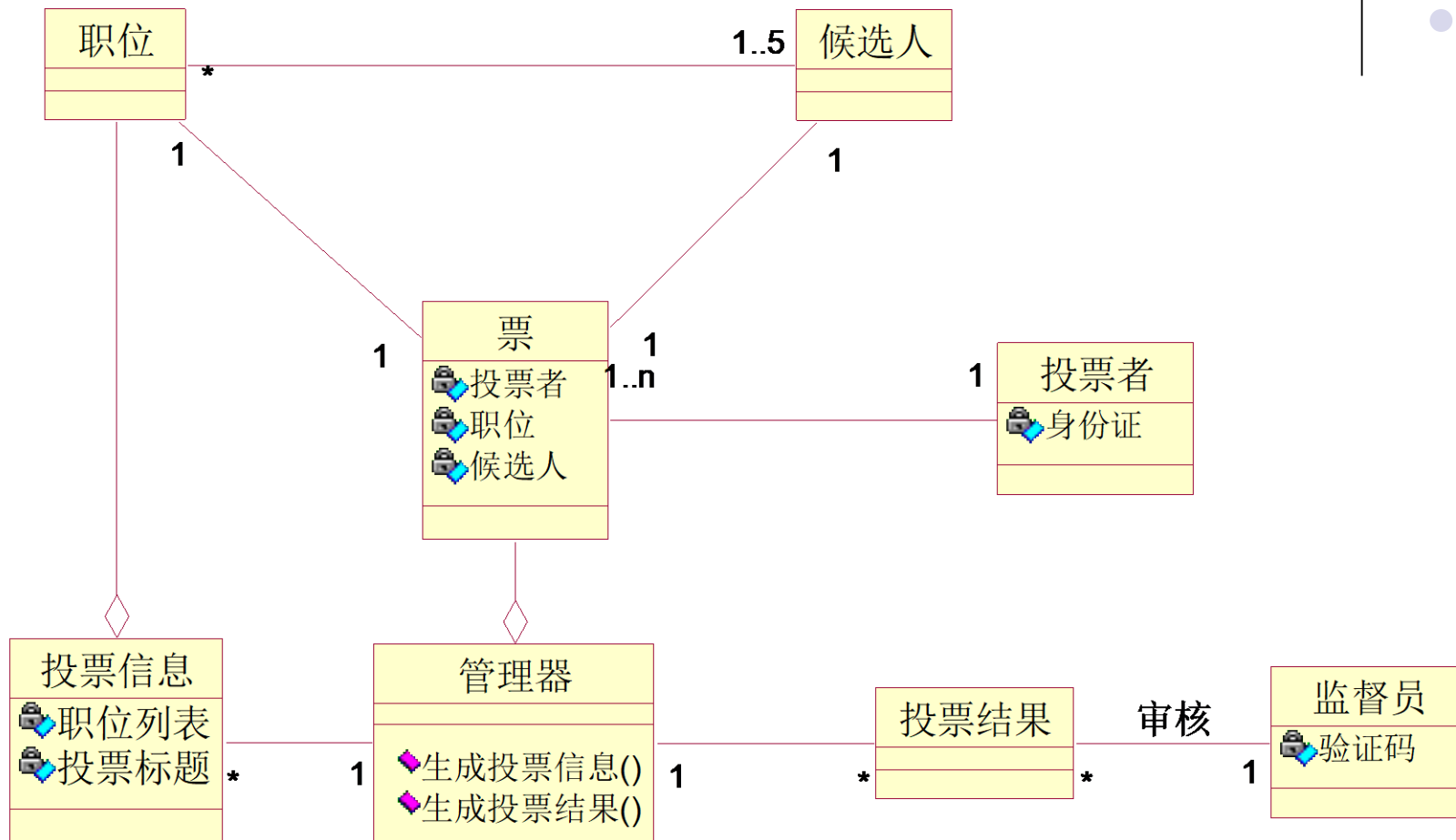


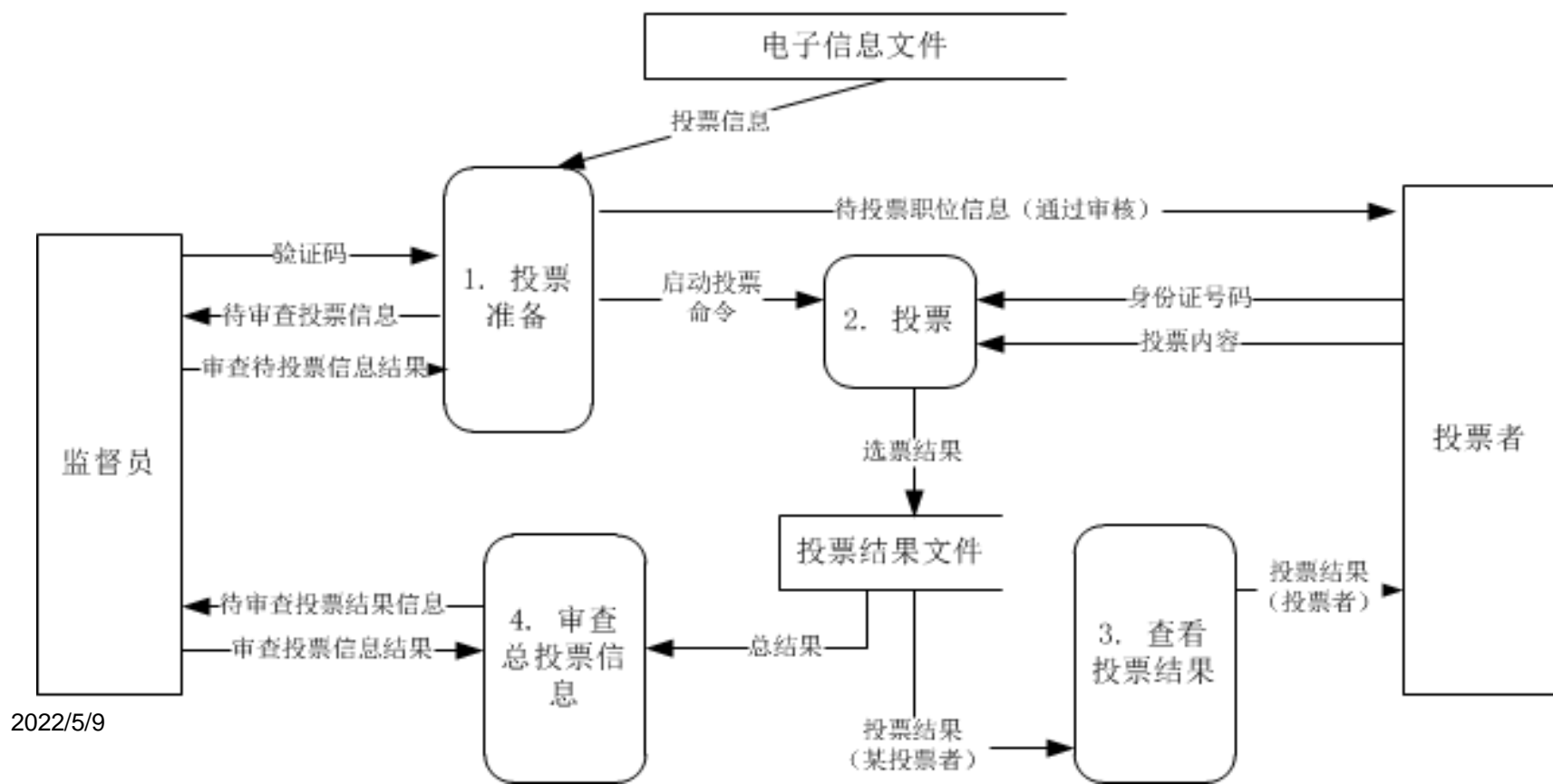
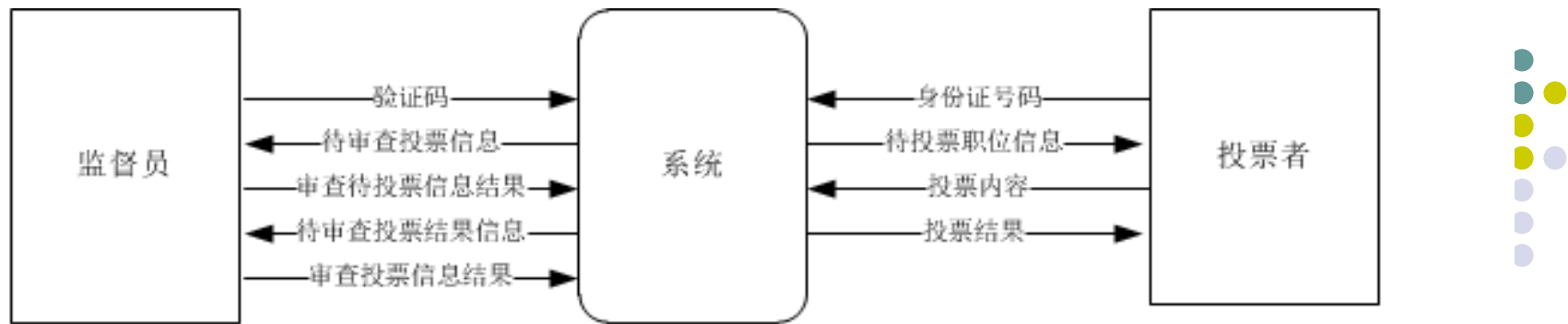
Continued...

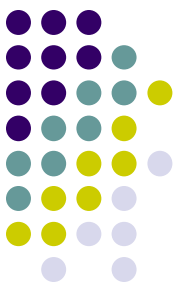
- 模型是否真实地反映了研究领域的实际。
- 模型和模型中的元素是否有清楚的目的和职责 (在面向对象方法中，系统功能最终是分配到每个类的操作上实现的，这个机制叫职责分配)。
- 模型和模型元素的大小是否适中。过于复杂的模型和模型元素是很难生存的，应将其分解成几个相互合作的部分。



案例分析（电子投票系统-类图）







- 1. 产生软件危机的主要原因是什么？软件工程的核心思想是什么？（10分）
- 2. 根据你的理解，谈谈结构化方法和面向对象方法的差别。（10分）
- 3. UML里面定义了哪些扩展机制，试举例说明。（10分）
- 4. RUP的特点是什么？其组织形式是怎样的（分为哪几个阶段，有哪些工作流）？（10分）
- 5. 什么是UML的视图，其作用是什么？请画出UML的视图结构模型。（10分）