UML 统一建模语言

主讲：穆海伦

杭州电子科技大学 计算机软件教研室

E-mail：helen_uml@163.com

电话：13750802617（612617）

QQ：1055874556

2015 年 9 月

目录

第一章　概述

内容概要

- ✧ 软件工程概述
- ✧ 面向对象分析与设计
- ✧ 模型
- ✧ UML 简介

§1.1 软件工程概述

一. 软件危机：

1. 软件失效的表现：
   1) 开发出没有功能的软件。
   2) 没有充分的针对某些领域专家或用户的需要。
   3) 软件表面上满足了需要，但底层运算可能不正确。
   4) 由于用户的错误产生故障。
   5) 由于响应时间太慢而失去实用价值。
2. 软件开发项目失败的原因：*复杂性*
   1) 应用领域复杂：在某些应用领域，软件的操作依赖于专业知识。
   2) 口头、书面语言中所固有的含糊性为领域专家与开发软件的技术人员之间的沟通又增添了一层复杂性：
      - ■ 不同的背景知识和不同的专业术语使人与人之间的有效交流变得困难。
      - ■ 自然语言的含糊性。
   3) 将大项目分割成几个小部分，每个小部分由不同的个人来开发，并且保证这些部分可以在一起工作，这样的过程已成为软件开发过程复杂性的另一个来源。
   4) 深刻领会大的开发项目的细节比较困难。
3. 解决办法：组织
   1) “软件危机”的出现使得人们开始对软件开发的方法进行重新审视。人们意识到，优秀的程序除了功能正确、性能优良以外，还应该易读、易用、易维护。而早期所谓的优秀程序常常通篇充满

了程序员的编程技巧，很难被别人看懂。

2) 通过组织，我们能够克服软件开发过程中的复杂性。

3) 软件工程的基本课题就是控制开发过程并生产出结构良好的、准确的软件解决方案。

4) 用于开发软件和进行有组织地开发的各种技术确定了软件开发的范型。

二. 软件开发范型：

一个软件开发范型是一个用来指导软件开发过程的技术集合。

1. 软件开发过程由三部分组成：
   1) 概念化
   2) 表示
   3) 实现
2. 项目概念化：

   项目的概念化关注软件开发者怎样考虑待解决的问题，用来考虑和讨论待编程系统的各种要素决定了它的概念化。

   例如：在一个面向过程的范型下，用过程(函数、过程和子程序)来刻画系统。

   系统的概念化描述了开发者用来组织他们对项目的思考和分析的思维结构。
3. 项目表示：

   将项目的概念化书写出来就叫做项目的表示，它被用在软件开发的范型中。表示必须能够以一种有效的、无二义的方式描述整个项目。

   1) 创建表示时，约定一个怎样表示项目不同部分的规则集合， 这些规则成为符号。

      例如：

      - *面向过程的范型中的符号可能会用椭圆表示进程，用有向边（箭头）表示进程之间的数据流；*
      - *面向对象的开发范型中可能用矩形表示对象，用有向边表示对象之间的关系。*

   2) 使用符号的一个基本目的是以如下方式表示系统，即以一种无二义的而且在观察特征时不受人的背景知识影响的方式来表示系统。

   3) 理想情况下，符号应该导致系统的这样一种表示，即它在领域专

家和技术人员看来是同一事物。

4. 项目实现：

项目的实现关注如何构造组成软件的源代码。

三. 软件工程：

软件工程是一门建立在系统化、规范化、数量化等工程原则和方法上的，关于软件开发各阶段定义、任务和作用的工程学科。软件工程包括两方面内容：软件开发技术和软件项目管理。软件开发技术包括软件开发方法学、软件工具和软件工程环境；软件项目管理包括软件度量、项目估算、进度控制、人员组织、配置管理和项目计划等。

1. 软件工程的生命周期：

经典的软件工程思想将软件开发分成 5 个阶段：

1) 需求分析阶段（Requirements Capture）：
2) 系统分析与设计（System Analysis and Design）：
3) 系统实现（Implementation）：
4) 测试（Testing）：
5) 维护（Maintenance）：

2. 软件工程技术简史：

1) 结构化编程：

在结构化编程中，goto 语句被函数调用所替代，程序由顺序、选择和循环三种基本控制结构复合而成。

2) 功能分解：

功能分解是一个把待实现的系统分解成一系列逐步增加细节的概念化过程，这种概念化可以利用一种称为结构图的表示来通信。

3) 结构化分析和设计

结构化分析是在结构化编程和功能分解的基础上建立起来的。随着结构化分析和设计的引入，最终实现系统的提交将成为软件开发过程中多个里程碑之一，而不是惟一的里程碑。

4) 以数据为中心的范型

以数据为中心的方法学的贡献在于用一种称为数据建模的技术扩展了结构化分析。

- 数据建模的目的在于确定整个组织所需的数据，从而创建一个类似于关系数据库的、集中的、完整的数据存储库，进而

可以开发单个应用并从这个集中的数据库中提取其所需的数据

- 数据建模用一种称为 ER 图的图形技术来表示
- 在数据建模初期之后，单个应用可以采用集中来自中心数据储存库的数据进行结构化分析和设计的方法来开发

5）面向对象范型：

- 抽象数据类型

  一个数据类型是一个潜在的不同种类数据块的集合（或编组），一个抽象数据类型不仅集中了数据而且包括对这些数据的运算。

- 继承

  抽象数据类型与面向对象范型中的对象相似，但对象具有抽象数据类型所没有提供的另外一些属性。尤其，继承是面向对象范型的一个重要概念。对象可以在一个继承层次内定义，这里，详细的对象类可以继承比它更通用的祖先类的数据和操作。

3. 三种主要的软件工程范型：

三种主要的软件工程范型是面向过程的范型、以数据为中心的范型与面向对象范型。

三种主要范型的三条统一的概念：

1）模块化

2）建模

3）抽象

§1.2 模型

一. 模型：

1. 模型：

简而言之，模型是对现实的简化。模型提供了系统的蓝图，包含细节设计，也包含对系统的总体设计。一个好的模型包括重要的因素，而忽略不相干的细节。每一个系统可以从不同的方面使用不同的模型进行描述，因此每个模型都是对系统从语义上近似的抽象。模型可以是结构的、侧重于系统的组织，也可以是行为的、侧重于系统的动作。

通常，模型由一组图示符号和组织这些符号的规则组成，利用它们来定义和描述问题域中的方法和概念。

2. 建模的重要性：

1）一个揭示建模重要性的例子：

如果你想给自己的爱犬盖个窝，开始的时候你的手头上有一堆木材、一些钉子、一把锤子、一把木锯和一把尺子。在开工之前只要稍微计划一下，你就可以几个小时之内，在没有任何人帮助的情况下盖好一座狗窝。只要它容得下你得的爱犬，能遮风挡雨就可以了。就算差一点，只要你的狗不那么娇贵也是说得过去的。

如果你想为你的家庭建一座房子，开始的时候你的手头上也有一堆木材、一些钉子和一些基本的工具。但是这将要占用你很长的时间，因为你家庭成员的要求肯定比你的狗高出很多。在这种情况之下，除非你长期从事这项工作，否则最好在打地基之前好好的规划一下。首先，要对将要建造的房子设计一幅草图。如果想建造一座满足家庭需要的高质量的房屋，你需要画几张蓝图。考虑各个房间的用途以及照明取暖设备的布局。做好以上工作以后，你就可以对工时和工料做出合理的估计。尽管以人的能力可以独自盖一座房子，但是你会发现同其他人合作会更有效率，这包括请人帮忙或者买半成品材料。只要坚持你的计划并且不超过时间和财政的限制，你的建造计划就成功了一半多。

如果你想建造一幢高档的写字楼，那么刚刚开始就准备好材料和工具是无比愚蠢的行为，因为你可能正在使用其他人的钱，而这些人将决定建筑物的大小、形状和样式。通常情况下，投资人甚至会在开工以后改变他们的想法，你需要做额外的计划，因为失败的代价巨大。你有可能只是很多个工作组之一，所以你的团队需要各种各样的图纸和模型同其他小组进行沟通。只要人员、工具配置得当，按照计划施工，你肯定会交付令人满意的工作。如果你想在建筑行业长久地干下去，你不得不在客户的需求和实际的建筑技术之间找到好的契合点。

2）软件的建模：

许多软件开发组织总是像建造狗窝一样进行软件开发，得到的结果通常情况下都是失败；如果你像盖房子或者盖写字楼一样开发软件，问题就不仅仅是写代码，而是怎么样写正确的代码和

怎么样少写代码。这就使得高质量的软件开发变成了一个结构、过程和工具相结合的问题。

3) 建模的作用：

用户可以通过模型直观地看到用户与系统间的交互；分析人员可以看到模型对象间的交互；开发人员可以看到要开发的对象和每个对象的任务；测试人员可以看到对象间的交互并根据这些交互准备测试案例；项目管理人员可以看到整个系统及各部分的交互；而信息总管可以看看高层模型，看看公司的各个系统如何相互交互。

二. 建模的目标：

建立模型可以帮助开发者更好地理解正在开发的系统。通过建模，要实现以下四个目标：

1. 帮助开发人员按照实际情况或按照开发人员所需要的样式对系统进行可视化；
2. 允许开发人员详细说明系统的结构或行为；
3. 给出一个指导开发人员构造系统的模板；
4. 对开发人员作出的决策进行文档化。

三. 建模的四条基本原则：

在工程学科中，对模型的使用有着悠久的历史，人们从中总结出了四条基本的建模原则：

1. 选择要创建什么模型对如何动手解决问题和如何形成解决方案有着意义深远的影响；
2. 每一种模型可以在不同的精度级别上表示；

使用者的身份和使用的原因是评判模型好坏的关键。分析者和最终用户关心“是什么”，而开发者关心“怎么做”。所有的参与者都想在不同的时期、从不同的层次了解系统。

3. 最好的模型是与现实相联系的；

所有的模型都是简化的现实，关键的问题是必须保证简化过程不会掩盖任何重要的细节。

4. 孤立的模型是不充分的。对每个重要的系统最好用一组几乎独立的模型去处理。

孤立的模型是不完整的。任何好的系统都是由一些几乎独立的模型拼凑出来的。（就像造一幢房子一样，没有一张设计图可以包括所有的细节。至少楼层平面图、电线设计图、取暖设备设计图和管

道设计图是需要的)。“几乎独立”是指每个模型可以分开来建立和研究，但是他们之间依然相互联系。

四. 面向对象建模：

在软件业中，建立模型有两种最常用的方法：

1. 基于算法方法建模；

传统的软件开发采用基于算法的方法。在这种方法中，主要的模块是程序或者函数，这使得开发人员将注意力集中在控制流和将庞大的算法拆分成各个小块。虽然说这种方法本身并没有错误，但是随着需求的变化和系统的增长，运用这种方法建立起来的系统很难维护。

2. 面向对象的建模。

现代的软件开发采用面向对象的方法。在这种方法中，主要的模块是对象或者类。对象通常是从问题字典或者方法字典中抽象出来的，类是一组具有共同特点的对象的描述。每一个对象都有自己的标识、状态和行为。

§1.3 面向对象分析与设计

一. 面向对象的基本概念：

面向对象＝对象＋类＋继承＋通信（Coad 和 Yourdon 给出的简单定义)。

1. 面向对象技术的基本观点：
   1) 客观世界由对象组成，任何客观事物都是对象，复杂对象可以由简单对象组成。
   2) 具有相同数据和操作的对象可归纳成类，对象是类的一个实例。
   3) 类可以派生出子类，子类除了继承父类的全部特性外还可以有自己的特性。
   4) 对象之间的联系通过消息传递来维系。

   由于类的封装性，它具有某些对外界不可见的数据，这些数据只能通过消息请求调用可见的方法来访问。
2. 面向对象方法的基本出发点就是尽可能地按照人类认识世界的方法和思维方式来分析和解决问题，使人们分析、设计一个系统的方法尽可能接近认识一个系统的方法。

二. 面向对象的核心元素：

1. 对象：

客观世界里的任何实体都可以被称之为对象，对象可以是具体的有形的物，如：人、汽车等；也可以是无形的事物或概念，如：抽象的规则、计划或事件。

2. 封装：

是面向对象方法的一个重要原则。是指把属性和操作封进一个对象里，它的内部信息对外界隐藏，不允许外界直接存取对象的属性，只能通过对象提供的有限的接口对对象的属性数据进行操作。对于外界来说，只能知晓对象的外部行为而无法了解对象行为的内部实现细节，这样可以保证对象内部属性数据的安全性。

封装有两层含义：

1）结合性，即把对象的全部属性和方法结合起来，形成一个独立的不可分割的单位；

2）信息隐藏性，即尽可能隐蔽对象的内部细节，对外形成一个边界，只保留有限的对外接口使之与外部发生联系。

3. 消息

消息就是向对象发出的请求，一个消息包含消息名、接收对象的标志、服务标志、输入标志、输入信息、回答信息等。

4. 类

类是对象的抽象，类好比是一个对象模板。

5. 继承

“泛化”

6. 多态性

同一个操作作用于不同的对象，可以有不同的解释，产生不同的执行结果，即在程序运行的时候，根据对外部触发进行反应的对象的不同，可以动态的选择某个操作的实现方法，这就是多态性。

1）多态性分为如下两种：

- 编译时的多态性。编译时的多态性是通过重载来实现的。系统在编译时，根据传递的参数、返回的类型等信息决定实现何种操作。
- 运行时的多态性。运行时的多态性就是指直到系统运行时，才根据实际情况决定实现何种操作。

2）多态性大致有以下三种表现方式:

- 通过接口实现多态性。

- 通过继承实现多态性。
- 通过抽象类实现多态性。

7. 结构与连接

1）为了使系统能够有效地映射问题域，系统开发者需认识并描述对象之间的以下几种关系：

- 对象的分类关系。
- 对象之间的组成关系。
- 对象属性之间的静态联系。
- 对象行为之间的动态联系。

2）面向对象方法分别用以下几种结构和连接来反映对象之间的几种关系：

- 一般/特殊结构。
- 整体/部分结构。
- 实例连接。
- 消息连接。

三. 面向对象的分析：

面向对象分析（Object-Oriented Analysis），简称 OOA,是指利用面向对象的概念和方法为软件需求建造模型，以使用户需求逐步精确化、一致化、完全化的分析过程。

1. 面向对象分析模型：

面向对象分析中建造的模型主要有对象模型、动态模型和功能模型。面向对象分析的关键是识别出问题领域内的对象，在分析它们之间的相互关系之后建立起问题领域的简洁、精确和可理解模型。

2. 面向对象分析的基本过程：

1）问题论域分析：业务范围、业务规则、业务处理过程、系统责任范围、边界。

2）发现和定义对象分类及内部特征：属性和服务。

3）识别对象的外部联系：一般特殊、整体与部分，实例连接，消息连接。

4）建立系统的静态结构模型：绘制对象类图和对象图，系统与子系统结构图等，编制文档。

5）定义用例，建立系统的动态行为模型：分析系统行为，建立动态模型，并用图形和文字表达。

6）建立详细说明。

7）原型开发。

3. 面向对象分析的基本原则：

1）抽象原则：

面向对象分析方法中的类就是抽象得到的：系统中的对象使对现实世界中的事物的抽象；类是对对象的抽象；一般类是对特殊类的进一步抽象；属性是事物静态特征的抽象；服务是事物动态特征的抽象。

2）分类原则：

分类就是把具有相同属性和服务的对象划分为一类，用类作为这些对象的抽象描述。分类原则实际上是抽象原则运用于对象描述时的一种表现形式，通过不同程度的抽象可以形成一般/特殊结构。

3）聚合原则：

聚合是把一个复杂的事物看出若干简单事物的组合体，从而简化对复杂事物的描述。在面向对象分析中运用聚合原则将一个较复杂的事物划分为几个组成部分，分别用整体和部分进行描述，这样形成的整体/部分结构不仅能清楚地表达事物的组成关系，还可以简化分析过程。

4）关联原则：

关联是人类思考问题时常用的方法，通过一个事物可以联想到另外的事物，产生联想的原因是事物之间存在着某些联系。在面向对象的分析过程中运用关联原则可以在系统模型中明确地表示对象之间的静态联系。

5）消息通信原则：

这一原则要求对象之间只能通过消息进行通信，而不允许在对象外直接地存取对象内部的属性。

四. 面向对象的设计：

分析是提取和整理用户需求，并建立问题域精确模型的过程；设计则是把分析阶段得到的需求转变成符合成本和质量要求的、抽象的系统实现方案的过程。从面向对象分析到面向对象的设计是一个逐步扩充模型的过程，也可以说面向对象设计是用面向对象观点建立求解域模型的过程。

1. 面向对象设计的模型

在设计期间主要扩充 4 个组成部分：

1) 人机交互部分

人机交互部分包括有效的人机交互所必需的实际显示和输入。

2) 问题域

问题域部分放置面向对象分析的结果并管理面向对象分析的某些类和对象、结构、属性和方法。

3) 任务管理

任务管理部分包括任务定义、通信和协调、硬件分配及外部系统。

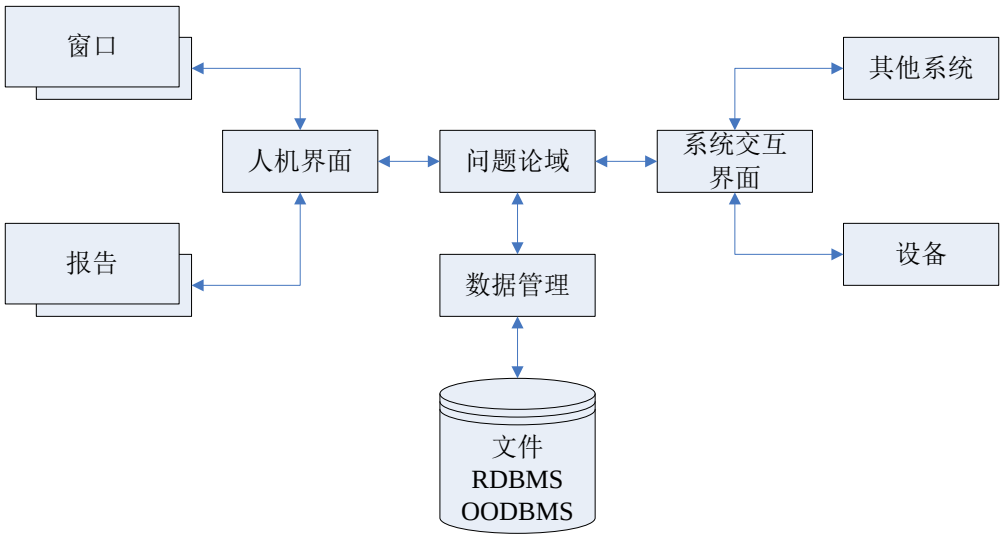
4) 数据管理。

数据库管理部分包括对永久性数据的访问和管理。

2. 面向对象的高层设计

高层设计的主要任务就是设计前面提到的 4 个部分：问题域部分、人机交互部分、任务管理部分和数据管理部分。

高层设计的结构模型如图所示：



1) 问题域子系统的设计

问题域子系统应该包括域应用问题直接有关的全部类和对象。识别和定义这些类和对象的工作在面向对象分析中已经开始，在面向对象分析阶段得到的有关应用的概念模型描述了要解决的问题。在面向对象设计阶段，将继续面向对象分析阶段的工作，对分析阶段的结果进行修改和增补，对分析时构造的模型中的某些类与对象、结构、属性、操作进行组合与分解。

问题域子系统的设计工作主要有以下几个方面：

- 复用已有的设计
- 把与问题论域相关的类关联，建立类的层次结构
- 创建一般化类
- 改进系统性能
- 加入较低层的构件

2）人机交互子系统的设计

在设计阶段必须根据需求把交互细节加入到用户界面设计中，包括人机交互所必需的实际显示和输入。

用户界面部分设计主要由以下几个方面组成：

- 用户分类。
- 描述人及其任务的脚本。
- 设计命令层。
- 设计与用户的详细交互。

  用户界面详细设计要尽量做到如下几点：

  - 采用一致的术语、一致的步骤和一致的活动；
  - 减少用户敲键和鼠标点击的次数，减少完成某件事所需要的下拉菜单的距离；
  - 当用户等待系统完成一个活动时，要提供一些反馈信息；
  - 在操作出现错误时，要全部或部分恢复到原来的状态；
  - 采取符合人类习惯图形界面。
- 继续进行原型设计。
- 设计人机交互类。

3）任务管理子系统的设计

任务是进程的别称，是执行一系列活动的一段程序。当系统中有许多并发行为时，需要依照各个行为的协调和通信关系，划分各种任务，以简化并发行为的设计和编码。

定义任务的工作主要包括如以下几个方面：

- 为任务命名，并简要说明这个问题。
- 定义各个任务如何协调工作，指出它是事件的驱动还是时钟驱动。
- 定义各个任务之间如何通信，任务将从哪里取值，任务执行得到的结果将送往何方。

4）数据管理子系统的设计

数据管理部分提供了在数据管理系统中存储和检索的对象

的基本结构，包括对永久性数据访问和管理。

3. 面向对象的类设计

在面向对象的分析过程中，对问题论域所需的基本类进行了模型化和抽象，但在系统的最终实现应用时只有这些类是不够的，还要根据需要追加一些类。追加辅助类的工作在类设计的过程中完成。

§1.4 UML 简介

一. UML：

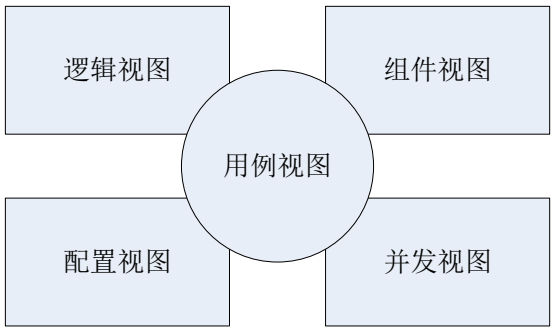
UML(Unified Modeling Language),即统一建模语言，是一个通用的可视化建模语言，用于对软件进行描述、可视化处理、构造和建立软件系统制品的文档。UML 是一种绘制软件蓝图的标准语言，它记录了对必须构造的系统的决定和理解，可用于对系统的理解、设计、浏览、配置、维护和信息控制。UML 适用于各种软件开发方法、软件生命周期的各个阶段、各种应用领域以及各种开发工具，是一种总结了以往建模技术的经验并吸收当今优秀成果的标准建模方法。

二. UML 的构成：

总体来说，UML 由以下几个部分构成：

1. 视图：

视图是表达系统的某一方面特征的 UML 建模元素的子集，视图并不是图，它是由一个或者多个图组成的对系统某个角度的抽象。UML 中的视图大致分为 5 种：（如图所示）



1）用例视图：

用例视图强调从系统的外部参与者（主要是用户）角度看到的或需要的系统功能。描述系统应该具备的功能。用例模型的用途是列出系统中的用例和参与者，并显示哪个参与者参与哪个用例的执行。用例视图是其他视图的核心，它的内容直接驱动其他

视图的开发。系统要提供的功能都是在用例视图中描述的，用例视图的修改会对所有其他的视图产生影响。此外，通过测试用例视图，还可以检验和最终校验系统。

2）逻辑视图：

逻辑视图从系统的静态结构和动态行为的角度显示如何实现系统的功能。描述用例视图中提出的系统功能的实现。与用例视图相比，逻辑视图主要关注系统内部，它既描述系统的静态结构（类、对象以及它们之间的关系），也描述系统内部的动态协作关系。系统的静态结构在类图和对象图中进行描述，而动态模型则在状态图、顺序图、协作图以及活动图中进行描述。逻辑视图的使用者主要是设计人员和开发人员。

3）并发视图：

并发视图。显示系统的并发性，解决在并发系统中存在的通信和同步问题。并发视图主要考虑资源的有效利用、代码的并发执行以及系统环境中异步事件的处理。除了将系统划分为并发执行的控制以外，并发视图还需要处理线程之间的通信和同步。并发视图的使用者是开发人员和系统集成人员。并发视图由状态图、协作图，以及活动图组成。

4）组件视图：

组件视图显示代码组件的组织结构。组件视图描述系统的实现模块以及它们之间的依赖关系。组件视图主要由组件图构成，它的使用者主要是开发人员。

5）配置视图：

配置视图显示系统的具体部署（部署是指将系统配置到由计算机和设备组成的物理结构上）。配置视图显示系统的物理部署，它描述位于节点上的运行实例的部署情况。配置视图主要由配置图表示，它的使用者是开发人员、系统集成人员和测试人员。

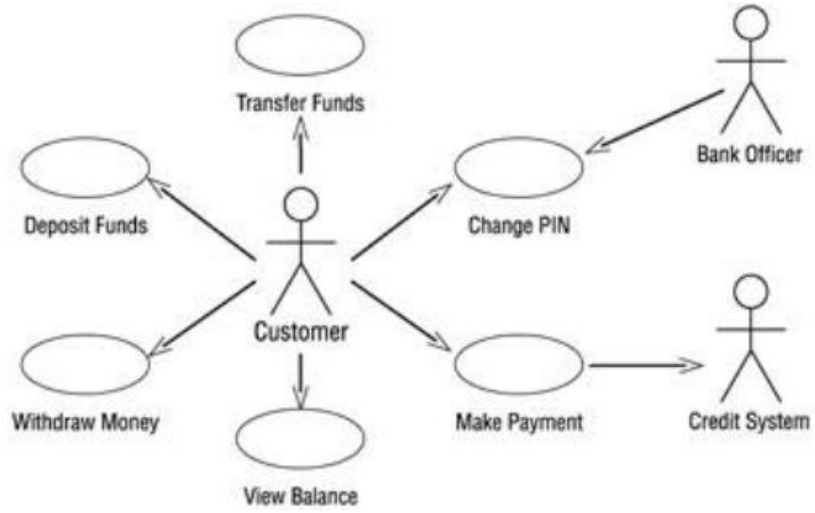
2. 图：

视图由图组成，UML 通常提供 9 种基本的图

1）用例图**(Use Case Diagram)**：

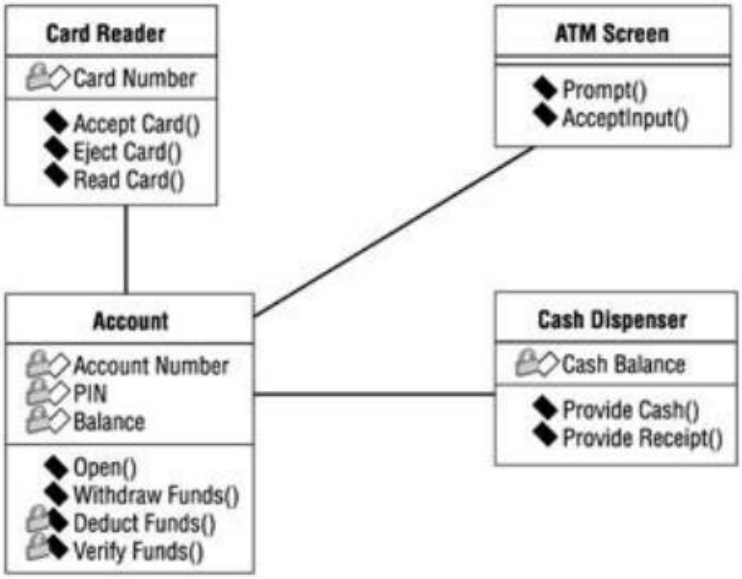
显示多个外部参与者以及他们与系统提供的用例之间的连接。用例是系统中的一个可以描述参与者与系统之间交互作用的功能单元。用例仅仅描述系统参与者从外部观察到的系统功能，并不描述这些功能在系统内部的具体实现。用例图的用途是列出

系统中的用例和参与者，并显示哪个参与者参与了哪个用例的执行。



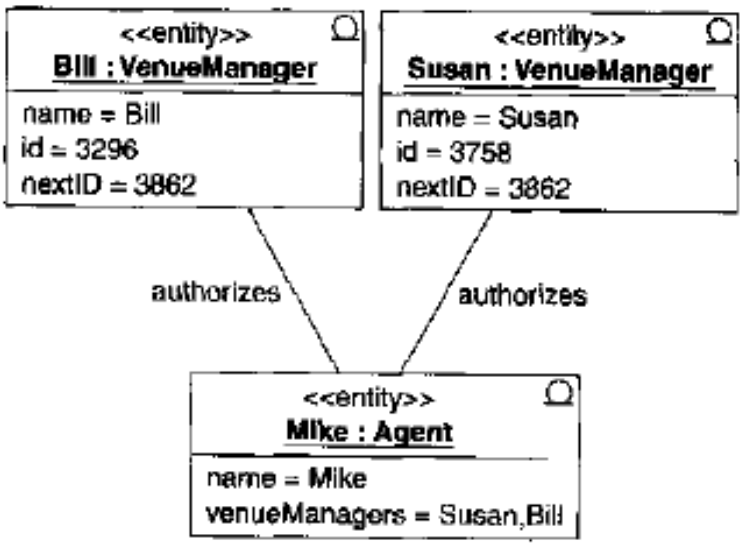
2）类图**(Class Diagram)**：

类是对应用领域或应用解决方案中概念的描述。类图以类为中心组织，类图中的其他元素或属于某个类，或与类相关联。



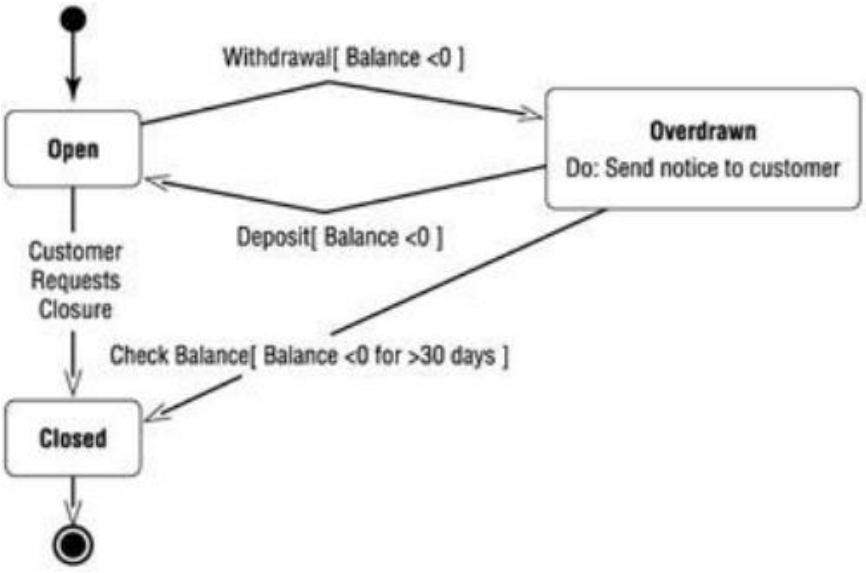
3）对象图**(Object Diagram)**：

对象图是类图的变体，它使用与类图相似的符号描述，不同之处在于对象图显示的是类的对象实例而非实际的类。可以说，对象图是类图的一个例子，用于显示系统执行时的一个可能的快照，即在某一时间点上系统可能呈现的样子。



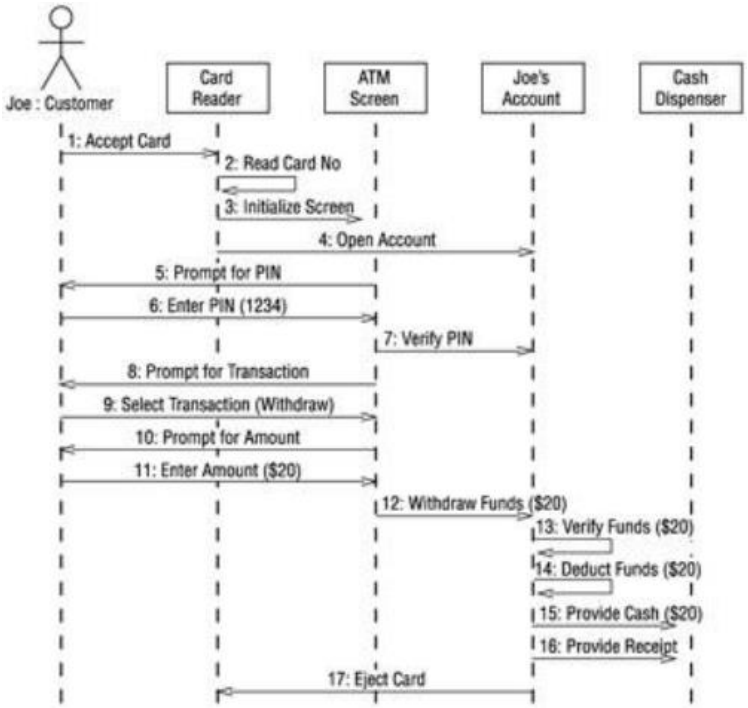
4）状态图**(State Diagram)**：

是对类描述的补充，它用于显示类的对象可能具备的所有状态，以及引起状态改变的事件。状态图由对象的各个状态和连接这些状态的转换组成。每个状态对一个对象在其生命周期中满足某种条件的一个时间段建模。事件的发生会触发状态间的转换，导致对象从一种状态转化到另一新的状态。



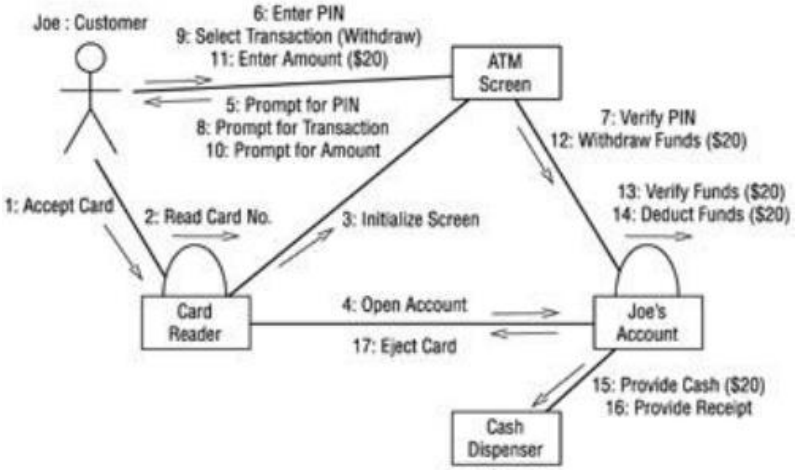
5）顺序图**(Sequence Diagram)**：

显示多个对象之间的动态协作，重点是显示对象之间发送的消息的时间顺序。顺序图也显示对象之间的交互，就是在系统执行时，某个指定时间点将发生的事情。顺序图的一个用途是用来表示用例中的行为顺序，当执行一个用例行为时，顺序图中的每条消息对应了一个类操作或状态机中引起转换的触发事件。



6）协作图**(Collaboration Diagram)：**

对在一次交互中有意义的对象和对象间的链建模。除了显示消息的交换（称之为交互）以外，协作图也显示对象以及他们之间的关系。

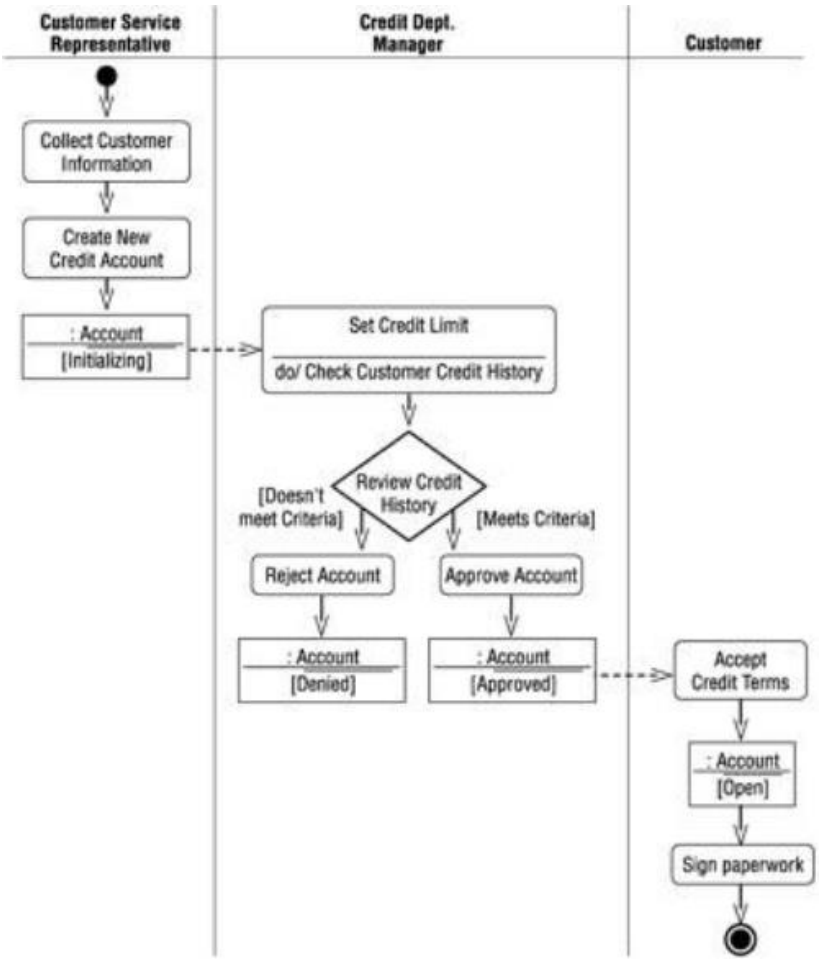


顺序图和协作图都可以表示对象间的交互关系，但它们的侧重点不同。顺序图用消息的几何排列关系来表达消息的时间顺序，各角色之间的关系是隐含的；协作图用各个角色的几何排列来表示角色之间的关系，并用消息来说明这些关系。

7）活动图**(Activity Diagram)：**

是状态图的一个变体，用来描述执行算法的工作流程中涉及

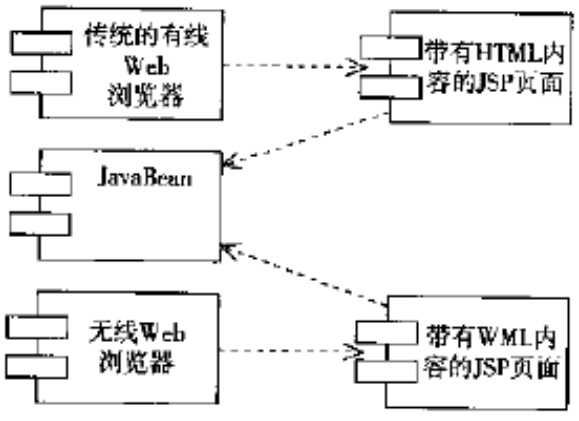
的活动。



状态图表示一个对象在一段时间内的状态变化，而活动图则描述多个对象的状态变化序列。

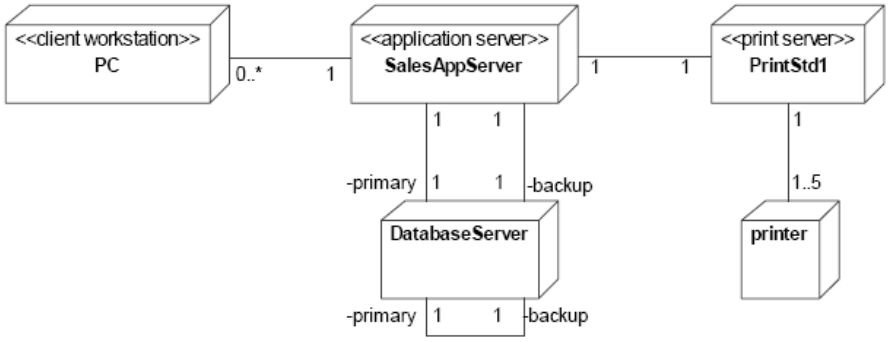
8）组件图**(Component Diagram)：**

用代码组件来显示代码物理结构，组件可以是源代码组件、二进制组件或一个可执行的组件。一个组件包含它所实现的一个或多个逻辑类的相关信息。

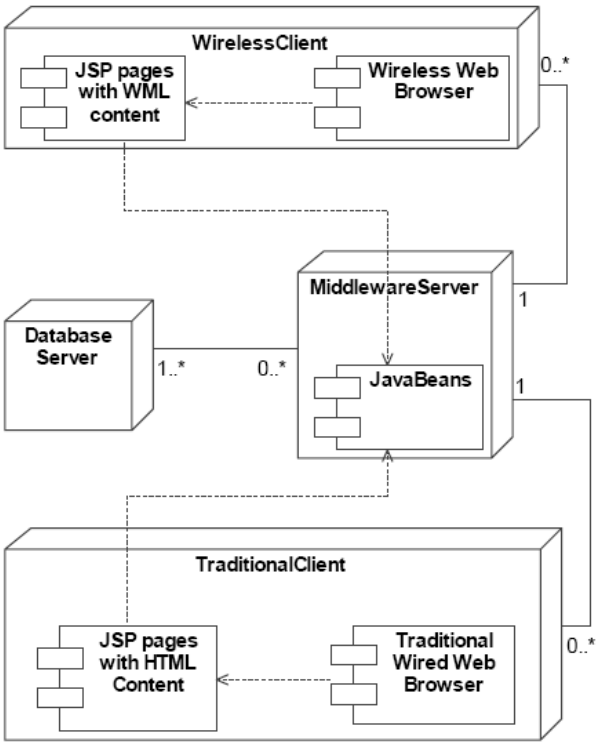


9）配置图**(Deployment Diagram)**：

用于显示系统中的硬件和软件的物理结构。配置图不仅可以显示实际的计算机和设备（节点），还可以显示它们之间的连接和连接的类型。在配置图中显示哪些节点内已经分配了可执行的组件和对象，以显示这些软件单元分别在哪个节点上运行。



组件图和配置图的组合：（如下图所示）



3. 模型元素：

UML 中的模型元素包括事物和事物之间的联系。

1）事物：

事物描述了一般的面向对象的概念，是 UML 模型中面向对象的基本的建筑块，它们在模型中属于静态部分，代表物理上或概念上的元素。如：类、对象、接口、消息和组件等。

UML 中的事物可分为以下四类：

- 结构事物

  结构事物共有 7 种：

  - 类
  - 接口
  - 协作
  - 用例
  - 活动类
  - 组件
  - 节点

- 动作事物

  是 UML 模型中的动态部分，它们是模型的动词，代表时间和空间上的动作。

  结构事物共有 2 种：

  - 交互
  - 状态机

- 分组事物

  是 UML 模型中组织的部分。

  分组事物只有一种，称为包。包是一种将有组织的元素分组的机制，结构事物、动作事物甚至其他的分组事物都可以放在一个包中。

- 注释事物

  是 UML 模型中的解释部分。

2）UML 中的关系

UML 中的关系有以下几种：

- 关联关系
- 依赖关系
- 泛化关系
- 实现关系
- 聚合关系

4. 通用机制：

1）修饰：

在使用 UML 建模时，可以将图形修饰（如：字体、颜色等）附加到 UML 图中的模型元素上，这种修饰为图中的模型元素增加了语义。

2）注释：

注释是以自由的文本形式出现的，它的信息类型是不被 UML 解释的字符串。注释可以附加到任何模型中去，可以放置在模型的任意位置上，并且可以包含任意类型的信息。

3）通用划分。

UML 对其模型元素规定了两种类型的通用划分：

- 型－实例
- 接口－实现

4）扩展机制。

UML 中包含 3 种主要的扩展组件：

- 构造型

  构造型是由建模者设计的新的模型元素，新的模型元素的设计要以 UML 已定义的模型元素为基础。

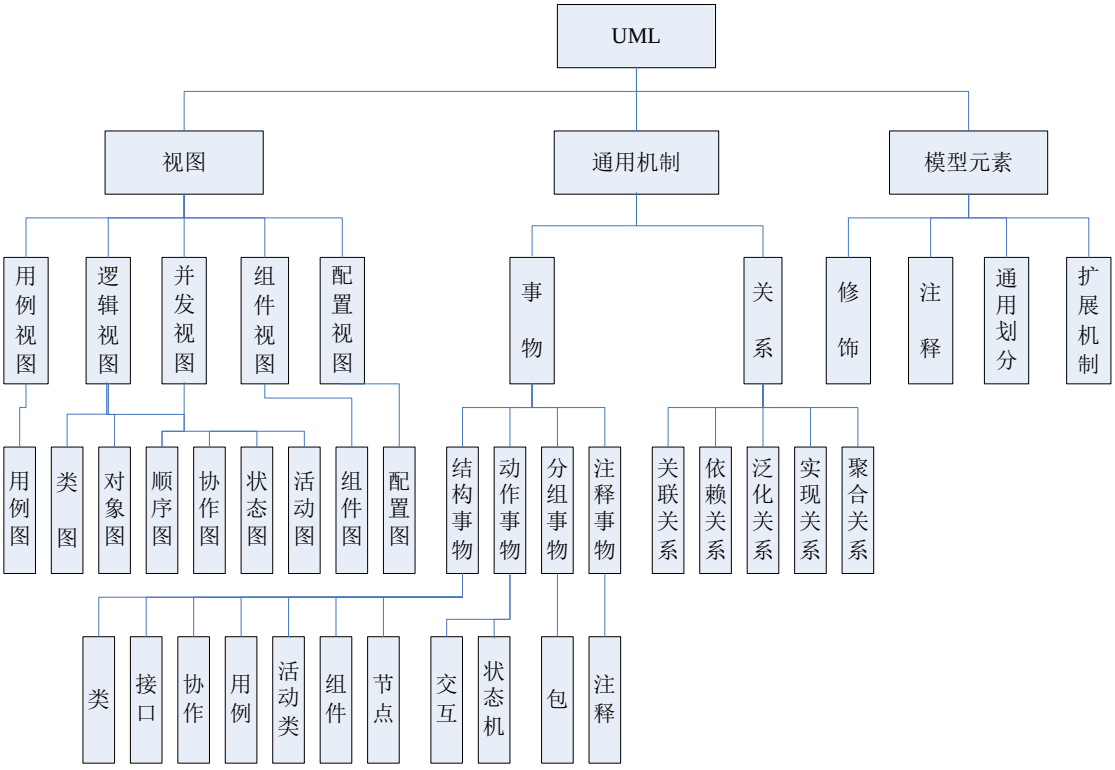
- 标记值

  标记值是附加到任何模型元素上的命名的信息块。

- 约束

  约束是用某种形式化语言或自然语言表达的语义关系的文字说明。

5. UML 的组成结构：（如下图所示）



三. 基于 UML 的软件开发：

1. 基于 UML 的面向对象分析、设计过程

运用 UML 进行面向对象的系统分析设计，通常都要经过如下 3 个步骤：

1）识别系统的用例和参与者。

首先要对项目进行需求调研，分析项目的业务流程图和数据流程图，以及项目中涉及的各级操作人员，识别出系统中的所有用例和参与者；接着分析系统中各参与者和用例间的联系，使用 UML 建模工具画出系统的用例图；最后，勾画系统的概念层模型，借助 UML 建模工具描述概念层的类图和活动图。

2）进行系统分析并抽象出类。

系统分析的任务是找出系统的所有需求并加以描述，同时建立特定领域模型，建立领域模型有助于开发人员考察用例。从实际需求中抽象出类，并描述各个类之间的关系。

3）设计系统，并设计系统中类及其行为。

设计阶段由结构设计和详细设计组成。结构设计是高层设计，其任务是定义包（子系统）、包间的依赖关系和主要通信机制。包有利于描述系统的逻辑组成部分以及各部分之间的依赖关系。详细设计主要用来细化包的内容，清晰描述所有的类，同时使用 UML 的动态模型描述在特定环境下这些类的实例的行为。

2. UML 建模的简单流程

利用 UML 建造系统时，在系统开发的不同阶段有不同的模型，并且这些模型的目的是不同的：

1）在分析阶段，模型的目的是捕获系统的需求，建立“现实世界”的类和协作的模型；

2）在设计阶段，模型的目的是在考虑实现环境的情况下，将分析模型扩展为可行的技术方案；

3）在实现阶段，模型是那些书写并编译的实际源代码；

4）在部署阶段，模型描述了系统是如何在物理结构中部署的。

§1.5 小结

第二章　用例模型

内容概要

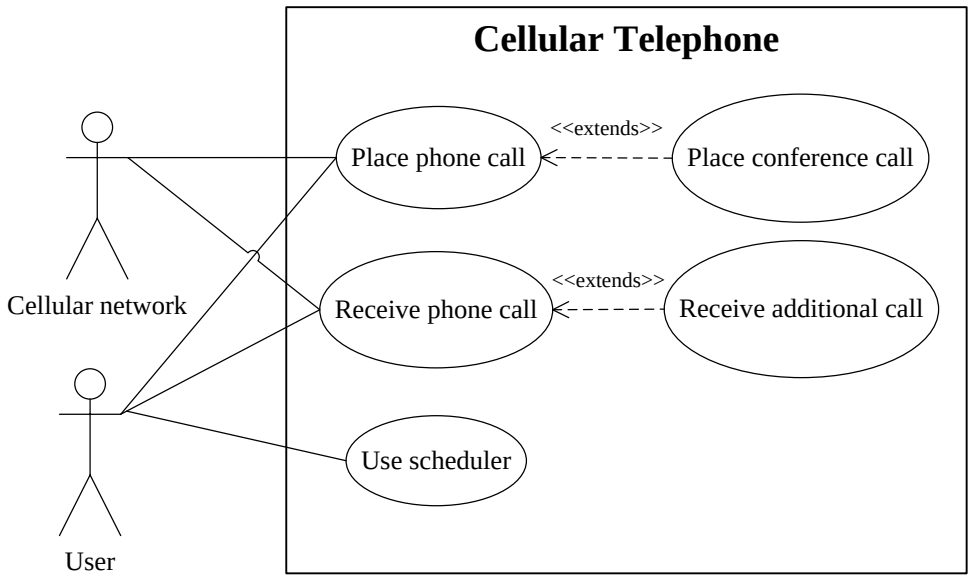
- ✧ 基本概念
- ✧ 用例建模
- ✧ 案例分析

§2.1 基本概念

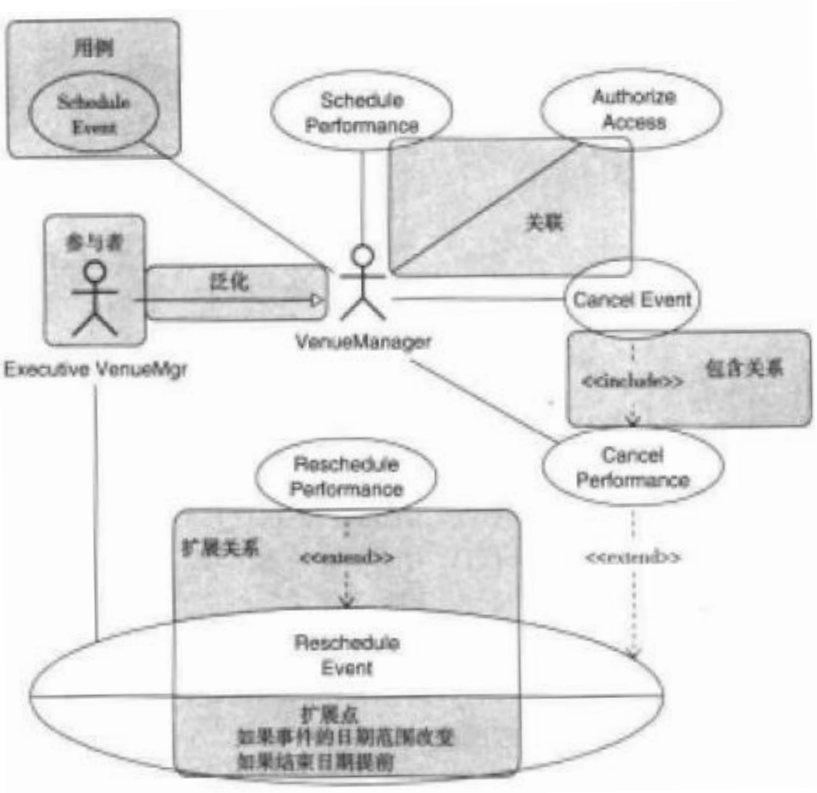
一. 用例图（Use Case Diagram）：

用例图是显示一组用例、参与者以及它们之间关系的图，主要用于对系统、子系统或类的行为进行建模，包括：参与者、系统边界、用例和关系等元素。

- ✧ 用例图主要用于对系统、子系统或类的行为进行可视化，以使用户能够理解如何使用这些元素，并使开发者能够实现这些元素；
- ✧ 用例图是由软件需求分析到最终实现的第一步，它描述人们希望如何使用一个系统；
- ✧ 用例图显示谁将是相关的用户、用户希望系统提供什么服务，以及用户需要为系统提供的服务，以便使系统的用户更容易地理解这些元素的用途，也便于软件开发人员最终实现这些元素。



1. 内容：



1）参与者：参与系统成功操作的某个人、系统、设备甚至是企业所扮演的角色。

2）用例：标志系统的某个关键行为。如果没有该行为，系统将不能满足参与者的需求。每个用例都表达了系统必须达到的目标或必须产生的结果。

3）系统边界

4）关系

5）注解和约束、包等

2. 一般应用：

用例图用于对系统的静态用例视图进行建模，这个视图主要支持系统的行为，即该系统在它的周边环境的语境中所提供的外部可见服务。

当对系统的静态用例视图建模时，通常可以用以下两种方式之一来使用用例图：

1）对系统的语境建模

对一个系统的语境进行建模，包括围绕整个系统划一个边界，并声明有哪些参与者位于系统之外并与系统进行交互。在这里，用例图说明了参与者以及他们所扮演的角色的含义。

2）对系统的需求建模

对一个系统的需求进行建模，包括说明这个系统应该做什么（从系统外部的一个视点出发），而不考虑系统应该怎样做。在这里，用例图说明了系统想要的行为。

3. 构造用例图的步骤：

1）定义系统的上下文；

- 确定参与者和它们的责任；
- 确定用例，即确定带有特定目的或产生特定结果的系统行为。

2）对参与者和用例进行权衡，以便精化模型。例如：分拆或合并定义；

3）衡量用例以找出包含关系；

4）衡量用例以找出扩展关系；

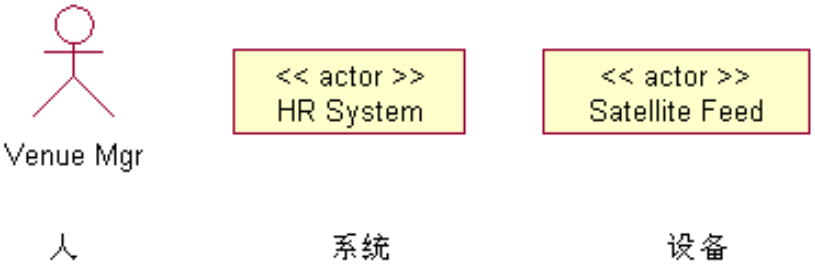
5）对参与者和用例进行研究，查找是否存在泛化关系。

二. 参与者（Actor）：

参与者是系统外部的一个实体，是参与系统成功操作的某个人、系统、设备甚至企业所扮演的角色。参与者以某种方式参与用例的执行过程，参与者通过向系统输入或请求系统输入某些事件来触发系统的执行。

- ✧ 参与者由参与用例时所担当的角色表示；
- ✧ 每个参与者可以参与一个或多个用例；
- ✧ 某个参与者的存在是因为该参与者与系统具有交互行为，否则该参与者就是不必要的，所谓与系统有交互行为，即参与者可以向用例发送相应的消息，并且可以接收用例反馈的消息；
- ✧ 参与者仅通过关联与用例相连，一个参与者和一个用例之间的关联表示两者之间的通信，任何一方都可发送和接收消息；
- ✧ 参与者通过交换信息与用例发生交互（因此也与用例所在的系统或类发生了交互），而参与者的内部实现与用例是不相关的，可以用一组定义其状态的属性充分描述参与者。

1. 符号：



2. 分类：
   1）真实的人，即用户，是最常用的参与者；
   2）其他的系统；
   3）一些可以运行的进程，如：时间。
3. 参与者描述：
   1）模板：

参与者：_____
参与者职责：
_____
_____
参与者识别问题：
_____
_____

   2）范例：

参与者：储户
参与者职责：
插入信用卡
输入口令
输入交易金额
参与者识别问题：
使用系统主要功能
对系统运行结果感兴趣

三. 用例：

用例是对一组动作序列（其中包括它的变体）的描述，系统执行该动作序列来为参与者产生一个可观察的结果值。

- ✧ 一个用例描述一组序列，每一个序列表示系统外部的事物（系统的参与者）与系统本身的交互，这些行为实际上是系统级的功能，用例可视化、详述、构造和文档化在需求获取和分析过程中所希望的系统行为，一个用例描述了系统的一个完整的功能需求；
- ✧ 用例是外部可见的系统功能单元，这些功能由系统单元所提供，并通过一系列系统单元与一个或多个参与者之间交换的信息所表达；
- ✧ 用例的用途是，在不揭示系统内部构造的前提下定义连贯的行为；
- ✧ 用例的定义包含它所必需的所有行为－执行用例的主线次序、标准行为的不同变形、一般行为下的所有异常情况及其预期反应。

1. 符号：

每个用例都必须有一个唯一的名字以区别于其他用例，用例的名字是一个字符串，包括：简单名（simple name）和路径名（path name），用例的路径名是在用例名前加上所属包的名字。

Package::Place order

2. 特征：

1）用例通常由某个参与者来驱动（启动）执行。

用例通常因参与者而得到执行，参与者总是直接或间接地驱动用例执行，当参与者启动一个用例执行时，通常伴随着某些事件的发生。

2）用例把执行结果的值反馈给参与者。

从系统的角度而言，每个用例都会产生用户可观察的结果，用例在执行完成之前要把结果的值通过一定的方式（往往通过需求定义）反馈给参与者。

3）用例在功能上具有完整性。

每个用例都必须从输入开始，直至产生结果值输出给参与者，否则就不成为用例。

3. 用例描述：

1）模板：

用例：<编号><名称>
特征信息
　　用例在系统中的目标（用例目标描述）
　　范围（当前考虑的是哪个系统）
　　级别（概要任务/首要任务/子功能）
　　前提条件（用例执行前系统应具有的状态）
　　成功后续条件（用例执行成功后应具有的状态）
　　失效后续条件（用例没有完成目标的状态）
　　首要参与者（与该用例关联的首要参与者）
　　触发（启动该用例执行的系统动作）
主要步骤
　　<步骤编号><动作描述>
扩展
　　<有变化情况的步骤编号><条件>：<动作或另一个用例>
变异
　　<步骤或变化编号><变异列表>
相关信息（可选）
　　优先级（该用例对于系统/组织的关键程度）
　　性能目标（该用例的执行时间耗费）
　　频度（该用例被执行的频度）
从属用例（可选，指出用例名）
下属用例
与首要参与者的联系渠道（包括交互式、静态文件、数据库等）
公开问题（可选）
　　列出关于该用例的未解决的问题

2）范例：

用例 5：购买物品
特征信息：
　　用例目标：购买者直接向公司提出购买请求，期望收到货物并付款
　　范围：公司
　　级别：概要任务
　　前提条件：公司知道购买者，以及他的地址等
　　成功后续条件：购买者得到物品，公司获得了物品的价钱
　　失效后续条件：公司没有发送物品，购买者没有寄出款额
　　首要参与者：购买者，任何负责该客户的代理商（或计算机）
　　触发：购买请求
主要步骤：
　　1. 购买者提出购买请求
　　2. 公司记录购买者的姓名、地址、要买的商品信息等
　　3. 公司告知购买者有关商品的信息、价格、发货日期等
　　4. 购买者签单
　　5. 公司产生订单，并将订货传送给购买者
　　6. 公司将发票发给购买者
　　7. 购买者付款
扩展：
　　3. 对于顾客所订之货，公司缺货：重新协商订货
　　4. 购买者直接以信用卡付帐：通过信用卡接收款额（用例 44）
　　7. 购买者返还物品：处理泛化的物品（用例 105）
变异：
　　1. 购买者可以使用以下工具提出购买请求：
　　电话/传真/互联网主页订单/电子交易
　　2. 购买者可以通过以下方式来支付货款：
　　现金或汇款单/支票/信用卡
相关信息：
　　优先级：高
　　性能目标：5 分钟处理订单，45 天内付款
　　频度：200 次/天
从属用例：管理客户关系（用例 2）
下属用例：
　　产生订单（用例 15）
　　通过信用卡接收货款（用例 44）
　　处理泛化的物品（用例 105）
与首要参与者的联系渠道：电话、文件或交互式
公开问题：
　　如果公司只有订单的部分信息该怎么办？
　　如果客户信用卡被盗怎么办？

四. 系统边界：

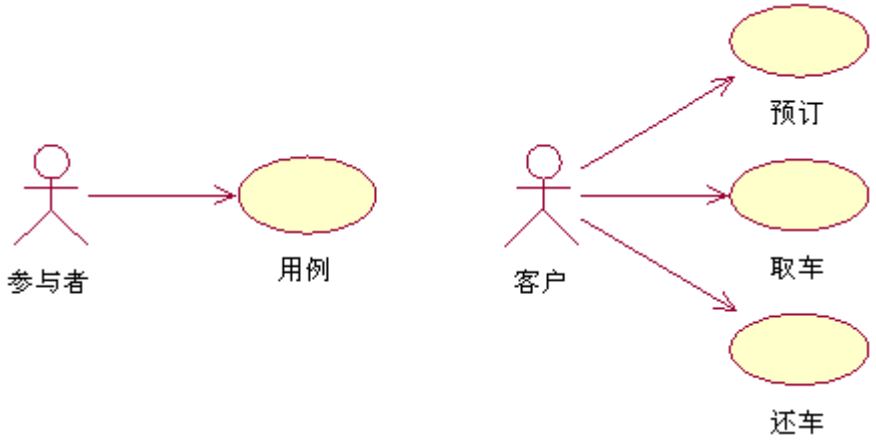
代表的是一个活动范围，用来说明构建的用例的应用范围。

五. 关系：

1. 关联关系：

描述参与者与用例之间的关系，表示参与者与用例之间的通信、交互。每个关联成为在用例描述中加以解释的对话，而每个用例描述又提供了一组脚本，它们有助于开发测试用例。

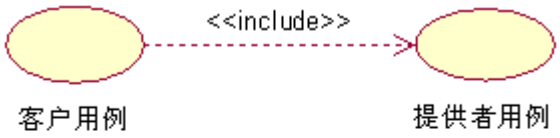
在 UML 中，关联关系使用箭头来表示，如下图所示：

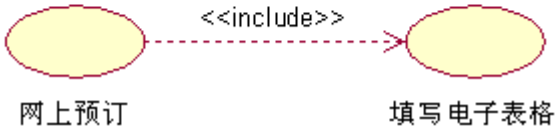


2. 包含关系：

一个用例可以简单地包含其他用例具有的行为，并把它所包含的用例行为作为自身行为的一部分，这被称作包含关系。包含关系标志一个可重用的用例。它可以被无条件地集成到其他的用例中，什么时候或者为什么调用该用例取决于调用它的用例。

在 UML 中，包含关系表示为虚线箭头加<<include>>字样，箭头指向被包含的用例，如下图所示：





包含关系使一个用例的功能可以在另一个用例中使用：

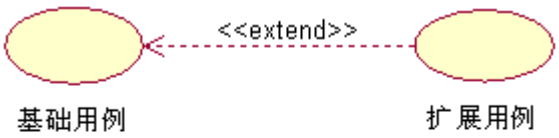
1) 如果两个以上用例有大量一致的功能，则可以将这个功能分解到另一个用例中，其他用例可以和这个用例建立包含关系；

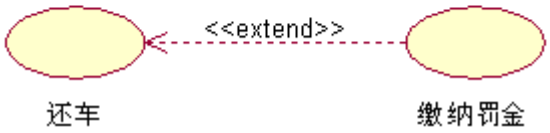
2) 一个用例的功能太多时，可以用包含关系建模两个小用例。

3. 扩展关系：

一个用例也可以被定义为基础用例的增量扩展，这称为扩展关系，扩展关系是把新的行为插入到已有用例中的方法。扩展关系表示一个可重用的用例被另外一个用例有条件地打断，以增加其功能。什么时候使用扩展用例取决于基础用例。

在 UML 中，扩展关系表示为虚线箭头加<<extend>>字样，箭头指向被扩展的用例（即基础用例），如下图所示：





基础用例提供了一组扩展点，在这些新的扩展点中可以添加新的行为，而扩展用例提供了一组插入片段，这些片段能够被插入到基础用例的扩展点上。基础用例不必知道扩展用例的任何细节，它仅为其提供扩展点（事实上，基础用例即使没有扩展用例也是完整的，这点与包含关系有所不同）。一个用例可能有多个扩展点，每个扩展点也可以出现多次。但是一般情况下，基础用例的执行不会涉及到扩展用例，只有特定的条件发生，扩展用例才被执行。

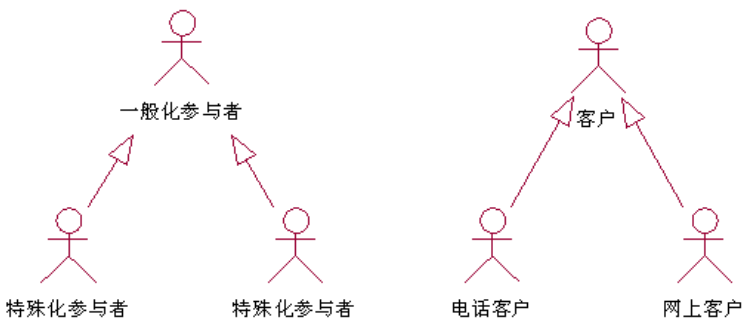
4. 泛化：

指的是参与者之间或用例之间的继承关系。

1）参与者之间的泛化关系：

在用例图中，使用参与者泛化关系来描述多个参与者之间的公共行为。如果系统中存在几个参与者，它们既扮演自身的角色，同时也扮演更具一般化的角色，那么就用泛化关系来描述它们。

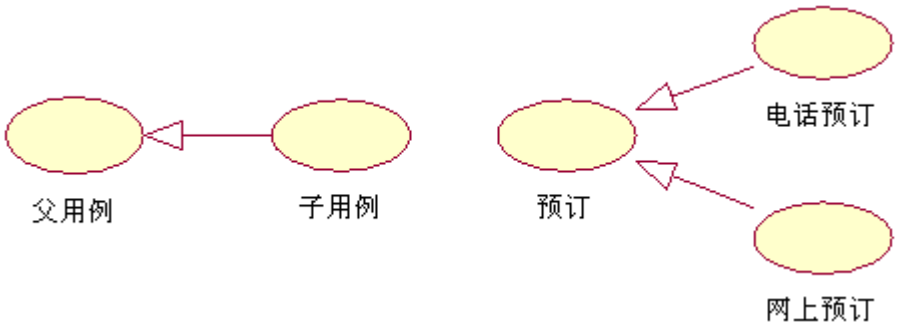
在 UML 中，参与者之间的泛化关系用一个三角箭头来表示，如下图所示：



2）用例之间的泛化关系：

一个用例可以被特别列举为一个或多个子用例，这被称作用例泛化。

在 UML 中，参与者之间的泛化关系用一个三角箭头来表示，如下图所示：



§ 2.2 用例建模

一. 普通建模技术：

1. 对系统的语境建模：

对于一个系统，会有一些事物存在于其内部，而一些事物存在于其外部。存在于系统内部的事物的任务是完成系统外部事物所期望的系统行为，存在于系统外部并与其进行交互的事物构成了系统存在的环境。所有存在于系统外部并与系统进行交互的事物构成了该系统的语境，语境定义了系统存在的环境。

在 UML 中，用用例图对系统的语境进行建模，所强调的是围绕在系统周围的参与者。对系统的语境建模，要遵循如下策略：

1）用以下几组事物来识别系统周围的参与者：

- 需要从系统中得到帮助以完成任务的组；
- 执行系统的功能时所必需的组；
- 与外部硬件或其他软件系统进行交互的组；
- 以及为了管理维护而执行某些辅助功能的组。

2）将类似的参与者组织成一般/特殊的结构层次；

3）在需要加深理解的地方，为每个参与者提供一个构造型；

4）将这些参与者放入用例图中，并说明从每个参与者到系统的用例之间的通信路径。

2. 对系统的需求建模：

需求就是根据用户对产品功能的期望，提出产品外部功能的描

述。需求分析所要做的工作是获取系统的需求，归纳系统所要实现的功能，使最终的软件产品最大限度的贴近用户的要求。

对系统的需求建模，要遵循如下策略：

1) 通过识别系统周围的参与者来建立系统的语境；
2) 对于每个参与者，考虑它期望的行为或需要系统提供的行为；
3) 把这些公共的行为命名为用例；
4) 分解公共行为，放入到新的用例中以供其他的用例使用；分解异常行为，放入新的用例中以延伸较为主要的控制流；
5) 在用例图中对这些用例、参与者以及它们的关系进行建模；
6) 用陈述非功能需求的注解修饰这些用例，可能还要把其中的一些附加到整个系统。

二. 识别参与者：

1. 如何识别参与者：

可以通过向用户提出以下几个问题来识别参与者：

1) 谁使用系统的主要功能？
2) 谁需要系统的支持以完成其日常工作任务？
3) 谁负责维护、管理并保持系统正常运行？
4) 系统需要应付（或处理）哪些硬件设备？
5) 系统需要和哪些外部系统交互？
6) 谁（或什么）对系统运行产生的结果（值）感兴趣？

2. 范例：《ATM(Auto Trade Machine)系统》

为了识别ATM系统的参与者,我们来回答前面提出的几个问题：

1) 问题 1：谁使用 ATM 系统的主要功能？
回答：储户。
2) 问题 2：谁需要 ATM 系统的支持以完成其日常工作任务？
回答：出纳员？还不确定，先放在这里。
3) 问题 3：谁负责维护、管理并保持 ATM 系统正常运行？
回答：ATM 系统工程师、银行人员。
4) 问题 4：ATM 系统需要应付（或处理）哪些硬件设备？
回答：目前还不清楚。
5) 问题 5：ATM 系统需要和哪些外部系统交互？
回答：储户身份、帐户标识卡（信用卡）。
6) 问题 6：谁（或什么）对 ATM 系统运行产生的结果（值）感兴趣？
回答：银行会计、储户。

回答了上述的几个问题，我们找到了这样一些重要的“参与者”（加引号表明有些还不一定就是最终的角色）：储户、出纳员、ATM 系统工程师和银行人员、信用卡和银行会计。

3. 注意事项：

在对参与者建模的过程中，开发人员必须牢记以下几点：

1）参与者对于系统而言总是外部的，因此它们可以处于人的控制之外；

2）参与者可以直接或间接地同系统交互，或使用系统提供的服务以完成某件事务；

3）参与者表示人和事物与系统发生交互时所扮演的角色，而不是特定的人或者特定的事物；

4）一个人或事物在与系统发生交互时，可以同时或不同时扮演多个角色；

5）每一个参与者需要一个具有业务一样的名字，在建模中不推荐使用类似于“新参与者”的名字；

6）每一个参与者必须有简短的描述，从业务角度描述参与者是什么；

7）和类一样，参与者可以具有表示参与者的属性和可以接受的事件，但使用得不频繁。

三．识别用例：

1. 如何识别用例：

为了便于识别用例，这里有几个辅助问题：

1）某个参与者要求系统为其提供什么功能？该参与者需要做哪些工作？

2）参与者需要阅读、创建、销毁、更新或存储系统中的某些（类）信息吗？

3）系统中的事件一定要告知参与者吗？参与者需要告诉系统一些什么吗？那些系统内部事件从功能的角度代表什么？

4）由于系统新功能的识别（如：那些典型的还没有实现自动化的人工系统），参与者的日常工作被简化或效率提高了吗？

上面的问题是针对已经识别的每个参与者来提的，下面的几个问题则主要针对系统：

1）系统需要什么样的输入/输出？输入来自哪里？输出去往哪里？

2）该系统的当前情况（该系统可能是人工而非自动化系统）存在有

哪些主要问题？

2. 注意事项：

一个结构良好的用例，应满足如下的要求：

1）为系统和部分系统中单个的、可标识的和合理的原子行为命名；

2）通过从它所包含的其他用例中提取公共的行为，来分解出公共行为；

3）通过把这些行为放入延伸它的其他用例中，来分解出变体；

4）清楚地描述事件流，足以使局外人轻而易举地理解；

5）是通过说明该用例的正常和变体语义的最小脚本集来描述的。

四. 参与者及用例审查：

完成了用例的识别后，需要仔细地检查一遍所做的工作和结果，针对参与者和用例可以提出如下一些检查问题：

- ✧ 是否所有的角色都存在一个用例与之关联？
- ✧ 代表共同职责的参与者间是否存在相似性？如果有，则可以抽象出基类参与者。
- ✧ 某些用例间是否具有相似性？如果有，则引入包含关系。
- ✧ 某些用例是否存在特殊的情况？如果有，则引入扩展关系。
- ✧ 存在有任何参与者或用例，它们没有相应的关联用例或关联参与者吗？如果存在的话，则一定有问题：为什么会有这些参与者（或用例）？
- ✧ 仔细审查系统，还有什么已知的功能需求未被用例处理吗？如果有，则针对这些功能需求建立新的用例。

五. 建立用例图：

在 UML 中创建用例图时，应当记住每个用例图只是一个系统的静态用例视图的图形化表示。也就是说，一个单独的用例图不必捕捉一个系统的用例视图的所有事情。将一个系统的所有用例图收集在一起，就表示该系统的完整的静态用例视图；每一个用例图只单独地表示一个方面。

一个结构良好的用例图，应满足如下的要求：

- ✧ 关注于与一个系统的静态用例视图的一个方面的通信；
- ✧ 只包含那些对于理解这方面必不可少的用例和参与者；
- ✧ 提供与它的抽象层次相一致的详细表示，只能加入那些对于理解问题必不可少的修饰（如延伸点）；
- ✧ 不应该过分简化和抽象信息，以致使读者误解重要的语义。

当绘制一张用例图时，要遵循如下的策略：

- ✧ 给出一个表达其目的的名称；
- ✧ 摆放元素时，尽量减少线的交叉；
- ✧ 从空间上组织元素，使得在语义上接近的行为和角色在物理位置上也接近；
- ✧ 使用注解和颜色作为可视化提示，以突出图的重要的特征；
- ✧ 尝试不显示太多的关系种类，一般来说，如果含有复杂的包含和延伸关系，要将这些元素放入另一张图中。

§2.3 案例分析：《图书馆管理系统》

一. 系统描述：

图书馆管理系统使对书记的借阅即读者信息进行统一管理的系统，具体包括读者的借书、还书、书籍订阅；图书馆管理员的书籍借出处理、书籍归还处理、预订信息处理；还有系统管理员的系统维护，包括增加书目、删除或更新书目、增加书籍、减少书籍、增加读者帐户信息、删除或更新读者帐户信息、书籍信息查询、读者信息查询等。系统的总体信息确定以后，就可以分析系统的参与者、确定系统用例了。

二. 确定系统的参与者：

确定系统参与者首先需要分析系统所涉及的问题领域和系统运行的主要任务：分析使用该系统主要功能的是哪些人，谁需要该系统的支持以完成其工作，还有系统的管理者与维护者。

根据图书馆管理系统的需求分析，可以确定如下几点：

- ✧ 作为一个图书馆管理系统，首先需要读者（借阅者）的参与，读者可以登录系统查询所需要的书籍，查到所需书籍后可以考虑预订，当然最重要的是借书、还书操作；
- ✧ 对于系统来说，读者发起的借书、还书等操作最终还需要图书馆管理员来处理，他们还可以负责图书的预订和预订取消；
- ✧ 对于图书馆管理系统来说，系统的维护操作也是相当重要的，维护操作主要包括增加书目、删除或更新书目、增加书籍、减少书籍等操作。

由以上分析可以得出，系统的参与者主要有 3 类：读者（借阅

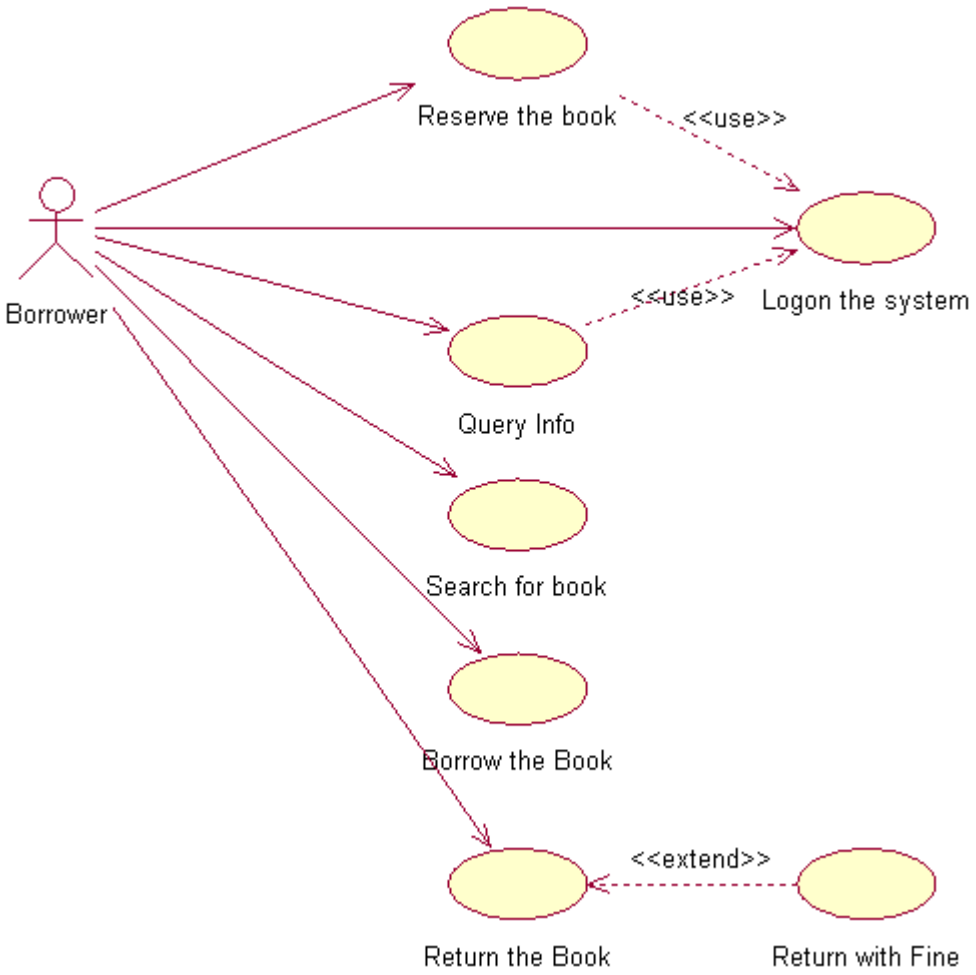
者)、图书馆管理员、图书馆关系系统维护者。

三. 确定系统用例：

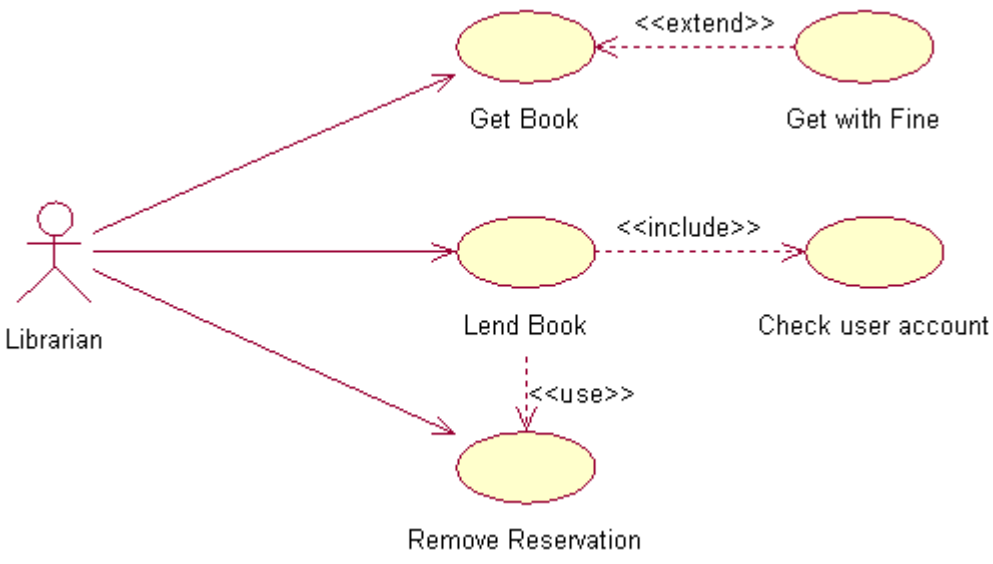
1. 借阅者请求服务的用例：
   1) 登录系统；
   2) 查询自己的借阅信息；
   3) 查询书籍信息；
   4) 预订书籍；
   5) 借阅书籍；
   6) 归还书籍。
2. 图书馆管理员处理借书、还书等的用例：
   1) 处理书籍借阅；
   2) 处理书籍归还；
   3) 删除预订信息。
3. 系统管理员进行系统维护的用例：
   1) 查询借阅者信息；
   2) 查询书籍信息；
   3) 增加书目；
   4) 删除或更新书目；
   5) 增加书籍；
   6) 删除书籍；
   7) 增加借阅者帐户；
   8) 删除或更新借阅者帐户。

四. 用例图：

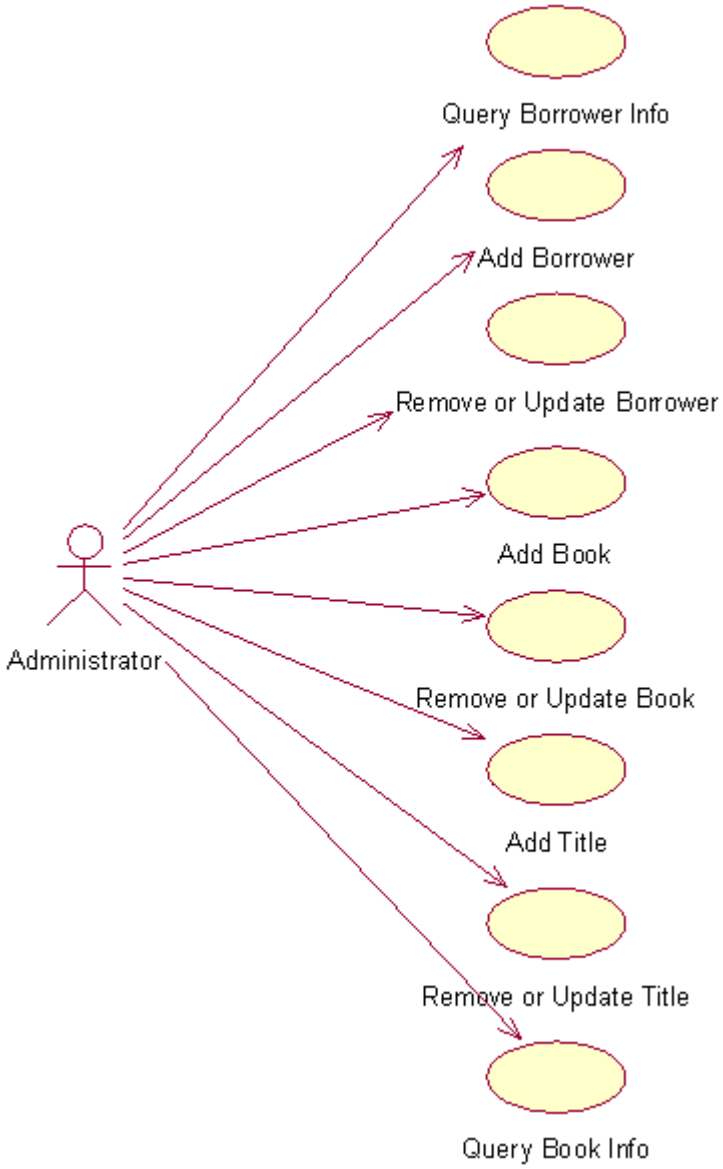
1. 借阅者请求服务的用例：



2. 图书馆管理员处理借书、还书等的用例：



3. 系统管理员进行系统维护的用例：



§2.4 补充案例：《订货中心系统》

一. 系统简介：

有这样一个订货中心，它接受客户的电话、传真、电子邮件、信件和 web 主页表单形式的订货请求，形成货物订单，并告知客户订单的价钱。根据客户要求的发货目标地点的信息，订货中心的经理以最经济的方式确定一家仓库来负责向客户发货。仓库人员收到订单后按一定的策略处理订单，发出货物，并在订单上填写所发货

物的数量信息，后把订单返回给订货中心。订货中心确认后把订单交给收费部门，由该部门负责关联客户收到货物后的付费。

客户在收到货物之前可以向订货中心查询他的订货处理情况。收到订货后，如果出现质量问题或者物品错送问题（即送的货物不是客户想要的货物），客户有权利向订货中心退货，订货中心必须接受退货，并退还用户所付款（如果用户已付款）。仓库在处理订单时由于受到库存货物有限这一现实情况的约束，因此采取一定的策略来保证那些优先级较高的订单先得到发货。

在订货中心的人工系统中，交流主要通过电话、传真，如订单传送等。在引入计算机管理后，订货中心、仓库、收费部门之间可以共享客户、订单信息，不仅省去了电话、传真的成本，同时重要的是提高了订货中心运作效率。

上面是订货中心系统的简要描述。下面将逐步地来识别参与者、描述参与者、识别用例、描述用例，最后给出完整的用例图。

二. 系统建模：

1. 参与者识别及描述：

问题 1：谁使用订货中心系统的主要功能？

回答：管理者（Manager）、发货人员（Shipper）、客户（Customer）和收款人员（TollCollector）。

问题 2：谁需要订货中心系统的支持以完成其日常工作任务？

回答：Manager、Shipper 和 TollCollector。

问题 3：谁负责维护、管理并保持订货中心系统正常运行？

回答：Manager。

问题 4：订货中心系统需要应付（或处理）哪些硬件设备？

回答：信用卡（Credit card）。

问题 5：订货中心系统需要和哪些外部系统交互？

回答：没有。

问题 6：谁（或什么）对订货中心系统运行产生的结果（值）感兴趣？

回答：Customer、Manager。

综上可得到订货中心系统的参与者：管理者（Manager）、发货人员（Shipper）、客户（Customer）、收款人员（TollCollector）和信用卡（Credit card），分别描述如下：

1）参与者：管理者（Manager）

参与者职责：

接受订货，计算价钱，选择仓库发货。

参与者识别问题：1、2、3、6。

2）参与者：发货人员（Shipper）

参与者职责：

根据订单发货给顾客，填写订单。

参与者识别问题：1、2。

3）参与者：收款人员（TollCollector）

参与者职责：

根据订单签收顾客的订货款，顾客退还商品时退款。

参与者识别问题：1、2。

4）参与者：客户（Customer）

参与者职责：

订货、退还订货、查询订单。

参与者识别问题：1、6。

5）参与者：信用卡（Credit card）

参与者职责：

电子付订货款。

参与者识别问题：4。

2. 用例识别及描述：

用例模型获取的第一步是识别参与者，通过参与者的识别可以进一步识别用例，从而得到用例模型。然而用例的识别要比参与者识别复杂得多，一方面要从系统的功能需求中抽象出用例，同时还要控制用例的数目。用例数目过多则造成用例模型过大，同时引入设计因素的可能性也加大了；用例数目过少则造成用例的粒度太大，不便于进一步分析，或者不充分。

订货中心系统是一个中等规模的系统，我们先可以不特别关注它的用例数目问题。根据前面介绍的方法，我们首先从各个参与者那里来识别用例：

1）从管理者（Manager）识别

管理者要求系统为其提供什么功能？管理者需要做哪些工作？

答：管理者要求系统提供：

- 接受顾客订货请求并创建订单；
- 计算订单价钱；
- 根据订单信息选择仓库，并将订单发送给该仓库；

- 查询订单货物发送情况；
- 查询客户订单付款情况；
- 评价商务结果；
- 客户退货处理；
- 把从仓库返回的订单发送到收费处；
- 商品价格更新。

管理者需要做：

- 生成订单；
- 查询输入订单号。

管理者需要阅读、创建、销毁、更新或存储系统中的某些（类）信息吗？

答：是，这些信息包括：

- 订单；
- 职员（仓库人员、收费人员）信息；
- 顾客信息；
- 物品条目及价格信息；
- 仓库信息；
- 税务信息。

系统中的事件一定要告知管理者吗？管理者需要告诉系统一些什么吗？那些系统内部事件从功能的角度代表什么？

答：是，这些事件包括：

- 仓库有关物品短缺以致无法满足某订单；
- 订单数据出现错误；
- 顾客超过期限未付款。

由于系统新功能的识别，管理者的日常工作被简化或效率提高了吗？

答：是。

2）从发货人员（Shipper）识别

发货人员要求系统为其提供什么功能？发货人员需要做哪些工作？

答：发货人员要求系统提供：

- 仓库存储物品的管理；
- 发货处理。

发货人员需要做：

- 从所有的订单中按顺序挑选处优先级较高的订单来发货；
- 在订单上签上发货的品名、数量。

发货人员需要阅读、创建、销毁、更新或存储系统中的某些（类）信息吗？

答：是，这些信息包括：

- 仓库存储物品信息；
- 顾客信息。

系统中的事件一定要告知发货人员吗？发货人员需要告诉系统一些什么吗？那些系统内部事件从功能的角度代表什么？

答：是，这些事件包括：

- 仓库有关物品短缺（发货人员报告）；
- 待发货订单有数据错误（发货人员报告）。

由于系统新功能的识别，发货人员的日常工作被简化或效率提高了吗？

答：是。

3）从收款人员（TollCollector）识别

收款人员要求系统为其提供什么功能？收款人员需要做哪些工作？

答：收款人员要求系统提供：

- 顾客付款处理；
- 顾客付款情况查询。

收款人员需要做：

- 签收付款；
- 确认并标注订单已付。

收款人员需要阅读、创建、销毁、更新或存储系统中的某些（类）信息吗？

答：是，这些信息包括：

- 顾客信息（阅读）；
- 顾客付款信息（阅读、创建、存储）。

系统中的事件一定要告知收款人员吗？收款人员需要告诉系统一些什么吗？那些系统内部事件从功能的角度代表什么？

答：是，这些事件包括：

- 客户逾期未付款。

由于系统新功能的识别，收款人员的日常工作被简化或效率提高

了吗？

答：是。

4）从客户（Customer）识别

客户要求系统为其提供什么功能？客户需要做哪些工作？

答：客户要求系统提供：

- 申请订货；
- 订单状况查询；
- 退货
- 付款。

客户需要做：

- 指明欲订货物的品名、数量；
- 指明自己的投递地址信息。

客户需要阅读、创建、销毁、更新或存储系统中的某些（类）信息吗？

答：不需要。

系统中的事件一定要告知客户吗？客户需要告诉系统一些什么吗？那些系统内部事件从功能的角度代表什么？

答：是，这些事件包括：

- 仓库中顾客订的货物短缺；
- 顾客逾期未付款。

由于系统新功能的识别，客户的日常工作被简化或效率提高了吗？

答：否。

5）从信用卡（Credit card）识别

信用卡要求系统为其提供什么功能？信用卡需要做哪些工作？

答：信用卡支付货款。

信用卡需要阅读、创建、销毁、更新或存储系统中的某些（类）信息吗？

答：是，这些信息包括：

- 帐户信息（阅读、更新）。

系统中的事件一定要告知信用卡吗？信用卡需要告诉系统一些什么吗？那些系统内部事件从功能的角度代表什么？

答：否。

由于系统新功能的识别，信用卡的日常工作被简化或效率提高了

吗？

答：否。

上面针对五个参与者回答了二十个问题，理清了系统的主要功能。还有两个问题是针对系统的：

系统需要什么样的输入/输出？输入来自哪里？输出去往哪里？

答：系统输入主要有：

- 新订单输入；
- 订单发货标记、订单号；
- 顾客信息。

系统输出主要有：

- 对各种查询的反馈；
- 发送的货物。

该系统的当前情况（该系统可能是人工而非自动化系统）存在有哪些主要问题？

答：存在的问题有：

- 订单在订货中心、仓库、收费部门之间的传送费时、费力；
- 订单上的数据常因为手写而不清晰或写错；
- 订货中心难于管理所有的订单；
- 订货中心难于及时地掌握订单处理的状况。

回答了上述所有问题后，我们可以整理出一系列用例，并加以描述。限于篇幅，这里将采用简便的办法来描述。

用例及其描述：

1）用例 1：增加订单（Add order）

顾客提出订货请求；

订货中心记录下顾客的信息，包括姓名、通讯地址、电话；

计算订单价钱并告诉顾客；

顾客签名确认。

扩展：

4．直接使用信用卡付款：信用卡付款（用例 5）。

包含：

计算订单价钱（用例 2）；

订货中心选择仓库发送订单（Select warehouse）；

订货中心把订单发送到收费处（Send to Billdepartment）。

2）用例 2：计算订单价钱（Calculate order price）

统计货品数目、价格；

计算应缴税额；

累加得到的订单价钱；

3）用例 3：发货（Ship order）

仓库按照一定优先级策略挑选订单；

按订单注明的信息发货；

仓库在已发货订单上签注已发货信息；

把订单返回到订货中心。

扩展：

2．仓库缺货：订货中心选择一个仓库发送订单（Select warehouse）。

4）用例 4：付款处理（Bill order）

签收订单货款；

发送发票给顾客；

把签收的订单返回到订货中心。

扩展：

1．顾客使用信用卡付款：信用卡付款（用例 5）。

5）用例 5：信用卡付款（Bill by credit card）

识别信用卡；

识别用户名和密码；

从信用卡上取出订单注明货款到订货中心账号。

扩展：

3．信用卡账号余额不足：其他方式付款。

6）用例 6：评价商务结果（Assess business results）

评价仓库工作情况；

评价职员工作情况；

评价顾客服务情况；

评价经营状况。

7）用例 7：退货服务（Order returning Service）

顾客提出退货要求。

扩展：

2a．顾客退货请求被否决：回绝退货请求（Reject service）；

2b．顾客退货请求被认可：处理顾客退货（用例 8）。

8）用例 8：处理顾客退货（Order returning processing）

标记原订单为退货订单；

货物处理。

扩展：

2a．如果顾客愿意更换货物：增加订单（用例 1）；

2b．顾客要求退款：退款处理。

9）用例 9：查询订单情况（Enquiry order）

顾客提出查询请求；

获取顾客订单号；

查询订单发货情况；

查询订单付款情况。

10）用例 10：维护（Maintenance）

职员信息维护；

仓库信息维护；

加入一个新顾客。

包含：

4．物品信息维护（用例 11）；

5．税务信息维护（用例 12）；

11）用例 11：物品信息维护（Maintenance goods items）

新增物品；

取消物品；

更改物品价格。

扩展：

1．税务信息库中没有该物品的税务记录：加入该物品的税率数据；

2．订货中心不再经营该产品：取消该物品的税率；

12）用例 12：税务信息维护（Maintenance goods taxes）

加入新物品的税务信息；

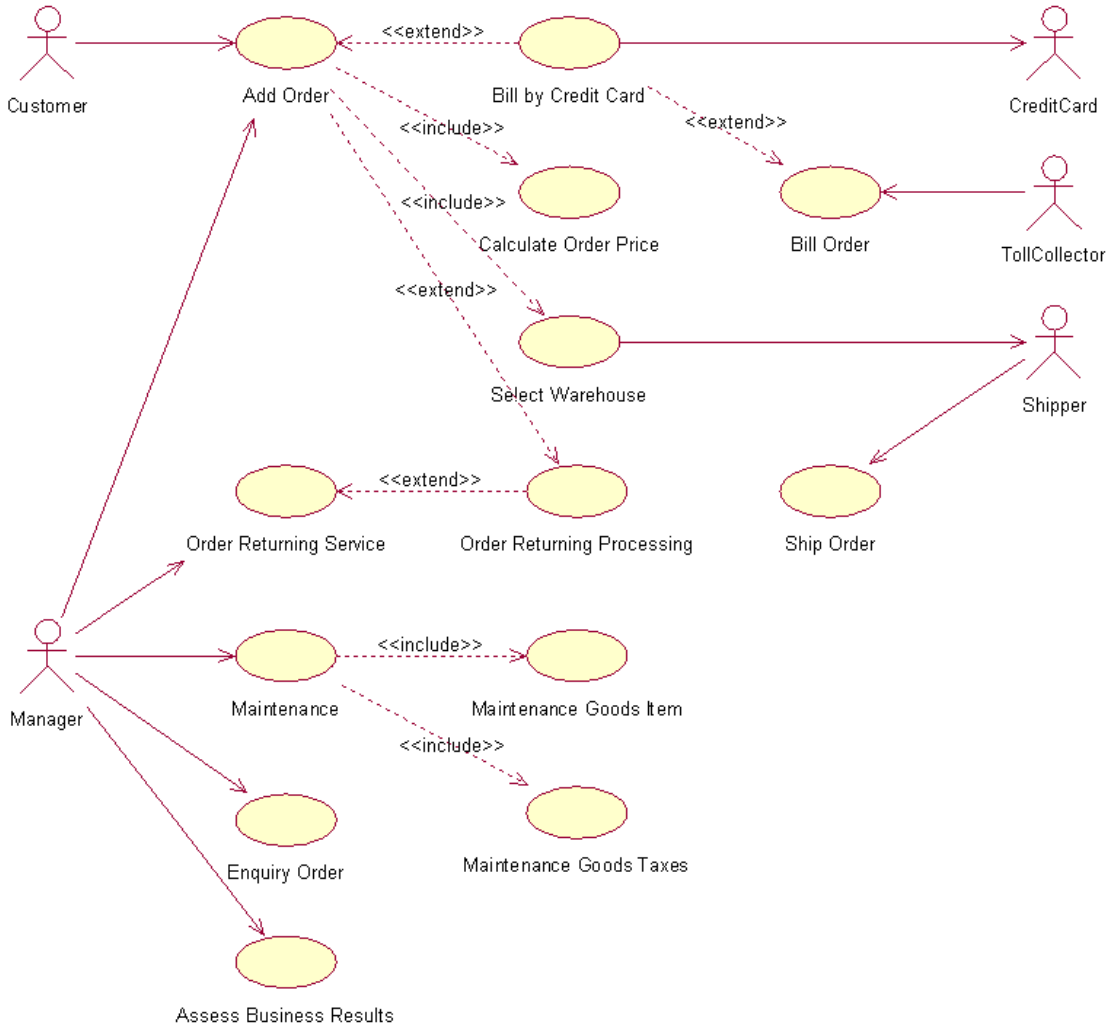
更改物品的税务信息；

取消物品的税务信息。

有些用例可以在设计时再描述它，分析时可以不去描述，如退款处理和加入新物品的税率等用例。另外，在设计阶段我们可能还对某些用例进行细化，如维护和评价商务结果等用例。在建立模型的时候，往往会得到很多用例，而且某些参与者会驱动很多用例等，这时把所有参与者和用例都画在一张图上会使整个图的清晰度降

低，因此需引入包机制来管理众多的用例。

3. 用例模型：



三. 建模体会：

在分析用例的时候一定要记住，某个功能是否作为一个用例单独存在，一方面要看是否有相应的参与者需要该项功能，另外还要分析该项功能是否是系统为了满足系统、商务客户（business customer）的商务请求（business requirements）而设置，如果是的话，就要考虑作为某个用例的一个活动步骤。

§ 2.5 小结

第三章　类图和对象图

内容概要

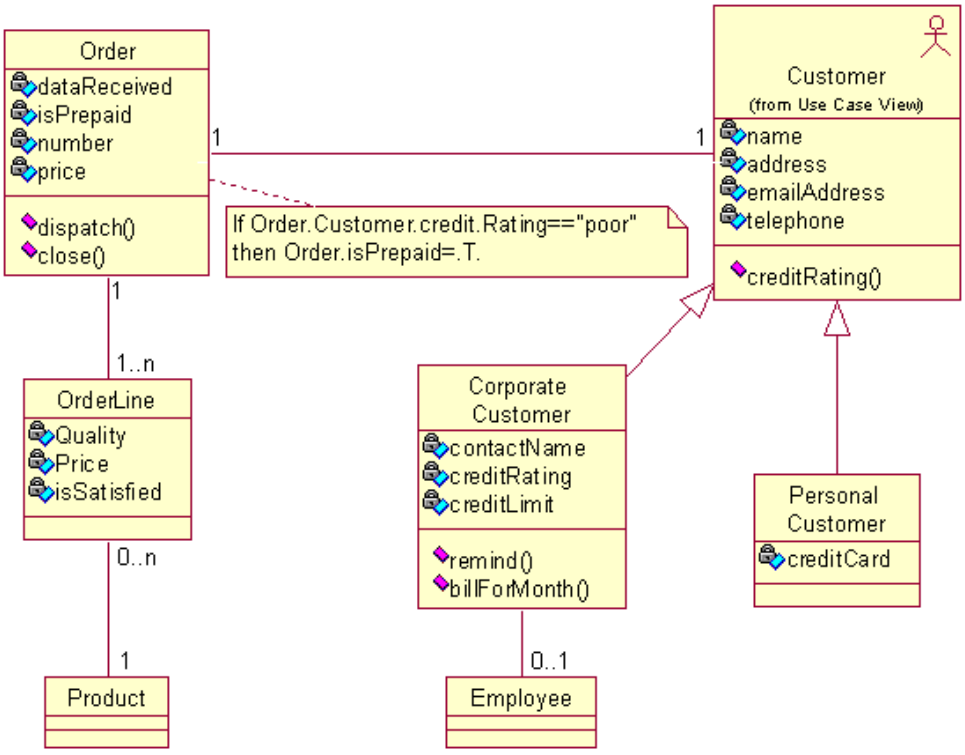
- ✧ 基本概念
- ✧ 静态结构建模
- ✧ 案例分析

§3.1 基本概念

一. 类图：

类图是描述类、接口、协作以及它们之间关系的图，用来显示系统中各个类的静态结构。

- ✧ 类图描述系统中类的静态结构，它不仅定义系统中的类，描述类之间的联系，如：关联、依赖、聚合等，还包括类的内部结构（类的属性和操作）；
- ✧ 类图描述的是一种静态关系，在系统的整个生命周期中都是有效的。



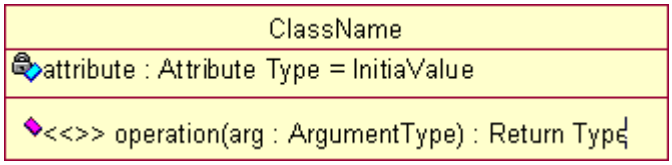
1. 类：

是面向对象系统组织结构的核心，是对一组具有相同属性、操作、关系和语义的对象的描述。

✧ 类定义了一组有着状态和行为的对象。其中，属性和关联用来描述状态；行为由操作来描述，方法是操作的实现。对象的生命周期则由附加给类的状态机来描述。

1）符号：

在 UML 中，类用矩形来表示，并且该矩形被划分为 3 个部分：名称、属性和操作。



2）名称：路径名::简单名

3）属性：是类的一个组成部分，也是一个特性，描述了类在软件系统中代表的事物（即对象）所具备的特性，这些特性是所有的对象所共有的。类的属性是类的信息包含，它可以确定并区分对象以及对象的状态。一个属性一般都描述类的某个特征，因此可以用来识别某个对象。类可以有任意数目的属性，也可以没有属性。

- 在 UML 中，类属性的语法为：
  [可见性] 属性名 [：类型] [=初始值] [{约束特性}]
- 在 UML 中，公有类型用“＋”表示，私有类型用“－”表示，受保护类型用“＃”表示。
- 按照 UML 的约定，单字属性名小写，如果属性名包含了多个单词，这些单词要合并，且除了第一个单词外其余单词的首字母要大写。

4）操作：是对类的对象所能做的事务的抽象，相当于一个服务的实现，且该服务可以由类的任何对象请求以影响其行为。

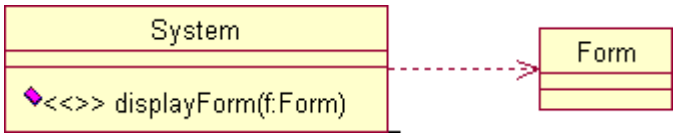
- 在 UML 中，类操作的语法为：
  [可见性] 操作名 [(参数表)] [：返回类型] [{约束特性}]

5）职责：是类或其他元素的契约或义务，类的职责是自由形式的文本，写成一个短语、一个句子或一段短文，在 UML 中，把职责列在类图底部的分隔栏中。

2. 类之间的关系：

1）依赖关系：

依赖表示两个或多个模型元素之间语义上的关系，它表示了这样一种情形，对于一个元素（提供者）的某些改变可能会影响或提供消息给其他元素（客户），即客户以某种形式依赖于其他类元。



- 根据这个定义，关联、泛化和实现都是依赖关系，但是它们有更特别的语义，所以在 UML 中被分离出来作为独立的关系。
- 在 UML 中，依赖用一个从客户指向提供者的虚箭头表示，用一个构造型的关键字来区分它的种类。

2）关联关系

关联是类之间的一种连接关系，通常是一种双向关系。从语义角度来说，关联的类之间要求有一种手段使得它们彼此能够索引到对方，这种手段就是关联。

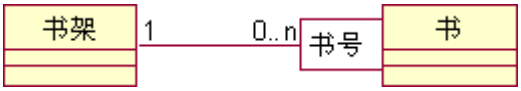
- 关联的表示：

  UML 中，关联用一根连接类的实线来表示。

- 关联的修饰：
  - 名称：关联的名称并不是必需的，只有需要明确地给关联提供角色名，或一个模型存在很多关联且要查阅、区别这些关联时，才有必要给出关联名称。
  - 角色：是关联关系中一个类对另一个类所表现出来的职责。类在参与关联时，相应于不同的关联它所体现的职责（或者担当的角色）往往是不同的，而不同的类所体现的职责也不同。为了更准确地描述关联，我们可以把类在关联中体现的职责标识出来。
  - 导航：描述的是一个对象通过链（关联的实例）进行导航访问另一个对象，即对一个关联端点设置导航属性意味着本端的对象可以被另一端的对象访问。如果模型中关联不指明方向，则默认关联是双向的。
  - 多重性：是指有多少对象可以参与该关联，多重性可以用来表达一个取值范围、特定值、无限定的范围或一组离散

值。多重性是关联对于关联的类的对象在参与关联时的数目约束。

- 关联的限定：在一对多或多对多的关联中，从多重性为 1 的对象（或者多对多关联的任意一个对象）中找到它所关联的类的对象通常是很困难的，因此在这种情况下人们就寻求一种方法来克服这个问题，这种方法就是在关联中加入限定成分，也就是候选键。通过这个键可以惟一地识别多重性为多一端的所有对象，这样对于这些对象的寻找（定位）就会变得容易和有效。

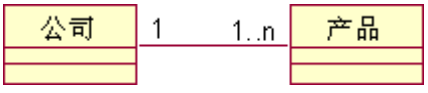


■ 关联的语言实现：

- 多重性为 1 对 1 的关联：

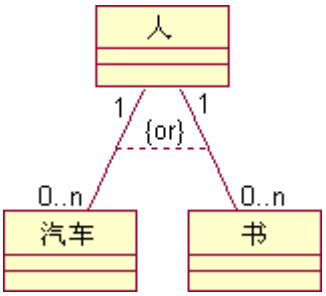

```
public class Programmer{
  public:
      //程序员类的公有属性和操作
  void SetUsedComputer(Computer * computer);
  Computer * GetUsedComputer(void);

  private:
      //程序员类的私有属性和操作
  Computer * UsedComputer;
};
public class Computer{
  public:
      //计算机类的公有属性和操作
  Void     SetOwnerProgrammer(Programmer*
  programmer);
  Programmer * GetOwnerProgrammer(void);

  private:
      //计算机类的私有属性和操作
  Programmer * OwnerProgrammer;
}
```

- 多重性为 1 对多的关联：



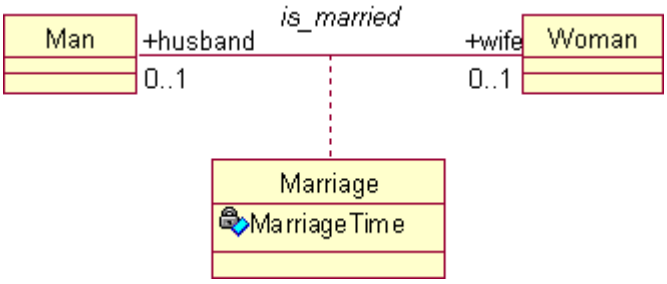
```
public class Company{
  public:
      //公司类的公有属性和操作
  void SetProduct(Product * product);
  Product * * GetProduct(void);

  private:
      //公司类的私有属性和操作
  Product * [ ] productAry; //指针动态数组
};
public class Product {
  public:
      //产品类的公有属性和操作
  Void     SetOwnerCompany(Company * company);
  Company * GetOwnerCompany();

  private:
      //产品类的私有属性和操作
  Company * OwnerCompany;
}
```

- 关联的其他情况：
  - 递归关联：关联的两个类实际上是同一个类，也就是一个类到自身的关联。递归关联表达的仍然是对象间的连接关系，只不过关联的对象是同一个类的不同实例。
  - 多重关联：在实践中可能会遇到一个类，同时与多个类具有关联关系，称为多重关联。有时一个对象只能同时与一个类的对象关联，而不能与另外一个类的对象关联，这时就需要对这两个关联进行限制，使得它们不能同时存在于一个对象上，对于这种情况，UML 使用关联的“或”限定这样的两个（或多个）关联同时存在于一个对象身上。



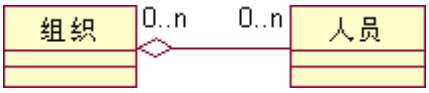
  - 关联类：有时由于系统本身的需要，需要关注那些属于关联自身的数据或操作，这时如果把这些数据或操作放在任何关联的类中会发现都不合适，因为它们描述的是关联，

而不是关联的类。为此，UML 引入了关联类，从本质上说，关联类和普通类没有什么区别，它可以具备普通的类所具有的全部特征，甚至还可以与另外的类有关联关系。



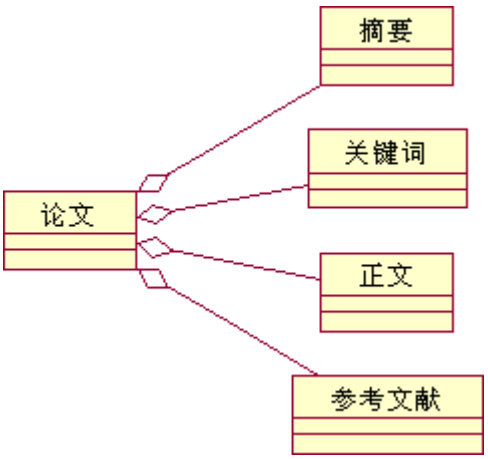
使用关联类时要格外谨慎小心，其中重要的一条原则就是任何两个对象如果具有关联关系，并且相应的关联类有对象实例，那么这个关联对象实例一定是惟一的，不能再有第二个，即对于两个确定的关联对象而言，关联类不能同时拥有两个关联对象实例。同时，一旦这两个对象间的关联关系被撤销时，与此关联关系对应的关联对象实例也将被撤销，变为不可见。

- 特殊的关联：
  - 聚合：是一种特殊类型的关联，它表示整体与部分关系的关联。



如果聚合体不存在或被撤销了，那么它的聚合元素还可继续存在。

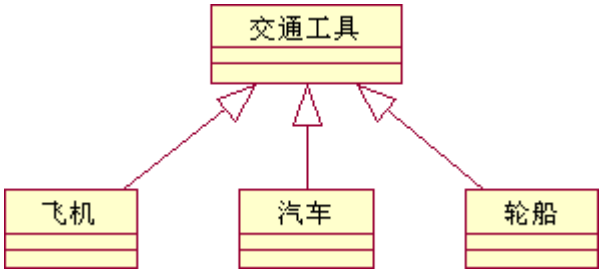
  - 组合：是聚合关系的一种特殊情况，是更强形式的聚合，又被称为强聚合。在组合中，成员对象的生命周期取决于聚合的生命周期，聚合不仅控制着成员对象的行为，而且控制着成员对象的创建和解构。



整体类拥有部分类，一旦整体对象不存在，部分对象也将不存在，整体对象与部分对象之间具有共存共亡的关系。

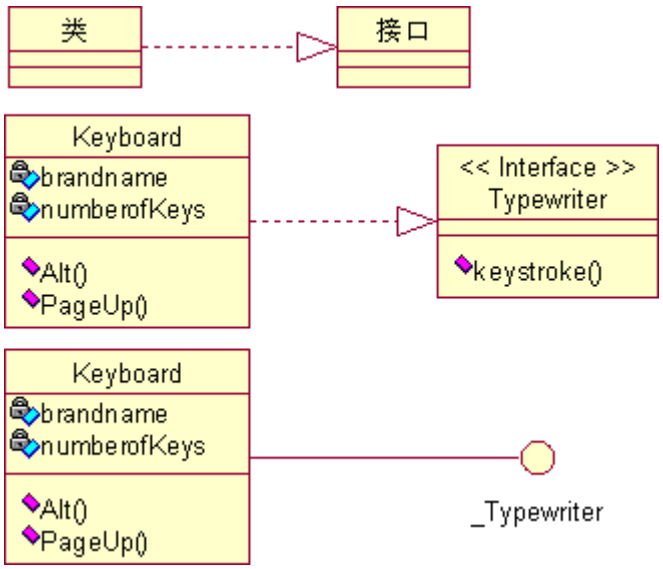
3）泛化关系

在 UML 中，泛化被定义为一个在抽象级别上更一般的元素与一个更具体的元素间的类属关系，并且该更具体的元素与更一般的元素外部行为上保持一致，而且还包含一些附加的信息，同时，更具体的类的实例可以替换所有更一般的类的实例在所有场合的出现。



4）实现关系

是规格说明和其实现之间的关系，它将一种模型元素与另一种模型元素连接起来，比如：类和接口。

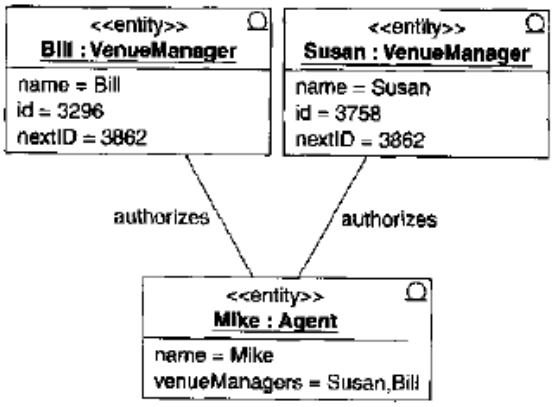


- 实现关系通常在两种情况下被使用：在接口与实现该接口的类之间；在用例以及实现该用例的协作之间。
- 在 UML 中，实现关系的符号与泛化关系的符号类似，用一条带指向接口的空心三角箭头的虚线表示。
- 实现关系还有一种省略的表示方法，即将接口表示为一个小

圆圈，并和实现接口的类用一条线段连接。

二. 对象图：

对象图描述的是参与交互的各个对象在交互过程中某一时刻的状态。对象图被看作是类图在某一时刻的实例。



三. 类图和对象图的区别：

类图	对象图
类具有 3 个分栏：名称、属性和操作	对象只有两个分栏：名称和属性
在类的名称分栏中只有类名	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”
类的属性分栏定义了所有属性的特征	对象则只定义了属性的当前值，以便用于测试用例或例子中
类中列出了操作	对象图中不包含操作，因为对于属于同一个类的对象而言，其操作是相同的
类使用关联连接，关联使用名称、角色、多重性以及约束等特征定义。类代表的是对对象的分类，所以必须说明可以参与关联的对象的数目	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

§3.2 静态结构建模

一. 类的识别：

- ✧ 长期以来，人们（特别是程序员）认识世界时总是不自觉地运用功能分解技术，因此他们看待一个实体世界时，充斥于头脑中的往往是一个个功能，而不是一个个有机的实体对象。
- ✧ UML 的最终目标是识别出所有必须的类来，并分析这些类之间的关系，从而通过编程语言来实现这些类，并最终实现整个系统。
- ✧ 类的识别是贯穿整个 OO 开发过程中的一个重要活动；在分析阶段，主要识别问题域相关的类；在设计阶段，则需加入一些反映

设计思想、方法的类以及实现问题域类所需的类等；在编码实现阶段，因为语言的特点，可能还有加入一些其他的类。

1. 名词识别方法：这种方法的关键就是识别系统问题域中的实体，一般问题域中的实体的描述都会以名词或名词短语形式出现。

   名词识别方法的过程可总结为如下步骤：

   1）使用开发组指定的语言，和系统问题域专家合作完成一个对系统的简要描述，描述应该使用问题域中的概念和命名，系统描述一般控制在一页纸以内，要求按照系统本身的流程依次把所以参与该系统的实体都包括在描述之内，不要求详细定义、描述系统的流程，因为该描述是专为分析员服务的，但是要求一定要完整地描述系统流程。

   2）从系统描述中表示名词、代词、名词短语。其中单数名词（代词）往往可识别为对象；而复数名词则往往可以识别为类。

2. 系统实体识别方法：（所谓实体，就是具有一定功能(左右)和状态保持能力的信息体）是一种自然、直接的类识别方法，它并不关心系统的运作流程，也不关心实体间的通讯状况，这种方法往往从经验的角度考虑系统中的人员、组织、地点、表格、报告等实体，得到了一系列系统实例后，经过分析可以将它们识别为类（或对象）。

   一般地，可以运用下面的策略来识别实体：

   1）系统需要存储、分析和使用的信息实体，或者要求有不同一般的处理逻辑的信息实体；

   2）系统内部需要处理的设备；

   3）与系统交互的外部系统；

   4）系统相关的人员（以职责形式命名），如：使用者、客户、系统操作员等；

   5）系统的组织实体，如：销售部门、收费部门、计划部门等。

3. 分解技术：分解技术的依据是人们有时按照从整体到部分的原则来认识系统，早先识别的某些类本身反映的可能是那些“大”的实体，使用分解技术对这些“大”的类进行分解，可以得到一系列较为“小”的类。

4. 抽象技术：通常分析员会发现所识别的类中，有些具有一定的相似性，根据这种相似性建立抽象类，并在抽象类与这些类之间建立基础关系。

5. 重用：重用是解决问题的最经济、最有效的办法，也是受益最多的

办法。一方面，重用的对象往往是那些已经实现的、经过实践检验的系统元素，其可用性和可靠性往往都比较高，通过重用可以明显地改善系统的质量；另一方面，通过重用节约了开发工作量，减少了开发时间，也降低了系统成本。

6. 从用例中识别类：

针对各个用例，可以提如下辅助识别问题

1）用例描述中出现了哪些实体？或者用例的完成需要哪些实体合作？

2）用例执行过程中会产生并存储哪些信息？

3）用例要求与之关联的每个角色的输入是什么？

4）用例反馈与之关联的每个角色的输出是什么？

5）用例需要操作哪些硬设备？

二. 类属性的识别：

为了有效地识别属性，我们可以采用如下策略：

1. 首先从类的语义完整性角度列举出类的候选属性；

所谓类的语义完整性是指类属性能够在一起完整地描述一个类所具有的特性和特征。

2. 针对系统目标和类在系统中的作用以及问题域相关特性对类的候选属性进行一次初步筛选；

3. 运用 Rumbaugh 提出的 7 条原则来检查每个候选属性，从而确定属性；

Rumbaugh 的 7 条检查原则从属性的多个方面来检查某个候选属性是否能最终确定为类的属性：

1）是属性还是一个类？

如果某个属性在问题域中不仅仅是一个值，它有其独立存在的必要，那么该属性应该被识别为一个类，并在当前的类中建立一个属性连接（关联或聚合）这个类。

2）属性存在是否合适？

如果某个属性的值依赖于某种特定的系统环境，那么就需要考虑该属性存在的合理性。

3）名字问题。

如果名字不依赖于特定的上下文环境就应该是一个属性。

4）标识符。

一定不要把编程语言为了准确地访问对象而赋予的惟一的

表示作为类的属性。（然而，如果在问题域中类具有一个标识，记应该作为一个属性）

5）链接属性。

如果某个候选属性依赖于一个类之间的链接而存在，那么该候选属性应该是链接（关联类）的属性，而不是关联的类的属性。

6）细节详细程度。

不要太专注于哪些详细细节属性，只要这些属性不影响类的行为和能力，就可以忽略它。

7）不一致的属性。

如果一个候选属性看起来与类的其他属性完全不相关，就可能意味着需要另外创建一个类，并把所有这些不相关的属性放到新创建的类中。应该牢记，任何一个类都应该是一致的、简单的，当然也应该是完整的（在问题域意义下）。

4. 运用 Shlaer 和 Mellor 推荐的 6 个属性原则来复查属性，并进行必要的调整。

通过检查的属性通常还要做进一步的复查，单复查并不是针对每个具体的属性，而是从属性应该遵守的约定原则进行。属性应该遵守的约定原则共有 6 条：

1）属性应该捕捉属性所属的类在问题域中的特征；

2）属性在任何时刻只能有一个在其运行范围内的确切的值；

3）属性反映的是完整实体的一个特征，而不是该实体的一个组成部分的特征；

4）属性一定不能包含一个内部结构；

5）如果一个抽象概念的类要与其他类（典型如那些实体类）交互，该抽象概念类的属性一定只与这个抽象概念相关，而与其他类无关；

6）如果一个类与另一个类之间具有一致关系，特别地这两个类拥有一个共同的超类（这两个类具有兄弟关系），则这个类中的属性一定是捕捉该类本身的特征，而不是这两个类之间的关系或另外一个类的特征。

三. 类操作的识别：

在识别操作时，我们可以通过回答下述五个问题完成：

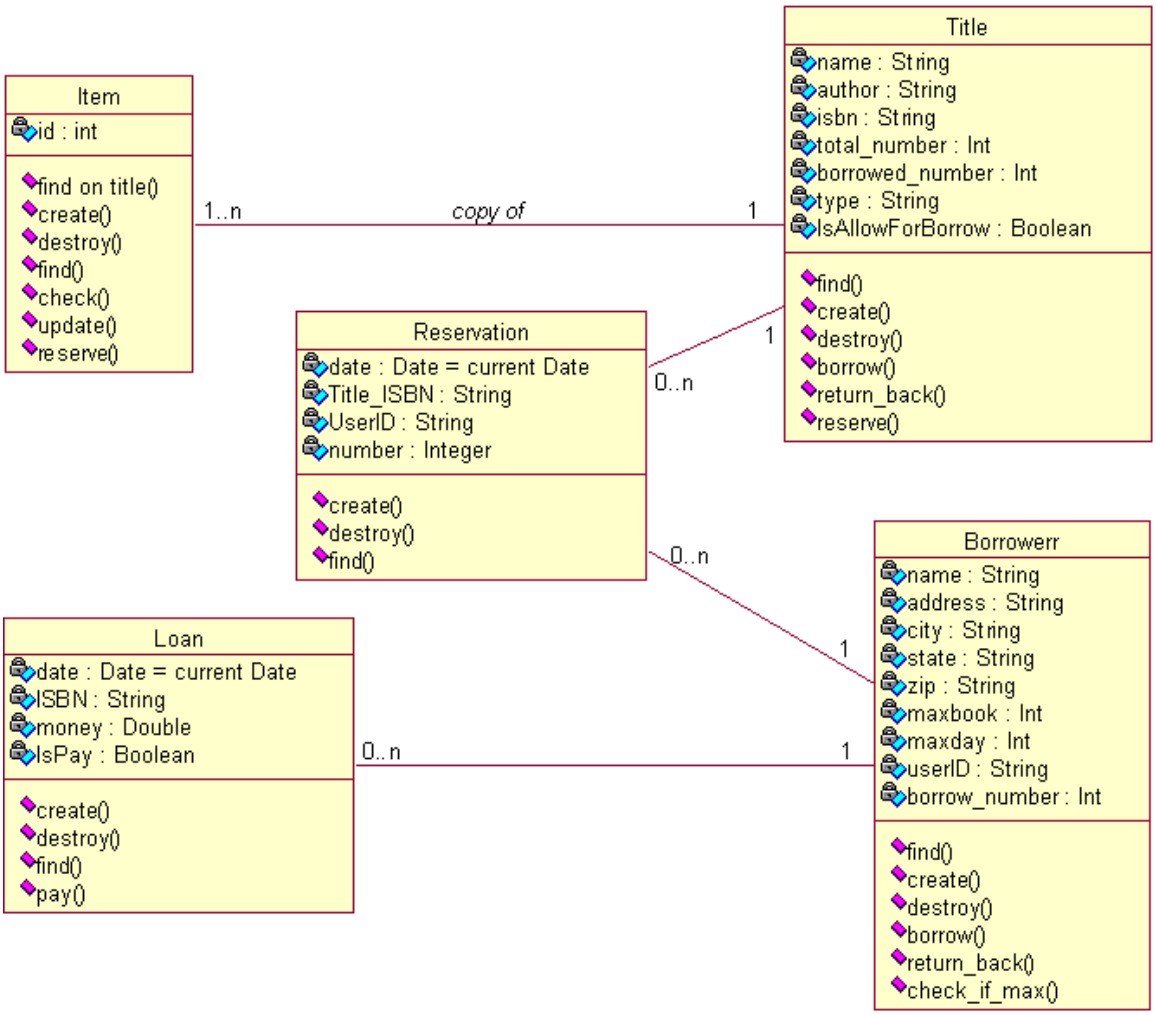
1. 有那些类会与该类交互（包括该类本身）？
2. 所有与该类具有交互行为的类会发送那些消息给该类？该类又会发

送哪些消息给这些类？

3. 该类如何响应别的类（也包括自身）发送来的消息？在发送消息出去之前，该类需要做何处理？
4. 从该类本身来说，它应该有哪些操作来维持其信息的更新、一致性和完整性？
5. 系统是否要求该类具有另外的一些职责（典型地并不体现在与类之间的交互中）？

§3.3 案例分析：《图书馆管理系统》

图书馆管理系统的静态模型如下：（类图）



§3.4 小结

§3.5 补充实例

第四章　状态图

§4.1 UML 动态建模机制

静态模型定义并描述了系统的结构和组成，动态模型的任务就是定义并描述系统结构元素的动态特性及行为。

UML 动态模型包括：状态模型、顺序模型、协作模型和活动模型，通常以状态图(State diagram)、顺序图(Sequence diagram)、协作图(Collaboration diagram)和活动图(Activity diagram)来表示。

- ✧ 状态模型关注一个对象的生命周期内的状态及状态变迁，以及引起状态变迁的事件和对象在状态中的动作等，因此状态模型常用于描述具有复杂控制逻辑的对象，如界面对象和控制对象等；
- ✧ 顺序模型和协作模型都强调对象间的协作关系，通过对象间的消息传递以完成系统的用例，不同的是顺序图强调对象交互的时间特性，而协作图则强调的是对象的协作视图；
- ✧ 活动图用于描述多个对象在交互时所采取的活动，它关注对象如何相互活动以完成一个事务。

§4.2 基本概念

一. 状态机：

状态机**(state machine)**是一个行为，它说明对象在它的生命期中响应事件所经历的状态序列以及它们对那些事件的响应。

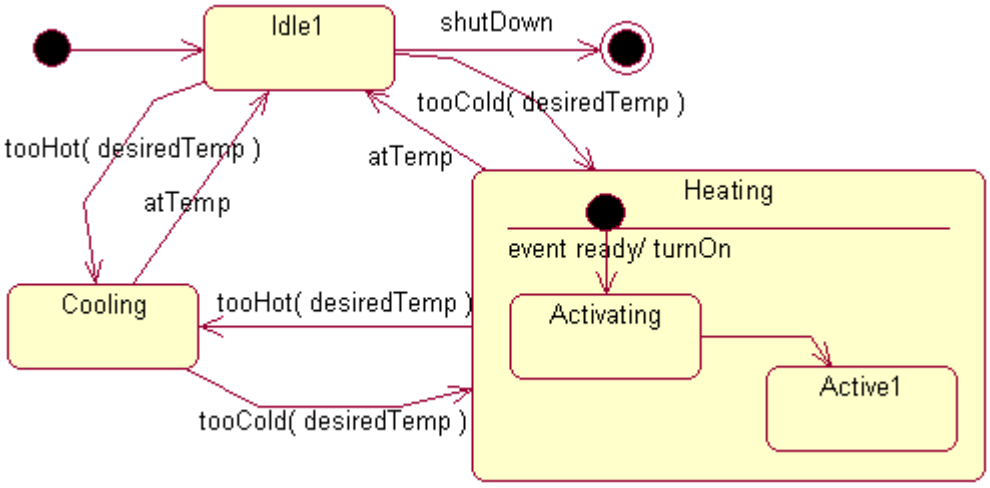
- ✧ 使用交互，可以对共同工作的对象群体的行为建模；使用状态机，可以对单个的对象的行为建模。
- ✧ 状态机用于对系统的动态方面建模。状态机包含了一个类的对象在其生命期间所有状态的序列以及对象对接收到的事件所产生的反应。
- ✧ 状态机是一个类的对象所有可能的生命历程的模型。对象被孤立地从系统中抽出和考察，任何来自外部的影响被概述为事件，当对象探测到一个事件后，它依照当前的状态做出反应，反应包括执行一个动作和转换到新状态。
- ✧ 状态机是一个对象的局部视图，一个将对象与其外部世界分离开

来并独立考查其行为的图。

- ✧ 利用状态机可以精确地描述行为：从对象的初始状态起，开始响应事件并执行某些动作，这些事件引起状态的转换；对象在新的状态下又开始响应和执行动作，如此连续进行直到终结状态。但不适合综合理解系统执行操作，如果要更好地理解整个系统范围内的行为产生的影响，那么交互视图将更有用些。
- ✧ 完整的状态机是一个可递归分解为子状态的组成状态。

二. 状态图：

是显示一个状态机（包括简单状态、转换、嵌套组成状态）的图，是系统分析的一种常用工具，它通过建立类对象的生存周期模型来描述对象随时间变化的动态行为。



- ✧ 一个状态图表示一个状态机，主要用于表现从一个状态到另一个状态的控制流。它不仅可以展现一个对象拥有的状态，还可以说明事件（如：消息的接收、错误、条件变更等）如何随着时间的推移来影响这些状态。
- ✧ 状态图用以捕获并识别一个对象、子系统或整个系统在其生命周期内的状态空间，并且指出在外界激励出现时，其状态的变化情况。
- ✧ 状态图展示对象所具有的所有可能的状态和状态间迁移的充分条件（如条件和转移前提）。
- ✧ 在面向对象分析与设计中，对象的状态、状态的转换、触发状态转换的事件、对象对事件的响应（即事件的行为）都可以用状态图来描述。

三. 事件：

事件是一个在时间和空间上占有一定位置的有意义的事情的规

格说明。

- ✧ 事件在时间上的一点发生，没有持续时间，如果某一事情的发生造成了影响，那么在状态机模型中它是一个事件。
- ✧ 在状态机的语境中，一个事件是一次激发的产生，激发能够触发一个状态转换。
- ✧ 事件可以是内部的或外部的，外部的事件是在系统和它的参与者之间传送的事件（例如：一个按钮的按下）；内部事件是在系统内部的对象之间传送的事件（例如：溢出异常）。
- ✧ 事件是对象状态变化的一个重要前提，如果没有相应的事件发生，状态通常就会维持不变。
- ✧ 在问题域中事件表现为消息，如：按钮的按下、开关的打开与关闭等。

事件可以分成明确或隐含的几种：

1. 信号事件：

信号是作为两个对象之间的通信媒介的命名的实体，信号的接收是信号接受对象的一个事件。

- ✧ 一个信号表示由一个对象异步地发送，并由另一对象接收的一个已命名的对象。
- ✧ 一个信号可以作为状态机中一个状态转换的动作而被发送，或者作为交互中一个消息的发送而被发送。
- ✧ 信号发送对象明确地创建并初始化一个信号实例并把它发送到一个或一组对象。

2. 调用事件：

调用事件是一个对象对调用的接收，这个对象用状态的转换而不是固定的处理过程实现操作。

- ✧ 一个调用事件代表一个操作的调度。
- ✧ 信号是一个异步事件，而调用事件一般来说是同步的。
- ✧ 当一个对象调用另一个具有状态机的对象的一个操作时，控制就从发送者传送到接受者，该事件触发转换，一旦调用的接收对象通过由事件触发的转换完成了对调用事件的处理（或调用失败而没有进行任何状态转换），则控制返回到调用对象，接受者转换到一个新的状态。与普通的调用不同，调用事件的接收者会继续它自己的执行过程，与调用处于并行状态。

3. 变化事件：

变化事件是表示状态中的一个变化或某些条件的满足的事件。

4. 时间事件：

时间事件代表时间的流逝，是表示一段时间推移的事件。

✧ 时间事件既可以被指定为绝对形式（天数），也可以被指定为相对形式（从某一指定事件发生开始所经历的时间）。

四. 状态：

是指在对象的生命期中满足某些条件、执行某些活动或等待某些事件时的一个条件或状况。状态是状态机的重要组成部分，它描述了状态机所建模对象的动态行为产生的结果，状态描述了一个类对象生命期中的一个时间段。

✧ 状态是一种存在状况，它通常具有一定的时间稳定性，即在一段时间内保持对象（或系统）的外在情况和内在特性的相对稳定。

✧ 状态描述了一个类对象生命期中的一个时间段，它可以用三种附加方式说明：

- 在某些方面性质相似的一组对象值；
- 一个对象等待一些事件发生时的一段时间；
- 对象执行持续活动时的一段时间。

✧ 状态通常是匿名的，并仅用处于该状态时对象进行的活动描述。

✧ 状态具有两层含义：

- 一层是对象的外在状况，如：电视的开与否，汽车是否在行驶等；
- 另一层则是对象的内在特性，如：电视对象中的“开关”属性的值，汽车对象“行驶指示器”属性的值等。

对象的外在状况是由其内在特性（属性值）所确定的，而对象的内在特性还包括与其他对象的连接情况（当然最终还是通过相关的属性值体现出来）。

✧ 事实上，类对象的任何一个属性值都是一个状态，全部的状态构成一个庞大的状态空间。在对系统建模时，我们只关心哪些明显影响对象行为的属性，以及由他们表达的对象状态，而不理睬那些与对象行为无关的状态。

1. 状态的表示：

在 UML 中，状态以带四个圆角的矩形框表示，其中黑色圆圈表示初始状态，而形状若牛眼般，未完全涂黑的圆圈表示对象的终结状态。

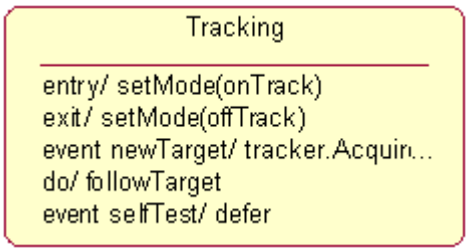
2. 状态的组成：

1）名称：

一个可以把该状态和其他状态区分开的字符串；状态可能是匿名的，即没有名称。

2）进入（入口）和退出（出口）动作：

分别为进入和退出某个状态时所执行的动作。



- 进入和退出动作不可以有参数或监护条件。然而，位于一个类的状态机的顶层的进入动作可以有参数，用来表示当创建该对象时状态机接收到的参数。
- 入口动作通常用来执行进入状态所需要的内部初始化，因此不能回避一个入口动作，任何状态内的动作在执行前都可以假定状态的初始化工作已经完成，不需要考虑如何进入这个状态。
- 无论何时从一个状态离开都要执行一个出口动作来进行善后处理工作，当出现代表错误情况的高层转换嵌套状态异常终止时，出口动作特别有用，出口动作可以处理这种情况以使对象的状态保持前后一致。

3）内部转换：

是不导致状态改变的转换，即当处于一个状态内，在不离开该状态的情况下处理事件。

- 内部转换有一个源状态但是没有目标状态，因此转换激发的结构不改变本状态。
- 内部转换的激发规则和改变状态的转换的激发规则相同。
- 如果一个内部转换带有动作，它也要被执行，但是没有状态改变发生，因此不需要执行入口和出口动作。
- 内部转换可以有带参数和监护条件的事件。所以，内部转换实质上是中断。

4）初态和终态：

- 初态，表示该状态机或子状态的缺省开始位置。一个初态用一个实心的圆表示。

- 终态，表示该状态机或外围状态的执行已经完成。一个终态用一个内部含有一个实心圆的圆圈表示。
- 初态和终态实际上都是伪状态。它们除了名称外，不具有正规状态的通常部分。

5）简单状态：

是指不包含其他状态的状态。简单状态没有子结构，但它可以具有内部转换、进入动作和退出动作等。

6）组合状态::

一个含有子状态（即嵌套状态）的状态被称作组合状态。

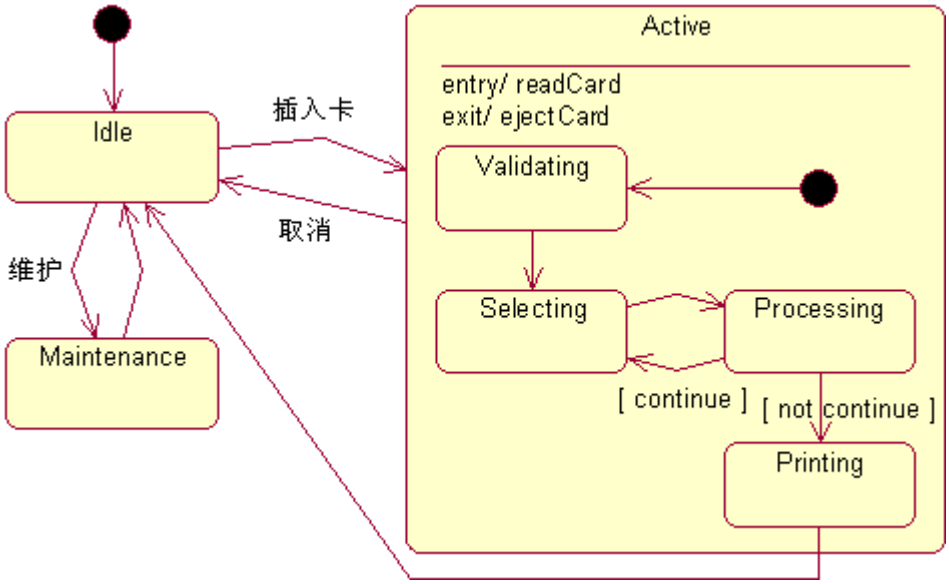
- 有时状态图中存在某些状态，他们一方面可能要执行一系列动作，另一方面则可能要响应一些事件，这时我们可以对这些状态进行分解，得到子状态图，用以描述一个状态的内部状态变化过程，无论对象处于状态图中的哪个状态，外部表现出来的仍是同一个状态。

7）子状态：

是嵌套在另一个状态中的状态，作为组成状态一部分的一个状态。

子状态包括：

- 顺序（互斥）子状态：



在一个封闭的组合状态的语境中给定一组不相交的子状态，对象被称为处在该组合状态中，且一次只能处于这些子状态（或终态）中的一个子状态上，这组子状态称为顺序子状态。

- 顺序子状态将组合状态的状态空间分为不相交的状态。
- 从一个封闭的组合状态外边的一个源状态出发，一个转换可以以组合状态为目标，也可以以一个子状态为目标。如果它的目标为一个组合状态，则这个嵌套状态机一定包括一个初态，以便在进入组合状态后或执行它的进入动作（如有）后，将控制传送给初态。如果它的目标是一个嵌套状态，在执行组合状态的进入动作（如有）和子状态的进入动作后，将控制传送给嵌套状态。
- 一个导致离开组合状态的转换，可能以一个组合状态或一个子状态来作为它的源。在每种情况下，控制都是首先离开嵌套状态（如有则执行它的退出动作），然后离开组合状态（如有则执行它的退出动作）。一个源是组合状态的转换本质上切断（中断）了这个嵌套状态机的活动。
- 一个嵌套的顺序状态机最多有一个初态和一个终态。

■ 并发（正交的）子状态：

对象可同时处于组合状态的不同的子状态中，这组子状态称为并发子状态。

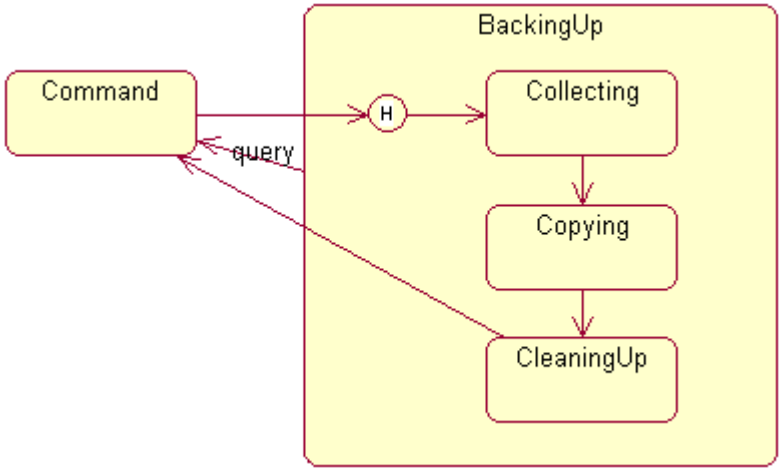
- 两个并发子状态的执行是并发的，最终，每个嵌套状态机都到达它的终态。如果一个并发子状态先于另一个到达它的终态，那么先到的子状态的控制将在它的终态等待。当两个嵌套状态机都到达它们的终态时，来自两个并发子状态的控制就汇合成一个流。

■ 顺序子状态和并发子状态的区别：

- 在同一层次给出两个或更多的顺序子状态，对象将处于这些子状态中的一个子状态或另一个子状态中。而在同一层次给出两个或更多的并发子状态，对象将处于来自每一个并发子状态的一个顺序状态中。
- 嵌套的并发状态机没有初态、终态或历史状态。但是组成一个并发状态的顺序子状态可以具有这些特征。

8）历史状态：

是一种伪状态，说明内部组成状态在退出之后仍然记得它之前的活动子状态。



- 除非特别说明，当一个转换进入一个组合状态时，嵌套状态机的动作就又处于它的初态（当然，除非这个转换的目标直接指向一个子状态）。然而，在许多情况下，对一个对象建模，想要它记住在离开组合状态之前最后活动着的子状态。在 UML 中，对这种惯用法建模的一个比较简单的方法就是使用历史状态。一个历史状态允许一个组合状态包含顺序子状态，以记住来自组合状态的转换之前的最后活动着的子状态。
- 历史状态允许顺序组成状态记住从组成状态发出的转换之前组成状态的最后一个活动子状态，转向历史状态的一个转换使前一个活动子状态再次成为活动的，并执行相应的入口动作和出口动作。
- 如果一个嵌套状态机到达一个终态，则会丢失它储存的历史，就像它未曾第一次进入一样。

9）延迟事件：

指在该状态下暂不处理，但将推迟到对象的另一个状态下排队处理的事件列表。

- 延迟事件是事件的一个列表，这些事件在状态中发生被延迟，直到激活了一个使这些列表事件不被延迟的状态，此时这些列表事件才会发生，并触发转换，就象它们刚刚发生一样。
- 延迟事件的实现需要有一个内部事件队列。如果一个事件发生，并被列为延迟事件，则进入队列。一旦对象进入一个不延迟这些事件的状态，这些事件就会从这个队列中被除掉。

五． 状态空间：

对象的状态空间描述了对象在其生命周期内所可能具有的状态，状态粒度的大小反映了分析者对于问题域的理解和问题域的本质。

六. 转换：

是两个状态间的一种关系，表示对象在第一个状态中执行一定的动作，并在某个待定事件发生而某个特定的条件满足时进入第二个状态，当状态发生这样的转变时，转换被称作激活了。在转换激活之前，称对象处于源状态；激活后，就称对象处于目标状态。

✧ 从状态出发的转换定义了处于此状态的对象对外界发生的事件所做出的反应。

1. 转换的表示：

2. 转换的组成：

1）源状态：

即受转换影响的状态；如果一个对象处于源状态，当该对象接收到转换的触发事件或满足监护条件（如果有）时，就会激活一个离出的转换。

2）事件触发：

是一个事件，源状态中的对象接收到这个事件使转换合法地激活，并使监护条件满足。

3）完成转换：

即没有事件触发的转换，当源状态已经完成它的活动时，它被隐式地触发。

4）自身转换：

自身转换是指源状态和目标状态相同的转换。

- 自身转换先执行该状态的退出动作，接着执行自身转换动作，最后执行该状态的进入动作。
- 一个自身转换会激发状态上的入口动作和出口动作的执行，而内部转换则不会。

5）监护条件：

是一个布尔表达式，当转换因事件触发器的接收而被触发时对这个布尔表达式求值；如果表达式取值为“真”，则激活转换；如果为“假”，则不激活转换。

- 监护条件与改变事件的区别：

监护条件只是在引起转换的触发器事件触发时和事件接收者对事件进行处理时被赋值一次，如果它为“假”，那么转换将不会被激发，条件也不会被再赋值；而变化事件被多次赋值直到条件为真，这时转换也会被激发。

- 监护条件只能在触发事件发生时被赋值一次，如果在转换发生后监护条件由原来的“假”变为“真”，则因为赋值太迟而不能触发转换。
- 通常，监护条件的设置要考虑到各种可能的情况以确保一个触发器事件的发生应该能够引起某些转换，如果有些情况没有考虑到，一个触发器事件没有引起任何转换，那么在状态机视图中要忽略这个事件。

6）动作：

是一个可执行的原子计算，它可以直接的作用于拥有状态机的对象，并间接作用于对该对象是可见的其他对象，它导致状态的变更或者返回一个值。

- 当转换被引起时，它对应的动作被执行。
- 动作是原子性的，一般是一个简短的计算处理过程，通常是一个赋值操作或算术计算。另外还有一些动作，包括给另一个对象发送消息、调用一个操作、设置返回值、创建和销毁对象，没有被定义的控制动作用外部语言来进行详细说明。
- 动作是原子的，这意味着它不能被事件中断，并因此一直运行到完成，而动作序列（活动）则相反，它可以被其他事件中断。
- 按照 UML 中的概念，动作的执行时间非常短，与外界事件所经历的时间相比是可以忽略的，因此，在动作的执行过程中不能再插入其他事件。然而，实际上任何动作的执行都要耗费一定时间，新到来的时间必须被安置在一个队列中。

7）目标状态：

在转换完成后活动的状态。

§4.3 状态图建模

一. 建立状态模型的步骤：

建立状态模型首先要识别状态空间，定义各个可能的状态；然后定义状态间转移的充分条件。状态间转移的充分条件可以表现为

多种多样，如：特定事件的发生、某个特定判定条件的满足、特定时间段的流逝等。

1. 识别对象状态空间

分析一个对象的状态时，首先一个问题就是该对象有哪些状态是问题域所关心的？当对象处于这些状态中时会有哪些动作？这就是对象的状态空间识别问题。

- ✧ 从广义上来说，对象的状态有很多（由对象的数据属性确定），有时甚至是无穷无尽的，但其中只有很少一部分为问题域所关注。
- ✧ 不仅如此，有些状态虽然是不同的状态（从数据属性取值角度），但从问题域的角度他们却是相同的，即问题域并不关心这些状态在某些属性取值方面的不同，这实质上是状态粒度问题，状态粒度越大，状态数目就会越少；反之状态数目则越多。状态粒度通常由问题域确定，而不由分析员随意指定。
- ✧ 状态刻画了对象存在的状况，因而状态变化反映了对象的活动情况，如：创建、运行、撤销等。可见对象的状态变化过程实质上反映了对象生命周期内的演化过程。

对象状态空间识别的步骤：

1）识别对象在问题域中的生命周期；

- ■ 直线式的：

  直线式的生命周期通常具有一定的时间顺序特性，即对象进入初始状态后，经过一段时间会过渡到后续状态，如此直至生命结束。

- ■ 循环式的：

  循环式的生命周期通常并不具备时间顺序特性，在一定条件下，对象会返回到以及经过的生存状态。

对象的生命周期描述只是相对于某个特定的问题域，不具有广泛的意义，同样的对象在不同问题域中的生命周期不尽相同。

2）确定对象生命周期阶段划分策略；

不管对象的生命周期是直线式的，还是循环式的，都具有一定的时间特性，因而都可以划分阶段，在某个阶段中则具有较强的时间特性。

3）重新按阶段描述对象生命周期，得到候选状态；

4）识别对象在每个候选状态下的动作，并对状态空间进行调整；

5）分析每个状态的确定因素（对象的数据属性）；

6）检查对象状态的确定性和状态间的互斥性。

最后还应对得到的状态进行检查，一般我们认为可以从对象的状态确定性和状态间的互斥性两个方面进行检查。所谓对象状态的确定性是指每个状态都可由对象某些数据属性的组合来惟一确定，在一般应用中，对象的不同状态间要求必须是互斥的，即任何两个状态之间不存在一个“中间状态”，使得该“中间状态”同时可以归结到这两个状态。从本质上来说，就任何两个状态所对应的对象数据属性都应该是不同的。

2. 识别状态转移

识别状态转移的步骤：

1）建立最简单的状态转移

在建立最简单的状态转移时可以考虑两种策略：

- 一是按照生命周期历程在所有的状态节点中找出一条“生命周期”路径，这时可能有些状态节点不在该路径上；
- 另一种是依次考虑每一个状态节点，对象由该状态可以之间变化到哪些状态，并在该状态节点和那些可以直接由该状态变化到达的状态节点间一一建立状态连接。

2）建立了最简单的状态转移后，通过为每一个状态转移识别激活的充分条件，得到较为详尽的状态图。

3）最后，还要检查整个状态图，并分析问题域中所有可能与该对象相关的事件，检查是否所有事件都已出现在状态图中，如果有些事件没有出现则要分析该事件是否不需要对象响应，否则就应该分析应该由哪些状态转移来响应这些事件。

二. 状态建模策略：

状态机对于单个对象的整个生命周期的行为建模。当对对象的生命周期建模时，主要描述以下 3 种事物：对象能响应的事件、对这些事件的响应、以及过去对当前行为的影响。对对象的生命周期建模，还包括决定该对象有意义地响应事件的顺序，从对象的创建时开始，一直到它被撤销。

1. 对对象的生命周期建模，要遵循如下的策略：

1）如果语境是一个类或一个用况，则收集相邻的类，包括这个类的所有父类和通过依赖或联到达的所有类。这些邻居是动作的候选目标或在监护条件中包含的候选项。

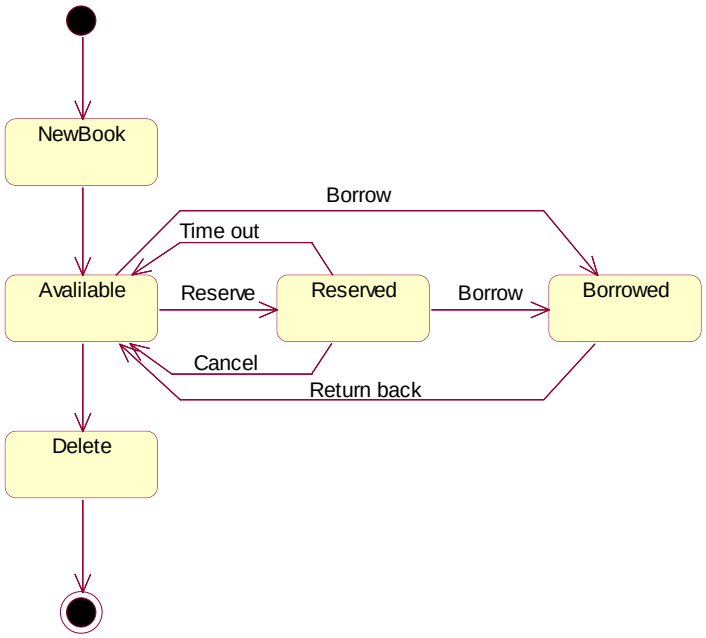
2) 如果语境是整个系统，则将你的注意力集中到这个系统的一个行为上。理论上，系统中每个对象都可以是系统的生命周期的模型中的一个参加者，而且除了最微小的系统，建立一个完整的模型将是非常棘手的。

2. 一个结构良好的状态机，应满足以下的要求：
   1) 是简单的，因而不包含任何多余的状态或转换。
   2) 具有清晰的语境，可以访问所有对闭合对象可见的对象（仅当执行由状态机描述的行为是必须的时候才使用这些相邻近对象）。
   3) 是有效的，因而应该根据执行动作的需要，取时间和资源的最优平衡来完成它的行为。
   4) 是可理解的，因而应该使用来自系统的词汇中的词汇，来命名它的状态和转换。
   5) 不要太深层地嵌套（一层或两层地前台子状态能够解决大多数复杂行为）。
   6) 像使用主动类那样借阅地使用并发子状态常常是更好的选择。
3. 当在 UML 中绘制状态机时，要遵循如下的策略：
   1) 避免交叉的转换。
   2) 只在为使图形可理解所必须的地方扩充组合状态。

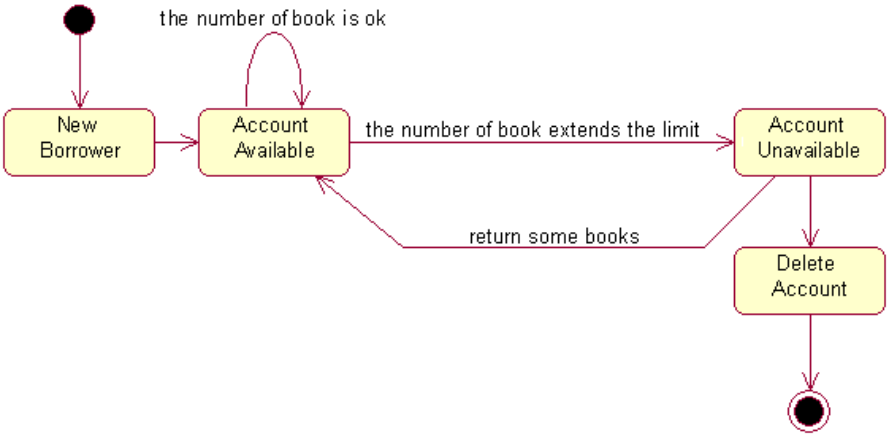
§4.4 案例分析：《图书馆管理系统》

在图书馆管理系统中，有明确状态转换的类包括：书籍和借阅者的账户（相当于包含特定个人信息的电子借阅证）。可以在系统中为这两类事物建立状态图。

一． 书的状态图：



二． 借阅者账户的状态图：



§4.5 补充案例：《电梯系统》

一． 系统描述：

电梯初始停在一层等待乘客的服务请求，一旦某个楼层有电梯服务请求，电梯就赶往该楼层。乘客进入电梯后需要指明（选择）

目的地楼层，然后电梯将把乘客送往目标楼层。只要有电梯服务请求，电梯就将不停地上上下下搭载、运送乘客。否则电梯将空闲在某一楼层，如果在一个限定的空闲时间段内仍没有服务请求，那么电梯将自动行驶到一层停靠并等待乘客的服务请求。

每个楼层都装有两个按钮，一个为向上按钮，另一个为向下按钮（一层只有向上按钮，顶层则只有向下按钮）。一旦乘客按下其中的某个按钮，就意味着发出一个服务请求，电梯将按照一定的策略来响应该请求。电梯内部控制台上有几个重要按钮，其中两个是控制电梯门开与关的按钮，要求电梯在行驶过程中必须保持电梯门关闭。当电梯停靠到某层时，将自动开启门，如果在一段时间以内乘客没有按下其中的开或关电梯门按钮，电梯门将自动关闭。如果乘客按下开按钮，电梯门将保持开启，电梯也将停泊在某楼层而不能运行，直至电梯门关闭。另外的一批按钮是楼层号按钮，用来选定乘客的目的地楼层。

电梯在向上运行过程中可以响应到达某个楼层或接纳上行乘客（即按下向上的服务请求按钮）而停靠，但是不响应下行的服务请求（即按下向下的服务请求按钮），直至电梯改变了运行方向。同样，当电梯在向下运行时，可以响应到达某个楼层和接纳下行乘客而停靠，但是不响应上行服务请求直至电梯改变了运行方向。

当电梯在某层停靠时，该层的所有服务请求按钮灯都将被灭掉，同时电梯内的对应该层的按钮灯也是如此。

二. 状态建模：

由上面的问题域描述，我们需要分析电梯对象、楼层服务请求按钮对象、楼号按钮对象、电梯门控制按钮对象的状态图。

1. 电梯对象的状态模型：

1）以电梯的运行状况为策略对电梯对象的生命周期进行阶段划分：

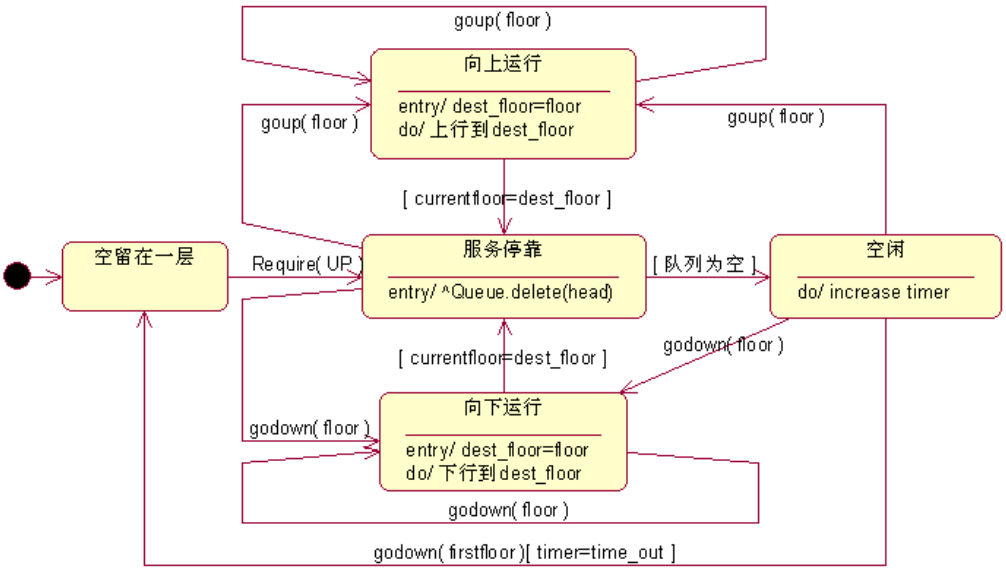
- 电梯停留在一层；
- 电梯向上运行满足乘客服务；
- 电梯停靠到某一楼层让乘客上下电梯；
- 电梯向下运行满足乘客服务；
- 电梯空闲在某一楼层。

2）由上面问题域的描述可知，电梯在运行过程中可能会收到多个服务请求，电梯必须能够保存这些请求（否则这些请求事件就丢失了）。从服务请求的“先来先服务”特点可知，应该有一个对了

来维护（接收、存储和转发）事件。同时电梯服务具有捎带特性（如电梯在上行去往 20 层途中，10 层有上行乘客，则电梯在 10 层停靠并继续上行），所有电梯的调度算法就体现在如何把某个事件（服务请求事件和目的地事件）放置到队列的正确位置中，关于事件队列有以下要求：

- 当前电梯的运行时作为队列头事件的响应；
- 直到电梯完成对当前事件的响应才从队列头删除该事件；
- 如果电梯处于向上运行状态，队列将把所有电梯本次运行所能捎带的中间楼层相关事件（搭载请求事件和到达事件），都按照楼层号的由小到大顺序排列在当前电梯运行的最高目的地楼层相关事件前面；
- 如果电梯不能捎带服务当前队列接收到的事件，那么将把该事件放置到队列尾；
- 如果电梯处于下行状态，队列将把所有电梯的本次运行能捎带服务的中间楼层相关事件，按照楼层由大到小顺序插入到当前电梯运行的最低的目的地楼层相关事件前面；
- 队列随时可以把队列头事件发送给电梯对象，但不删除队列头事件。

3）电梯的状态图如下所示：



2. 服务停靠状态的子状态模型：

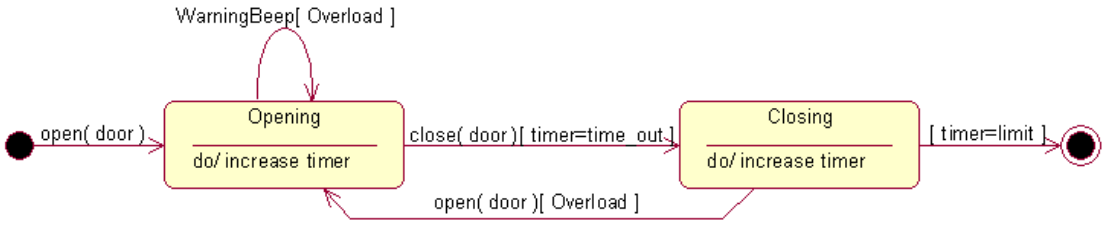
通过上述分析，我们发现状态图不能“记忆”事件（这是因为有穷状态机 FSM 本身的原因所造成的，对于 FSM 如果要记住一个东西，就必须增加一个状态），因此必须借助于队列和相应的调度算法，否则状态图本身将无法描述电梯运行的复杂逻辑。同时，也要

注意状态图只表示对象如何对事件作出响应并进行状态转移，它并不能表示对象需要记忆的事件和状态（如：当电梯处于服务停靠状态时，电梯本身并不知道它的前一个状态是什么）。此时如果问题域要求对象要记忆某些事件和状态，就要考虑引入一下辅助手段（如队列等）。

服务停靠是一个复杂的状态，我们要分析它的子状态图。当电梯一旦停靠在某一楼层，首先电梯门将自行打开，在一个限定的时间内如果没有来自电梯门按钮的事件，电梯门则自行关闭。如果乘客按下关电梯门按钮，则电梯门立即关闭；而如果按下了开按钮，此时如若电梯门是开的，则没有响应，如果电梯门是关闭的，并且电梯还没有运行，则电梯门将打开。在电梯门关闭后，如果有服务请求的话，电梯将持续一段较短的事件后开始运行。另外，电梯一旦停靠某个楼层，就立即发送相应的熄灭按钮灯事件给该楼层对应的楼层按钮，这是一个 entry 事件。

另外还要提出一点，电梯有可能会出现一个重要异常事件，即超载。这个事件通常出现在服务停靠状态，如果不能校车这个异常，则电梯将保持停止状态。一旦有超载事件，传感器将不停地向电梯发送该事件，使得电梯只能停靠在某个楼层，同时电梯将发出嘟嘟的警告声。

服务停靠状态的子状态图如下所示：



对于楼层停靠按钮（召唤按钮）、目的地楼号按钮和其他的按钮对象而言，可以仿照上面的做法建立它们的状态图，并注意此时这些按钮事件并不直接发给电梯，而是发给队列，然后才由队列经过调度算法有选择地把事件发送给电梯。

§ 4.6 小结

第五章　活动图

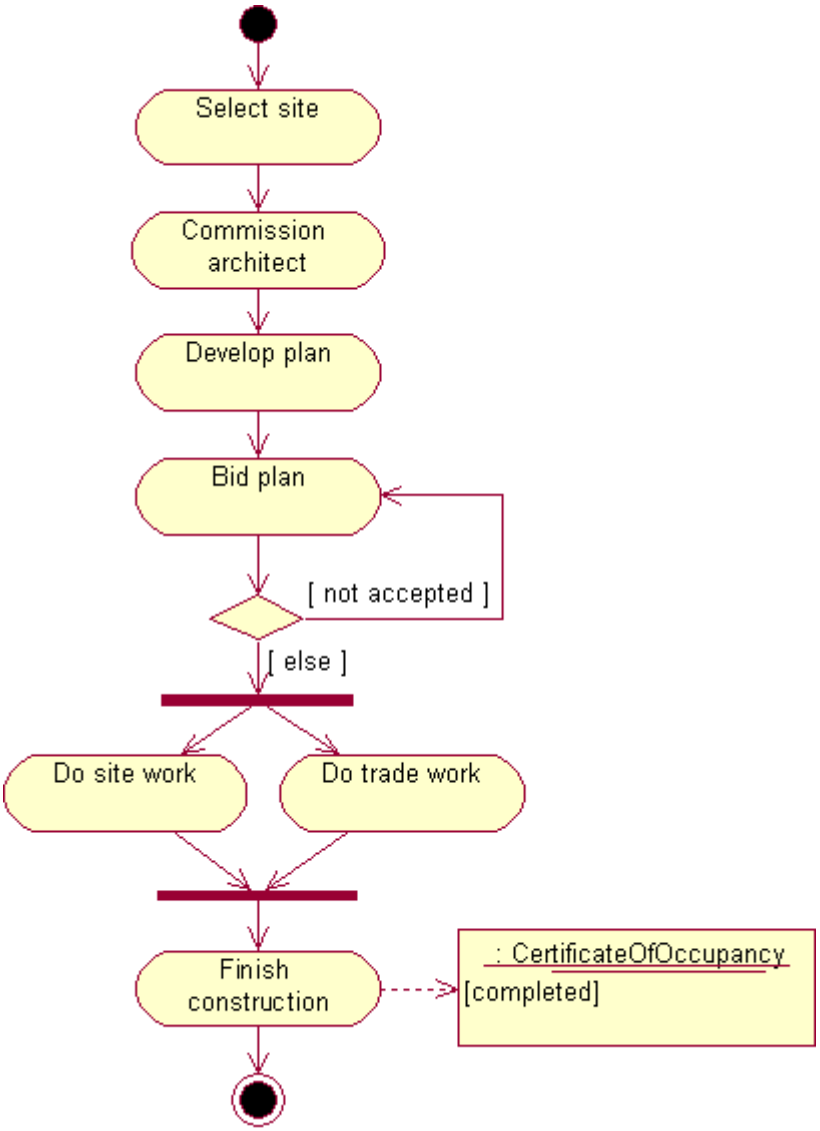
§5.1 基本概念

一. 活动图：

是 UML 用于对系统的动态行为建模的另一种常用工具，它描述活动的顺序，展现从一个活动到另一个活动的控制流。

- ✧ 活动是某件事情正在进行的状态，既可以是现实生活中正在进行的某一项工作，也可以是软件系统某个类对象的一个操作。
- ✧ 活动图本质上是一种流程图。

1. 活动图的组成：



- 动作状态
- 活动状态
- 动作流
- 分支与合并
- 分叉与汇合
- 泳道
- 对象流

2. 活动图与流程图的区别：

活动图描述系统使用的活动、判定点和分支，看起来和流程图没什么两样，并且传统的流程图所能表示的内容，大多数情况下也可以使用活动图表示，但是两者是有区别的：

- 流程图着重描述处理过程，它的主要控制结构是顺序、分支和循环，各个处理过程之间有严格的顺序和时间关系；而活动图描述的是对象活动的顺序关系所遵循的规则，它着重表现的是系统的行为，而非系统的处理过程。
- 活动图能够表示并发活动的情形，而流程图不能。
- 活动图是面向对象的，而流程图是面向过程的。

二. 动作状态：

是指执行原子的、不可中断的动作，并在动作完成后通过完成转换转向另一个状态。

1. 动作状态的表示：

在 UML 中动作状态使用平滑的圆角矩形表示。

2. 动作状态的特点：

1) 动作状态是原子的，它是构造活动图的最小单位，已经无法分解为更小的部分。
2) 动作状态是不可中断的，它一旦开始运行就不能中断，一直运行到结束。
3) 动作状态是瞬时的行为，它所占用的处理时间极短，有时甚至可以忽略。
4) 动作状态可以有入转换，入转换既可以是动作流，也可以是对象流。动作状态至少有一条出转换，这条转换以内部动作的完成为起点，与外部事件无关。
5) 动作状态和状态图中的状态不同，它不能有入口动作和出口动

作，更不能有内部转移。

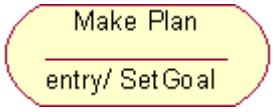
6）在一张活动图中，动作状态允许多处出现。

三． 活动状态：

活动状态用于表达状态机中的非原子的运行。

1. 活动状态的表示：

与动作状态的表示相同。



2. 活动状态的特点：

1）活动状态可以分解成其他子活动或动作状态，由于它是一组不可中断的动作或操作的组合，所以可以被中断。

2）活动状态的内部活动可以用另一个活动图来表示。

3）和动作状态不同，活动状态可以有入口动作和出口动作，也可以有内部转移。

4）动作状态是活动状态的一个特例，如果某个活动状态只包括一个动作，那么它就是一个动作状态。

四． 动作流：

动作状态之间的转换流称为动作流。

1. 动作流的表示：

与状态图中的转换相同，用带箭头的直线表示，箭头的方向指向转入的方向。

2. 动作流的特点：

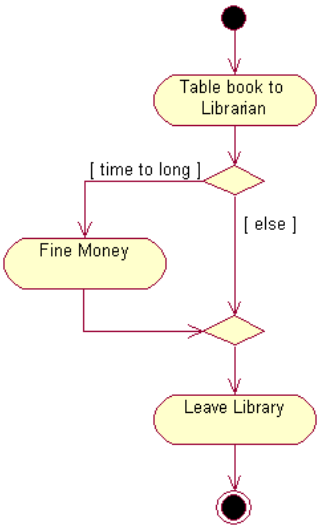
与状态图不同，活动图的转换一般都不需要特定事件的触发。一个动作状态执行完成本状态需要完成的动作后会自发转换到另一个状态。

五． 分支与合并：

分支用于表示对象类所具有的条件行为。一个无条件的动作流可以在一个动作状态的动作完成后自动触发动作状态的转换以激发下一个动作状态，而有条件的动作流则需要根据条件，即一个布尔表达式的真假来判定动作的流向。条件行为用分支和合并表达。

1. 分支与合并的表示：

用空心小菱形表示。



2. 分支与合并的特点：

1）分支包括一个入转换和两个带条件的出转换，出转换的条件应当是互斥的，这样可以保证只有一条出转换能够被触发。

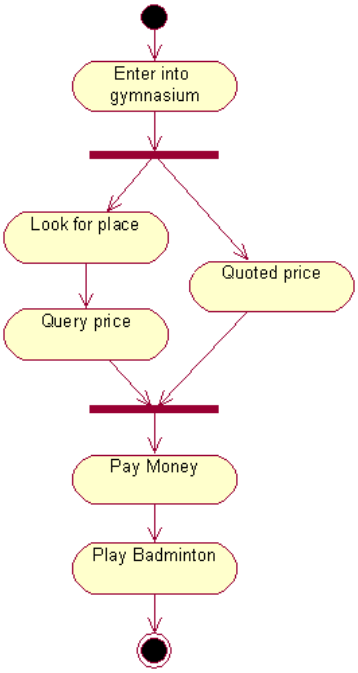
2）合并包括两个带条件的入转换和一个出转换，合并表示从对应的分支开始的条件行为的结束。

六. 分叉与汇合：

对象在运行时可能会存在两个或者多个并发运行的控制流，为了对并发的控制流建模，在 UML 中引入了分叉与汇合的概念。分叉用于将动作流分为两个或者多个并发运行的分支，而汇合则用于同步这些并发分支，以达到共同完成一项事务的目的。

1. 分叉与汇合的表示：

用加粗的水平（或垂直）线段表示。



2. 分叉与汇合的特点：

1）分叉可以用来描述并发线程，每个分叉可以有一个入转换和两个或多个出转换，每个转换都可以是独立的控制流。

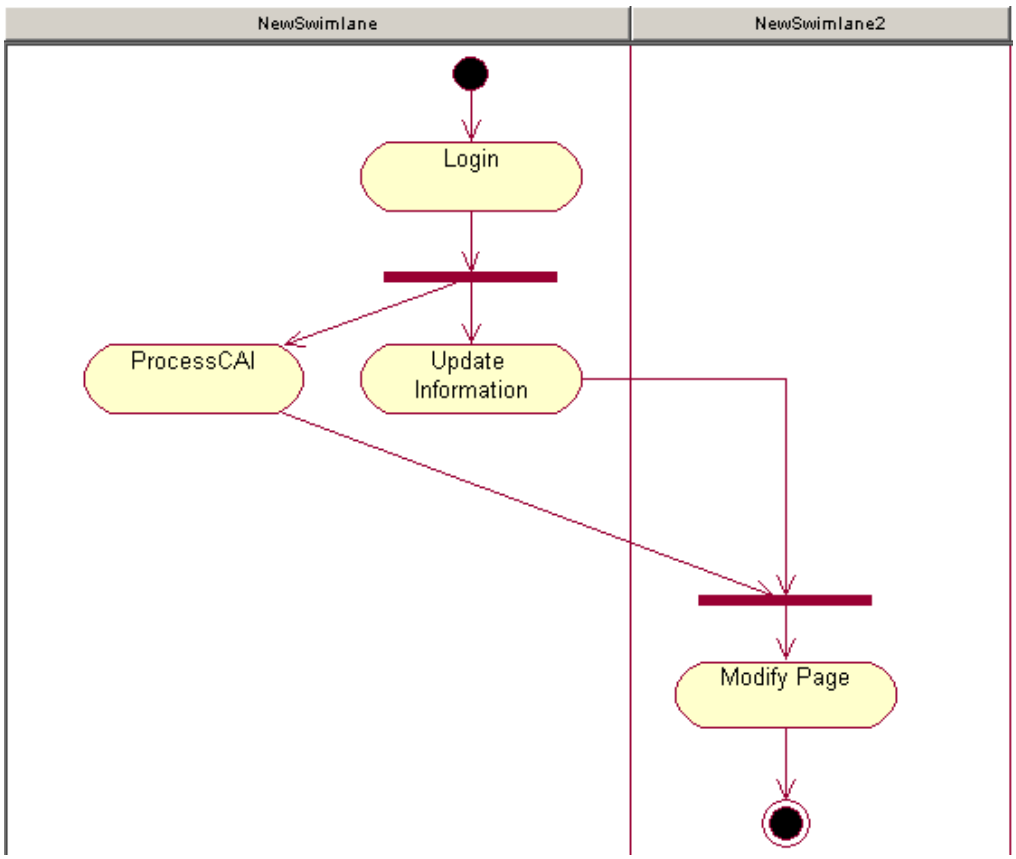
2）汇合代表两个或多个并发控制流同步发生，当所有的控制流都达到汇合点后，控制才能继续往下进行。每个汇合可以有两个或多个入转换和一个出转换。

七. 泳道：

泳道将活动图中的活动划分为若干组，并把每一组指定给负责这组活动的业务组织，即对象。

1. 泳道的表示：

在活动图中，泳道用垂直实线绘出，垂直实线分隔的区域就是泳道，在泳道上方可以给出泳道的名字或对象（对象类）的名字，该对象（对象类）负责泳道内的全部活动。



2. 泳道的特点：

泳道没有顺序，不同泳道中的活动既可以顺序进行也可以并发进行，动作流和对象流允许穿越分隔线。

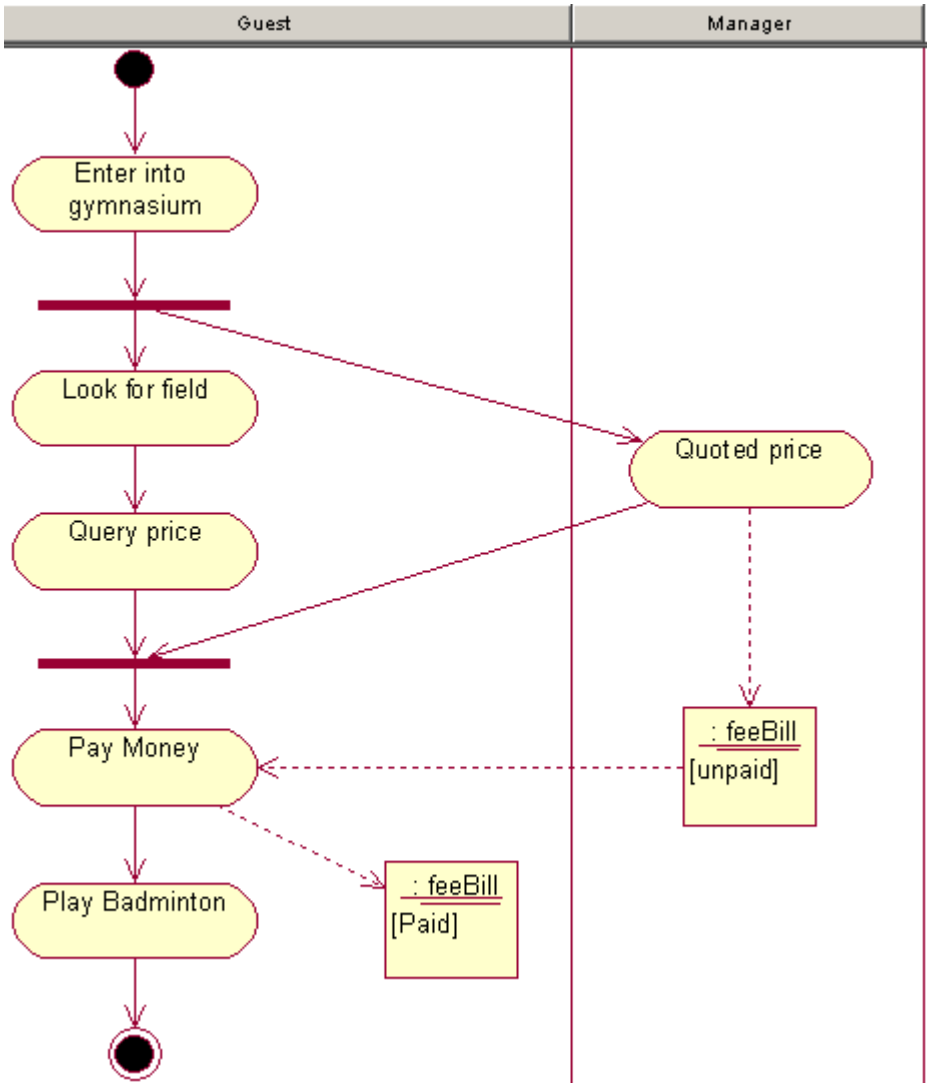
八. 对象流：

是动作状态或者活动状态与对象之间的依赖关系，表示动作使

用对象或者动作对对象的影响。

1. 对象流的表示：

在活动图中，对象流用带箭头的虚线表示。如果箭头从动作状态出发指向对象，则表示动作对对象施加了一定的影响，施加的影响包括创建、修改和撤销等；如果箭头从对象指向动作状态，则表示该动作使用对象流所指向的对象。



2. 对象流中的对象的特点：

1）一个对象可以由多个动作操纵。

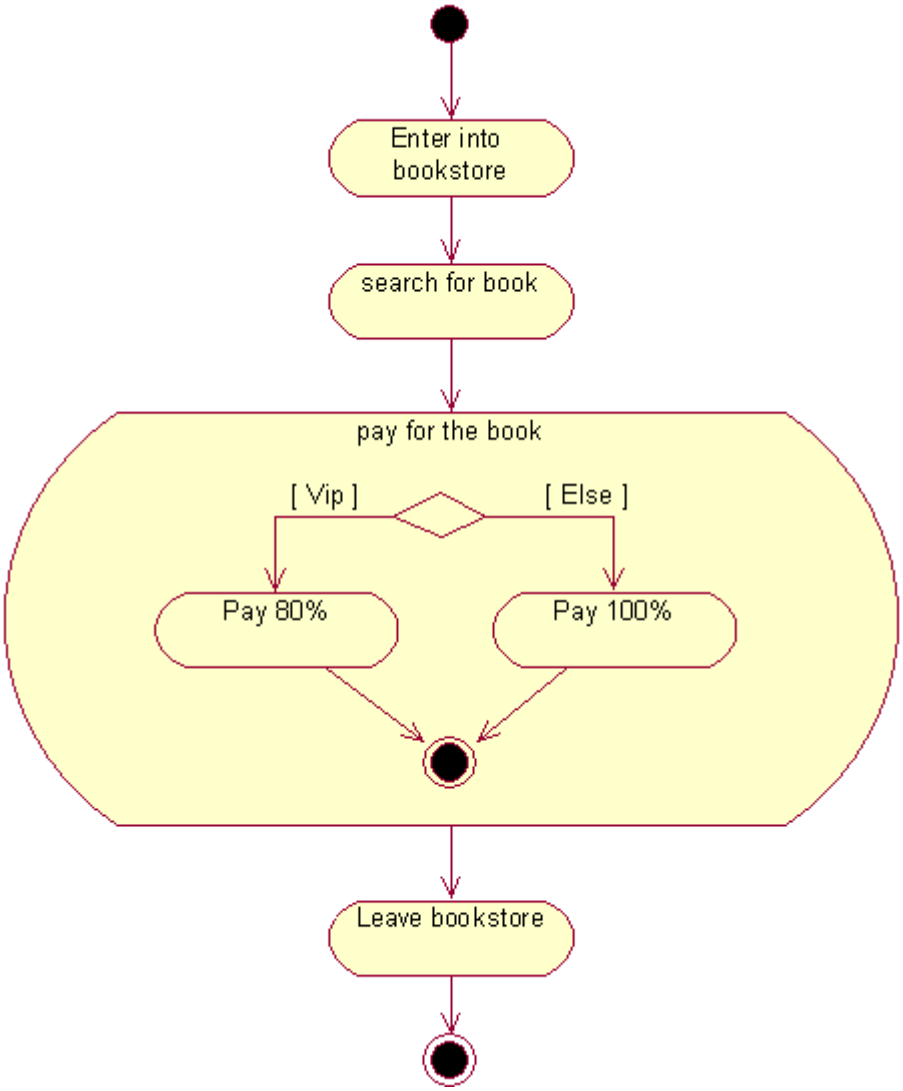
2）一个动作输出的对象可以作为另一个动作输入的对象。

3）在活动图中，同一个对象可以多次出现，它的每一次出现表明该对象正处于对象生存期的不同时间点。

九. 分解：

一个活动可以分为若干个动作或子活动，这些动作和子活动本身又可以组成一个活动图。不含内嵌活动或动作的活动称之为简单

活动；嵌套了若干活动或动作的活动称之为组合活动，组合活动有自己的名字和相应的子活动图。



§ 5.2 活动图建模

一. 对工作流建模：

关注于与系统进行协作的参与者所观察到的活动，工作流常常位于软件系统的边缘，用于可视化、详述、构造和文档化开发系统所涉及的业务过程。在活动图的这种用法中，对对象流的建模是特别重要的。

对工作流建模，要遵循如下的策略：

✧ 为工作流建立一个焦点。除非很小的系统，否则不可能在一张图上显示所以感兴趣的工作流。

- ✧ 选择为全部工作流中的一部分有高层职责的业务对象。这些业务对象可以是系统的词汇中的真实事物，也可能较为抽象。无论哪种情况，为每个重要的业务对象建立一个泳道。
- ✧ 识别该工作流初始状态的前置条件和该工作流停止状态的后置条件，这对工作流的边界建模是重要的。
- ✧ 从该工作流的初始状态开始，说明随着时间发生的动作和活动，并在活动图中把它们表示成活动状态或动作状态。
- ✧ 将复杂的动作或多次出现的动作集合归并到一个活动状态，并对每个这样的活动状态提供一个可将它展开的单独的活动图。
- ✧ 找出连接这些活动和动作状态的转换。首先从工作流的顺序流开始，然后考虑分支，接着再考虑分叉和汇合。
- ✧ 如果工作流中涉及重要的对象，则也把它们加入到活动图中。如果对表达对象流的意图是必要的，则显示其变化的值和状态。

二. 对操作建模：

把活动图作为流程图使用，对一个计算的细节部分建模。在活动图的这种用法中，对分支、分叉和汇合状态的建模是特别重要的。采用这种方式，活动图只是一个操作的动作的流程图。

对操作建模，要遵循如下的策略：

- ✧ 收集这个操作所涉及的抽象。包括操作的参数（含它的返回类型，如果有）、所属类的属性以及某些临近的类。
- ✧ 识别该操作的初始状态的前置条件和停止状态的后置条件。也要识别在操作执行过程中必须保持的所属类的不变式。
- ✧ 从该操作的初始状态开始，说明随着时间发生的活动和动作，并在活动图中将它们表示为活动状态和动作状态。
- ✧ 如果需要，使用分支来说明条件路径和迭代。
- ✧ 仅当这个操作属于一个主动类时，才在必要时用分叉和汇合来说明并行的控制流。

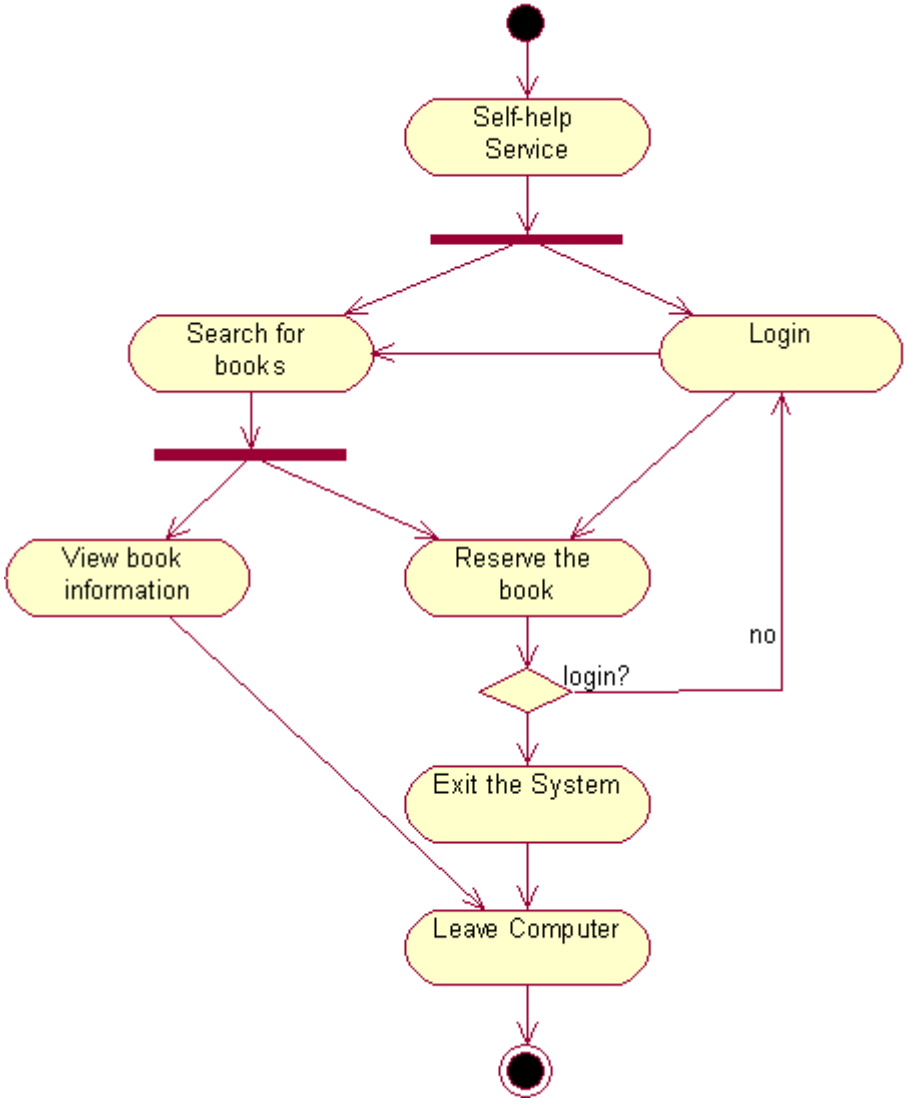
三. 活动图建模要求：

1. 一个结构良好的活动图，应满足如下的要求：
   1) 关注于与系统动态特性的一个方面的交流。
   2) 只包含那些对于理解这个方面必不可少的元素。
   3) 提供与它的抽象层次相一致的细节；只加入那些对于理解问题必不可少的修饰。
   4) 不应该过分简化和抽象信息，以致使读者误解重要的语义。

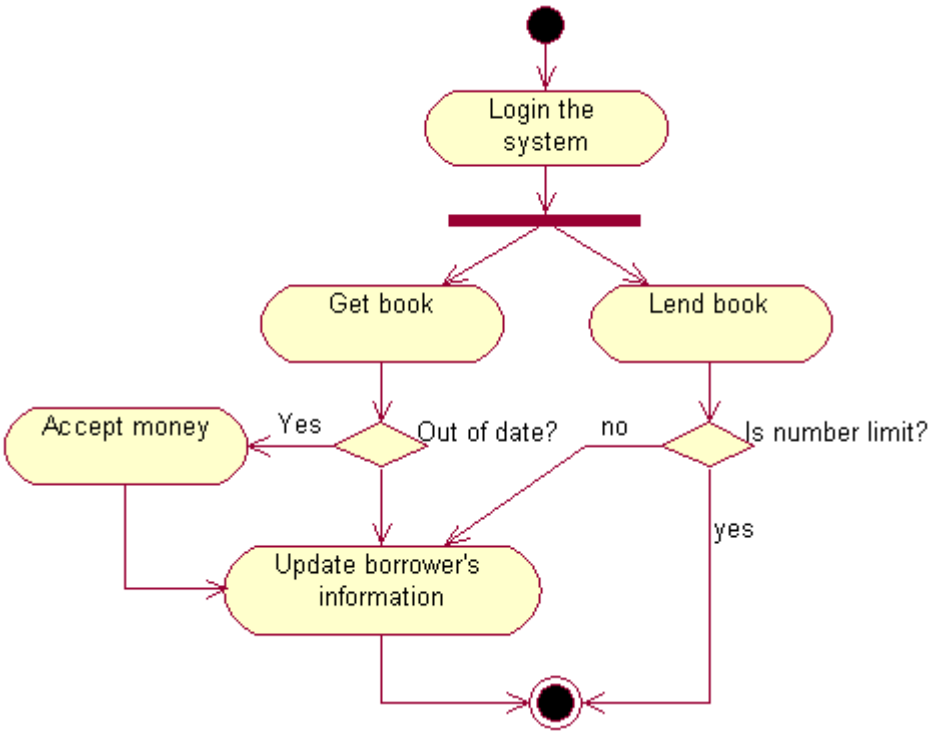
2. 当绘制活动图时，要遵循如下的策略：
   1）给出一个能反映其目的的名称。
   2）首先对主要的流建模，其次标出分支、并发和对象流，它们也可能被放在几张分开的图中。
   3）摆放它的元素以尽量减少线的交叉。
   4）使用注解和颜色作为可视化提示，以突出图形中重要的特征。

§5.3 案例分析：《图书馆管理系统》

一. 借阅者的活动图：

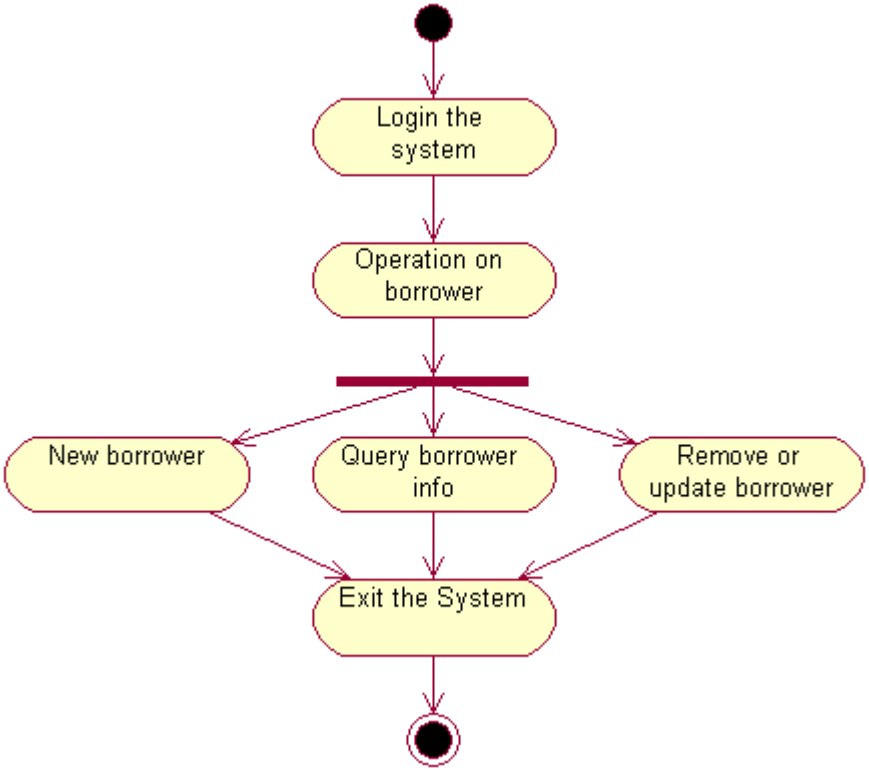


二. 图书管理员的活动图：

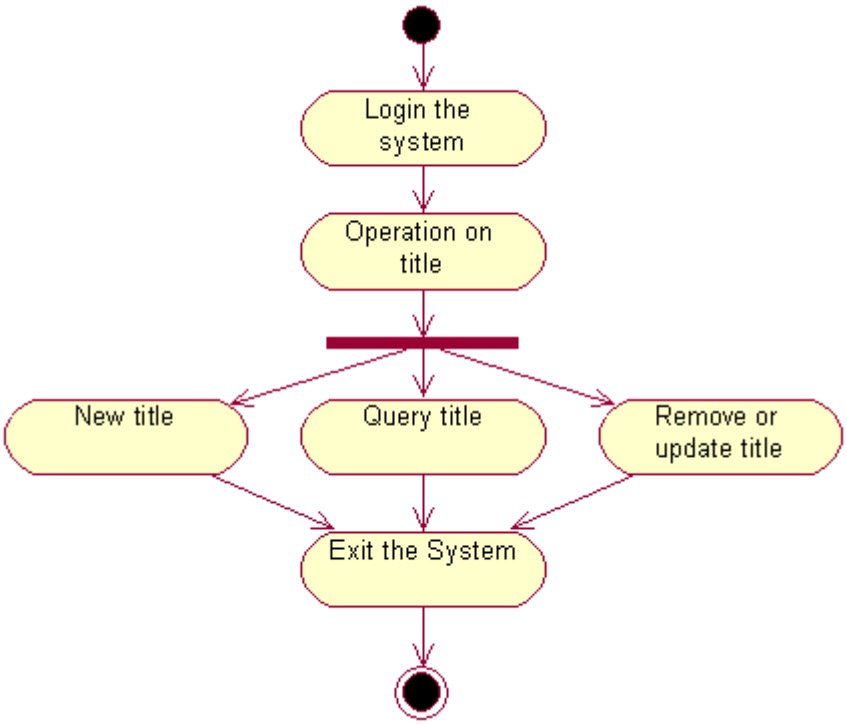


三. 系统管理员的活动图：

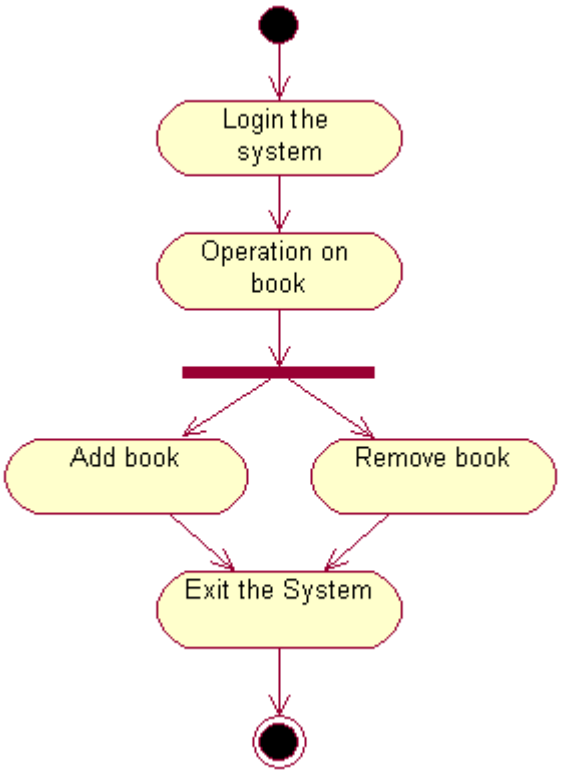
1. 系统管理员维护借阅者帐户的活动图：



2. 系统管理员维护书目信息的活动图：



3. 系统管理员维护书籍信息的活动图：



§5.4 小结

第六章　交互图

§6.1 基本概念

一. 交互模型

1. 交互：

   是一组对象之间为完成某一任务（如：实现一个操作）而进行的一系列消息交换的行为说明。

2. 交互图（interaction diagram）：

   显示一个交互，由一组对象和它们之间的关系构成，其中包括在对象间传递的消息。

   交互图中表示对象之间交互的模式，交互图在同一个信息基础上发展为不同的形式，各自有不同的侧重点。它们是：顺序图、协作图。

   - ✧ 顺序图表示按时间排序的交互，着重表现参与交互对象的生命线和它们交换的信息，顺序图不表示对象之间的链。
   - ✧ 协作图表示执行操作的对象间的交互，它类似于对象图，表示了实现高层操作所需的对象和它们之间的链。
   - ✧ 顺序图和协作图用不同的方式表示了类似的信息，顺序图表示消息的确切次序，更适合于实时说明和复杂的情形；协作图表示对象之间的关系，更适合于理解对象的全部作用和过程设计。

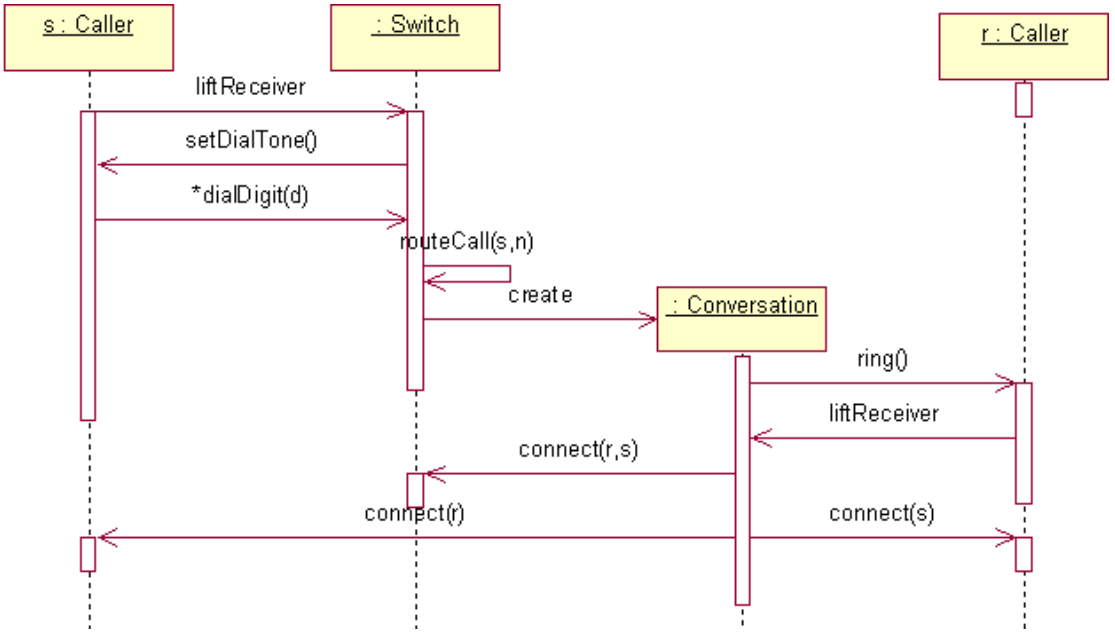
3. 交互模型：

   对象在其生命周期内要不断地与其他对象交互，交互模型用来描述对象间的动态交互行为。

   UML 使用两个模型来表示交互模型：顺序模型和协作模型，顺序模型用顺序图(Sequence diagram)表示，协作模型用协作图(Collaboration diagram)表示。

二. 顺序图

以时间顺序显示对象交互的图，它显示了参与交互的对象和所交换消息的顺序。



- ✧ 由于对象生存期的引入，顺序图具备了时间顺序的概念，从而可以清晰地表示对象在其生存期的某一时刻的动态行为。
- ✧ 顺序图描述了对象之间传送消息的时间顺序，它用来表示用例中的行为顺序。当执行一个用例行为时，顺序图中的每条消息对应了一个类操作或状态机中引起转换的触发事件。
- ✧ 对于简单的对象交互情况，顺序图表现得非常优秀。一旦交互的对象数目增加，交互情况变得复杂时，顺序图将显得非常庞大和杂乱，很多消息不得不穿越多个对象得生命线才能到达目的对象。这使得顺序图得清晰性和可理解性急剧下降。
- ✧ 在 UML 中，顺序图将交互关系表示为二维图，即顺序图具有两个方向：垂直方向代表时间，水平方向代表参与交互的对象，通常，时间向下延伸。通常只有消息的顺序是重要的，但是在实际应用中，时间轴可以是一个实际的测量尺度，对象的水平次序没有重要意义。

1. 顺序图的组成：

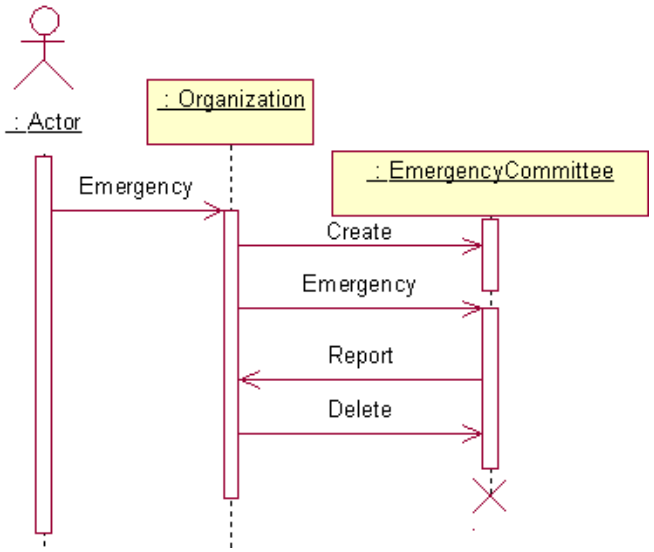
1）对象：

- ✧ 顺序图中对象的符号和对象图中对象所用的符号一样，都是使用矩形将对象名称包含起来，并且对象名称下有下划线。
- ✧ 每个对象显示在单独的列中，对象符号放置在代表生成这个对象的消息的箭头的端点，其垂直位置表示这个对象第一次创建的时间。将对象置于顺序图的顶部意味着在交互开始的

时候对象就已经存在了，如果对象的位置不在顶部，那么表示对象是在交互的过程中被创建的。

- ✧ 在交互图中出现的大多数对象存在于整个交互过程中，所以，这些对象全都排列在图的顶部，其生命线从图的顶部画到图的底部，但对象也可以在交互过程中创建，它们的生命线从接收到构造型为 create 的消息时开始；对象也可以在交互过程中撤销，它们的生命线在接收到构造型为 destroy 的消息时结束（并且给出一个大 X 的标记表明生命的结束）。

■ 对象的创建与撤销：



2) 生命线：

是一条垂直的虚线，表示顺序图中的对象在一段时间内的存在。生命线是一个时间线，从顺序图的顶部一直延伸到底部，所用的时间取决于交互持续的时间。

- ✧ 当对象存在时，生命线用一条虚线表示，当对象的过程处于激活状态时，生命线是一个双道线。

3) 消息：

定义的是对象之间某种形式的通信，它可以激发某个操作、唤起信号或导致目标对象的创建或撤销。

- ✧ 消息实质上是一种便于对象进行交互的协议，消息具有一定语法结构，当然更重要的是必须包含语义信息。
- ✧ 消息通常可以表现为一个对象对另一个对象的操作的调用，控制从被调用的操作返回到调用者（对象）。
- ✧ 在网络上通过一定通讯机制传送的实际消息（如询问等）。
- ✧ UML 的所有动态模型都采用消息机制来进行对象的交互。

- ✧ 消息用从一个对象的生命线到另一个对象生命线的箭头表示。箭头以时间顺序在图中从上到下排列。
- ✧ 消息在生命线上所处的位置并不是消息发生的准确时间，只是一个相对的位置。如果一个消息位于另一消息的上方，只说明它先于另一个消息被发送。

消息可分为四种类型：同步消息、异步消息、简单消息和同步返回消息。

- ■ 简单消息：

  仅仅指明从一个对象到另一个对象有一条消息发送，具体该消息的细节情况（如消息参数等）不得而知，或者在模型中属于无关信息。通常可以使用简单消息来表示操作调用的返回。

- ■ 同步消息：

  表明一个对象一旦发出一条同步消息出去后，必须等待该消息接受者对此同步消息的回应，然后才能继续往下执行相应的动作（这就是同步的概念）。同步消息通常表现为一个对象对另一个对象的操作的调用。

- ■ 异步消息：

  与同步消息相反。消息发送者在发送消息之后并不需要等待消息接受者对消息的回应而继续执行其相应的动作，这种情况在并发执行环境中较为常见。

- ■ 同步立即返回消息：

  本质上就是一条同步消息，只不过强调消息返回的立即性，所以用同步消息与简单消息的叠加来表示。

4) 消息条件：

有时对象在发送消息前需验证某个条件，只有条件为真才发送消息，否则就不发送。这个条件可以称为消息条件，可直接标在消息上面。

5) 标号：

在 UML 中，在对象生命线旁边标注时间的符号、数字等统称为标号。标号一般可分为两种，一种为时间约束，另一种定义消息发送的重复操作。

重复标号一般使用在一下两种情况：

- ■ 对象需要向多个同类型对象发送相同的消息；

■ 对象需要向某个对象重复发送相同的消息。

6）激活（控制期）：

是操作的执行，表示一个对象直接地或通过从属操作完成操作的过程，它对执行的持续时间和执行与其调用者之间的控制关系进行建模。

✧ 激活是执行某个操作的实例，它包括这个操作调用其他从属操作的过程。

✧ 激活用一个细长的矩形框表示，表示一个对象执行一个动作所经历的时间段，既可以是直接执行，也可以是通过下级过程执行。矩形的顶部表示动作的开始，底部表示动作的结束（可以由一个返回消息来标记）。

✧ 在过程代码中，激活表示一段持续时间，在这段持续时间中过程或者被初始过程调用的从属过程是活动的，换而言之，所以活动的嵌套过程激活被同时表示，对于具有现存激活的对象的第二次调用而言，第二次激活的符号画在第一次激活符号的右边，这样它们好像叠加了起来，被叠加的调用可以嵌套于任意深度，调用可以针对同一个操作（即递归调用），也可以针对同一个对象的不同操作。

✧ 激活表示该对象被占用以完成某个任务；去激活指的是对象处于空闲状态，在等待消息。

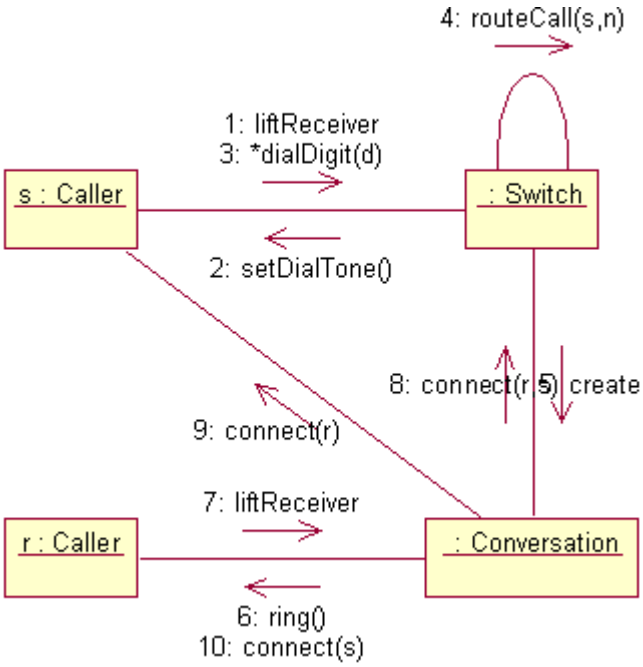
2. 顺序图的生成：

形成顺序图时，首先把参加交互的对象放在图的上方，沿 X 轴方向排列，通常把发起交互的对象放在左边，较下级对象依次放在右边，把这些对象发送和接收的消息沿 Y 轴方向按时间顺序从上到下放置，这样就提供了控制流随时间推移的清晰的可视化轨迹。

当一个顺序图规模变得较大时，众多的消息会使图变得非常拥挤，纵横交错，可读性会变得很差。此时，除非必须，一般的同步消息就不要画出相应的返回消息了。

三. 协作图：

是表示角色间交互的图，主要用来描述对象间的交互关系。



✧ 协作图是一种基于结构的表示交互的方法，强调参加交互的对象的组织。

1. 协作图的组成：对象、链和消息。

1）对象：

协作图与顺序图中对象的概念是一样的，只不过在协作图中无法表示对象的创建和撤销，所以对象在协作图中的位置没有限制。

2）链：

协作图中链的符号和对象图中链所用的符号是一样的，即一条连接两个角色的实线。

3）消息：

协作图中的消息类型与顺序图中的相同，只不过为了说明交互过程中消息的时间顺序，需要给消息添加顺序号。顺序号是消息的一个数字前缀，是一个整数，由 1 开始递增，每个消息都必须有惟一的顺序号。

2. 协作图的生成：

协作图使用对象图作为基础，产生一张协作图，首先要将参加交互的对象作为图的顶点；然后，把链接这些对象的链表示为图的狐；最后，用对象发送和接收的消息来修饰这些链，这就提供了在协作对象的结构组织的语境中观察控制流的一个清晰的可视化轨

迹。

四. 顺序图与协作图的比较：

顺序图和协作图都可以用来描述对象和交互行为。但是顺序图着重描述的是对象行为的时间特性，而协作图着重描述的是对象交互的空间特性。所谓时间特性即可以描述某个行为的发生时间和持续时间等，而空间特性则指与某个对象有交互行为关系的所以对象可以很方便地在协作图中表现出来。

1. 顺序图和协作图之间的相同点：

1）规定责任

两种图都直观地规定了发送对象和接收对象的责任，将对象确定为接收对象，意味着为此对象添加一个接口，而消息描述称为接收对象的操作特征标记，由发送对象触发该操作。

2）支持消息

两种图都支持所有的消息类型。

3）衡量工具

两种图还是衡量耦合性的工具。耦合性被用来衡量模型之间的依赖性，通过检查两个元素之间的通信，可以很容易地判断出它们的依赖关系，如果查看对象的交互图，就可以看见两个对象之间消息的数量以及类型，从而简化或减少消息的交互，以提高系统的设计性能。

2. 顺序图和协作图的区别：

1）顺序图和协作图都表示对象之间的交互作用，只是它们的侧重点不同。顺序图描述了交互过程中的时间顺序，但没有明确地表达对象之间的关系；协作图描述了对象之间的关系，但时间顺序必须从顺序号获得。协作图的重点是将对象的交互映射到它们之间的链上，即协作图以对象图的方式绘制各个参与对象，并且将消息和链平行放置。这种表示方法有助于通过查看消息来验证类图中的关联或者发现添加新的关联的必要性；但是顺序图却不把链表示出来，在顺序图的对象之间，尽管没有相应的链存在，但也可以随意绘制消息，不过这样做的结果是有些逻辑交互根本就不可能实际发生。

2）顺序图可以描述对象的创建和撤销的情况。新创建的对象可以被放在对象生命线上对应的时间点，而在生命线结束的地方放置一个大写的 X 以表示该对象在系统中不能再继续使用；而在协作图

中，对象要么存在要么不存在，除了通过消息描述或约束，没有其他的方法可以表示对象的创建或结束，但是由于协作图所表现的结构被置于静止的对象图中，所以很难判断约束什么时候有效。

3）顺序图还可以表现对象的激活和去激活情况，但对于协作图来说，由于没有对时间的描述，所以除了通过对消息进行解释，它无法清晰地表示对象的激活和去激活情况。

§6.2 交互建模

一. 交互建模

交互图用于对系统的动态方面建模，包括以下两个方面：

1. 按时间顺序对控制流建模：（使用顺序图）

按时间顺序对控制流建模，强调时间展开的消息的传送，这在一个用况脚本的语境中对动态行为可视化尤其有用，顺序图对简单的迭代和分支的可视化要比协作图好。

按时间顺序对控制流建模，要遵循如下策略：

1）设置交互的语境，这些语境可以是系统、子系统、操作、类、用例或协作的脚本。

2）通过识别对象在交互中扮演的角色，设置交互的场景，以从左到右的顺序将对象放到顺序图的上方，其中较重要的放在左边，与它们相邻的对象放在右边。

3）为每个对象设置生命线。通常情况下，对象存在于整个交互过程中。对于那些在交互期间创建和撤销的对象，在适当的时刻设置它们的生命线，并用适当的构造型消息显式地说明它们的创建和撤销。

4）从引发某个消息的信息开始，在生命线之间画出从顶到底依次展开的消息，显示每个消息的特性（如参数）。若有需要，解释交互的语义。

5）如果需要可视化消息的嵌套或实际计算发生时的时间点，可以用激活修饰每个对象的生命期。

6）如果需要说明时间或空间的约束，可以用时间标记修饰每个消息，并附上合适的时间和空间约束。

7）如果需要形式化地说明某控制流，可以为每个消息附上前置和后置条件。

2. 按组织对控制流建模：（使用协作图）

按组织对控制流建模，强调交互中实例之间的结构关系以及所传送的消息，协作图对复杂的迭代和分支的可视化以及对多并发控制流的可视化要比顺序图好。

按组织对控制流建模，要遵循如下策略：

1) 设置交互的语境，这些语境可以是系统、子系统、操作、类、用例或协作的脚本。
2) 通过识别对象在交互中扮演的角色，设置交互的场景，将对象作为图的顶点放在协作图中，其中较重要的对象放在图的中央，与它邻近的对象放在外围。
3) 对每个对象设置初始特性。如果某个对象的属性值、标记值、状态或角色在交互期间发生重要变化，则在图中放置一个复制的对象，并用这些新的值更新它，然后通过构造型<<become>>或<<copy>>的消息将两者连接。
4) 描述对象之间可能有信息沿着它传递的链。首先安排关联的链，这些链是最主要的，因为它们代表结构的连接，然后再安排其他的链，用适当的路径（如<<global>><<local>>）来修饰它们，显示地说明这些对象是如何相互联系的。
5) 从引起交互的消息开始，适当地设置其顺序号，然后将随后的每个消息附到适当的链上，可以用带小数点的编号来表示嵌套。
6) 如果需要说明时间或空间的约束，可以用时间标记修饰每个消息，并附上合适的时间和空间约束。
7) 如果需要更形式化地说明整个控制流，可以为每个消息附上前置和后置条件。

二. 交互模型的识别步骤：

1. 列出用例相关的所以对象（类）；
2. 根据用例活动确定对象间的消息通讯；
3. 定义对象间的消息连接和消息格式；
4. 确定消息发生的时间顺序；
5. 确定消息编号（只对协作图）；
6. 画出交互模型。

三. 使用顺序图和协作图的建议：

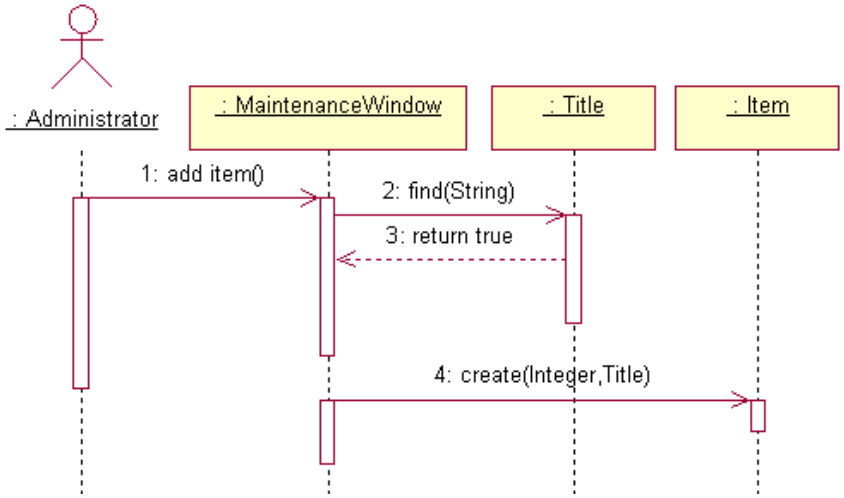
1. 如果对象数目不多，交互情况也不复杂，顺序图和协作图可以相互替换；

2. 如果系统关系对象交互行为的时间特性，应该选择顺序图；
3. 如果对象数目很多，且交互情况较复杂，可能使用协作图，但是其中的某些“场景片段”可以使用顺序图来专门描述其时间特性。

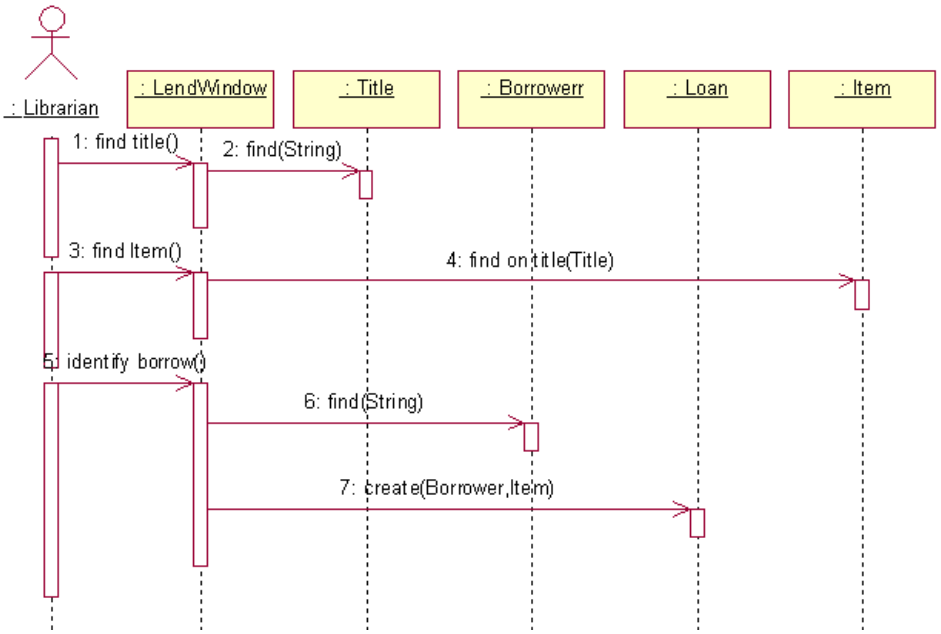
§6.3 案例分析：《图书馆管理系统》

一. 顺序图：

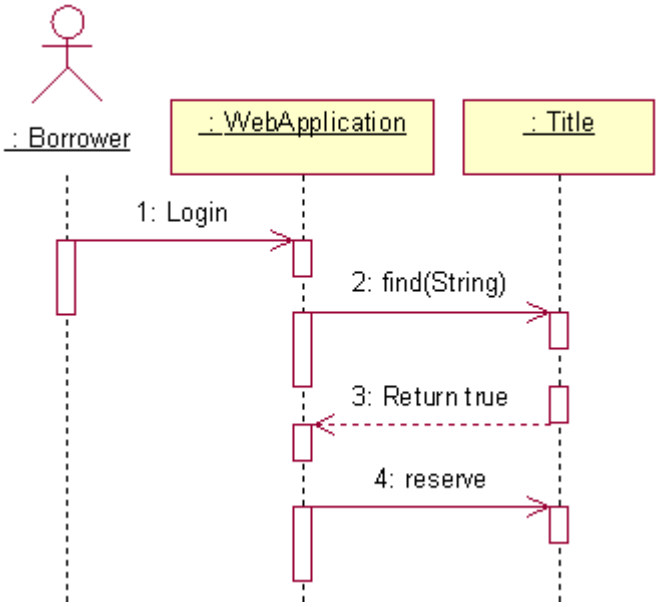
1. 系统管理员添加书籍：



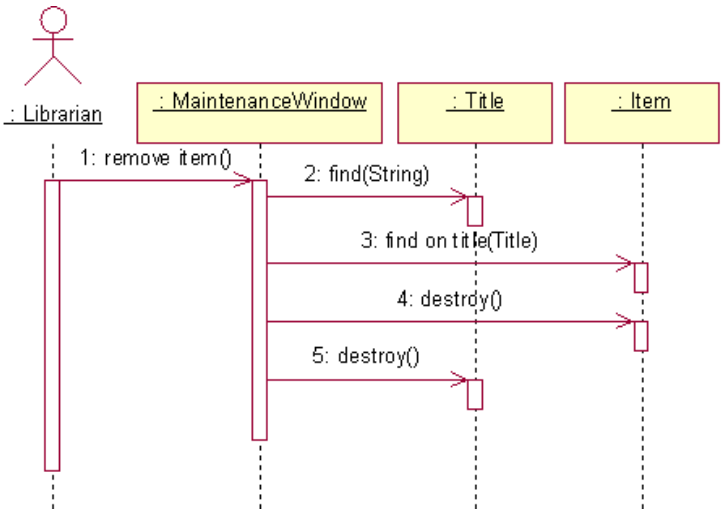
2. 图书馆管理员处理书籍借阅：



3. 系统管理员删除书目：

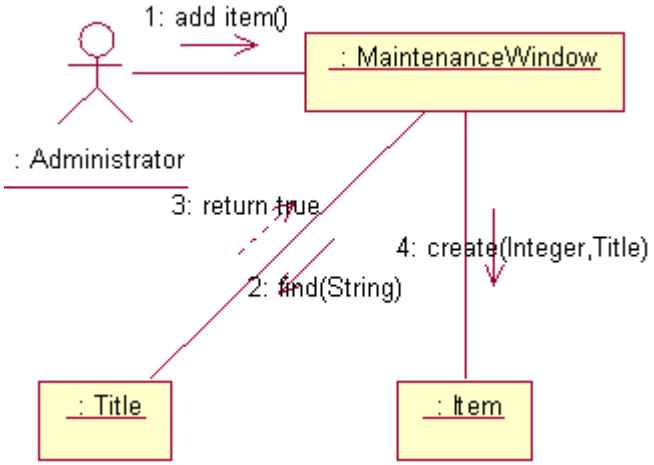


4. 借阅者预订书籍：

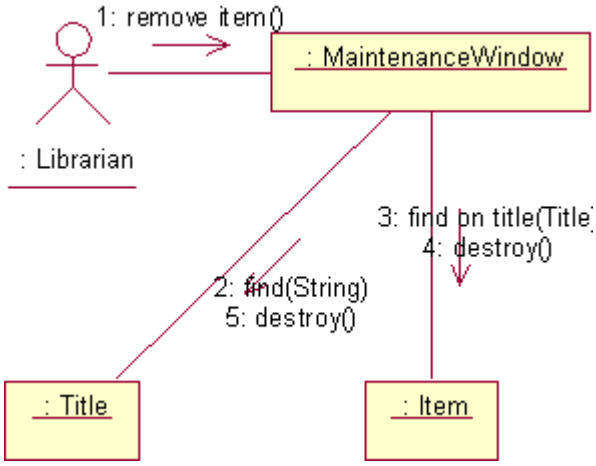


二. 协作图：

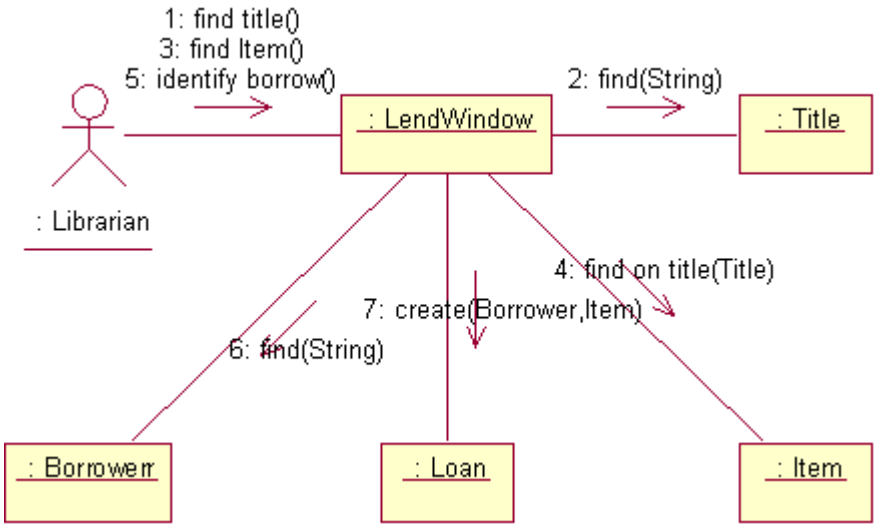
1. 系统管理员添加书籍：



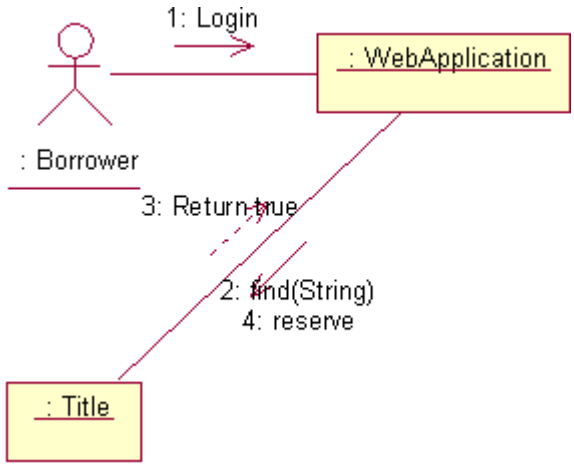
2. 图书馆管理员处理书籍借阅：



3. 系统管理员删除书目：



4. 借阅者预订书籍：



§6.4 补充案例：《订货中心系统》

一. 用例说明：

用例：订货

1. 购买者提出购买请求（包括货物品名、数量）
2. 公司记下购买者信息（包括姓名、地址、电话）
3. 公司告知购买者货物的规格、价钱
4. 购买者确认订货
5. 公司产生一个新的订单
6. 购买者付款
7. 公司送货给购买者

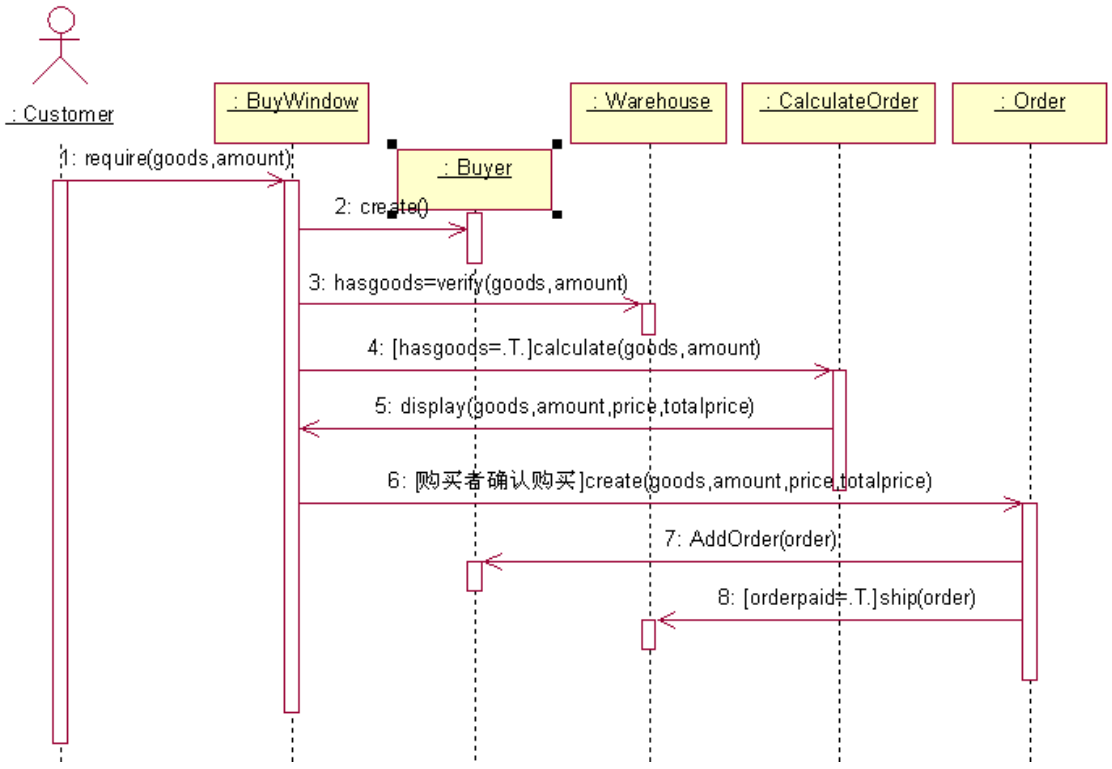
二. 对象识别：

1. 购买窗口对象（:Buywindow）
2. 购买者（:Buyer）
3. 仓库（:Warehouse）
4. 订单（:Order）
5. 计算订单（:CalculateOrder）

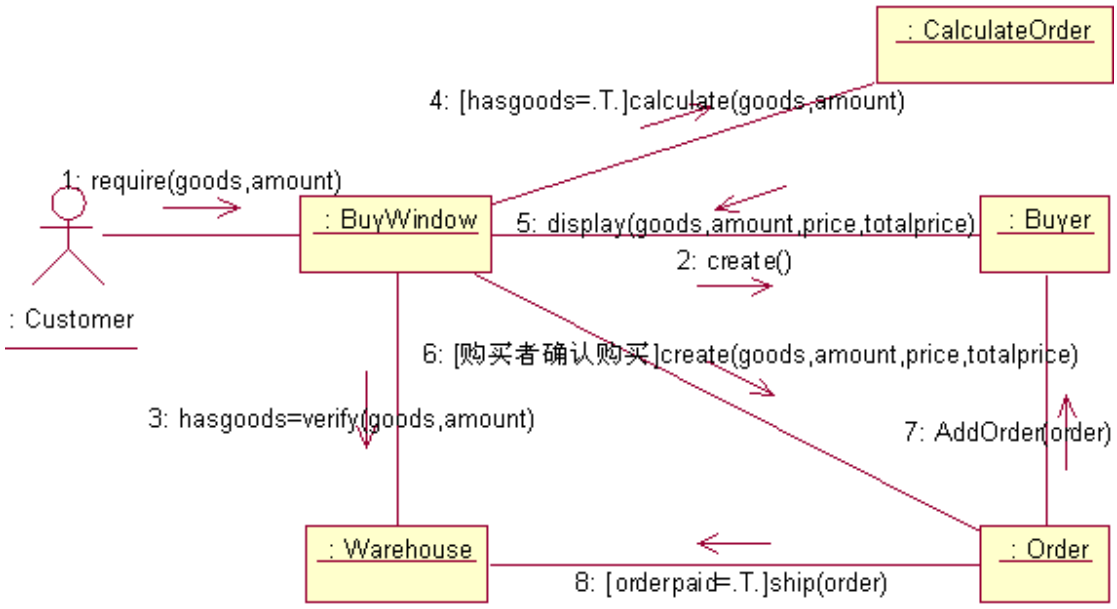
三. 消息识别：

四. 交互图：

1. 顺序图：



2. 协作图：


: CalculateOrder
4: [hasgoods=.T.]calculate(goods,amount)
1: require(goods,amount)
: BuyWindow
5: display(goods,amount,price,totalprice)
: Buyer
2: create()
: Customer
6: [购买者确认购买]create(goods,amount,price,totalprice)
3: hasgoods=verify(goods,amount)
7: AddOrder(order)
: Warehouse
: Order
8: [orderpaid=.T.]ship(order)

§ 6.5 小结

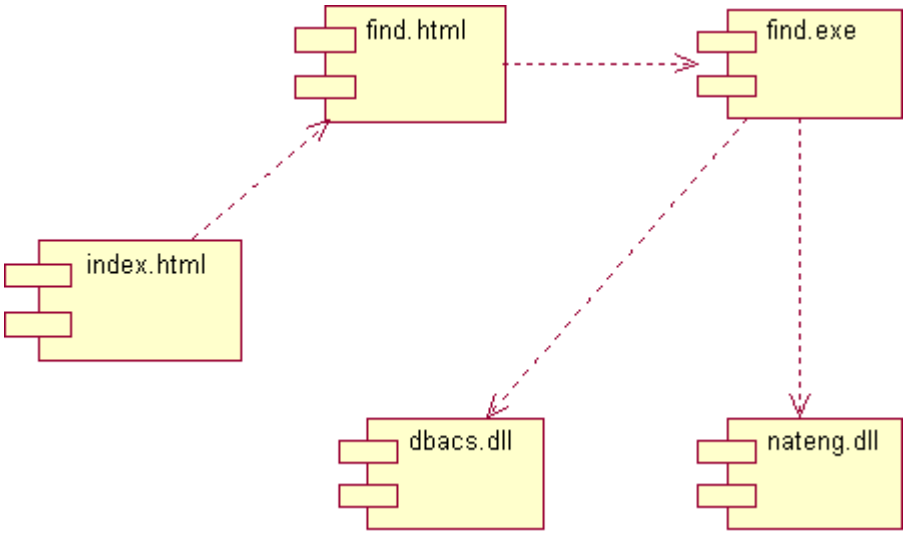
§ 6.6 补充实例

第七章　构件图

§7.1 基本概念

一. 构件图：

显示一组构件及它们之间的关系的图。



1. 内容：
   1）构件
   2）接口
   3）依赖关系
2. 应用：

   构件图用于对系统的静态实现视图建模，在对系统的静态实现视图建模时，通常按下列四种方式之一来使用构件图：

   1）对源代码建模

   采用当前大多数面向对象编程语言，将使用集成化开发环境来分割代码，并将源代码存储到文件中，可以使用构件图来为这些文件的配置管理建模，这些文件代表了产品构件。

   2）对可执行体的发布建模

   软件的发布是交付给内部或为外部用户使用的相对完整而且一致的制品系列，在构件语境中，一个发布注重交付一个运行系统所必需的部分。当用构件图对发布建模时，其实是在对构成软件的物理部分（即实施构件）所做的决策进行可视化、详述、

文档化。

3) 对物理数据库建模

可以把物理数据库看作模式在比特世界中的具体实现，实际上模式提供了对永久信息的应用程序编程接口（API）；物理数据库模型表示了这些信息在关系型数据库的表中或者在面向对象数据库的页中的存储，可以用构件图表示这些以及其他种类的物理数据库。

4) 对可适应的系统建模

某些系统是相对静态的；其构件进入现场，参与执行，然后离开。另一些系统则是较为动态的，其中包括一些活动代理或者为了负载均衡和故障恢复而进行迁移的构件。可以将构件图与对行为建模的 UML 的一些图结合起来表示这类系统。

二. 构件：

构件是系统中遵从一组接口且提供其实现的物理的、可替换的部分。

✧ 构件是物理的

它存在于二进制世界中，而不存在于概念世界中。

✧ 构件是可替换的

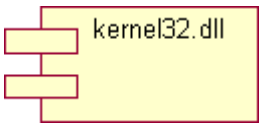
一个构件可以用其他遵从相同接口的构件来替换。

✧ 构件是系统的一部分

构件很少单独存在，一个给定的构件通常与其他构件协作并存在于打算使用它的体系结构或技术语境中。构件在逻辑上和物理上都是内聚的。构件可在多个系统中复用，一个构件表示一个可以以它为基础进行系统设计和组装的基本构造块，这个定义是递归的，即某个抽象级别的系统可以作为更高一层抽象级别系统的构件。

✧ 构件遵从一组接口并提供对一组接口的实现。

1. 符号：



2. 分类：

1) 实施构件：

这类构件是构成一个可执行系统必要和充分的构件，如：动态链接库(DLL)和可执行体(EXE)。

2) 工作产品构件：

这类构件主要是开发过程的产物，包括创建实施构件的源代码文件及数据文件等。这些构件并不是直接地参加可执行系统，而是开发过程中的工作产品，用来产生可执行系统。

3) 执行构件：

这类构件是作为一个正在执行的系统的结果而被创建的，如：由 DLL 实例化形成的 COM+对象。

3. 应用于构件的构造型：

UML 中定义了 5 种标准的应用于构件的构造型：

1) 可执行体(executable)：指定能够在一个节点上执行的构件。
2) 库(library)：指定一个动态或静态的对象库。
3) 表(table)：指定一个表示数据库表的构件。
4) 文件(file)：指定一个表示文档的构件，其中包括源代码或数据。
5) 文档(document)：指定一个表示文档的构件。

4. 构件和类的关系：

构件在许多方面与类相同：二者都有名称；都可以实现一组接口；都可以参与依赖、泛化和关联关系；都可以被嵌套；都可以有实例；都可以参与交互。但是构件和类之间也有一些显著的差别：

1) 类表示逻辑抽象，而构件表示存在于比特世界中的物理抽象。简言之，构件可以存在于节点上，而类不可以。

当对系统建模时，决定采用构件还是采用类涉及一个简单的决策——当准备建模的事物直接存在于节点上时，采用构件；否则，采用类。

2) 构件表示的是物理模块而不是逻辑模块，与类处于不同的抽象级别。

构件是一组其他逻辑元素（如类及其协作关系）的物理实现，构件与其所实现的类之间的关系可用依赖关系显示地表示，多数情况下，不必通过图形方式可视化这些关系，而是将这些关系作为构件规格说明的一部分。

3) 类可以直接拥有属性和操作；一般情况下，构件仅拥有只能通过其接口访问的操作。

接口是构件和类之间的桥梁，虽然构件和类都可以实现一个接口，但是构件的服务一般只能通过其接口来访问。

5. 构件的构建：

当在 UML 中对构件建模时，注意是在物理维上建模。一个结果良好的构件，应满足如下的要求：

1）提供从系统的物理方面抽取的一些事物的明确抽象。

2）提供对一组小的、定义完好的接口的实现。

3）经济有效地直接实现一组共同工作以完成这些接口语义的类。

4）相对其他构件是松散耦合的，通常对构件建模一般只涉及依赖关系和实现关系。

当在 UML 中绘制一个构件时，要遵循如下的策略：

1）除非有必要显式地展示接口提供的操作，否则一般只需以图符方式来展示。

2）仅显示在给定语境中对理解构件的含义是必要的那些接口。

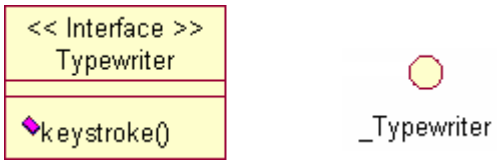
3）当用构件为库和源代码建模时，显示与版本有关的标记值。

三．接口：

接口是一个用来描述一个类或一个构件所提供的服务的操作集合。

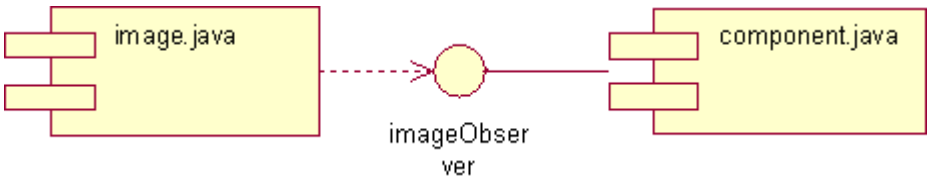
- ✧ 接口可以作为把构件绑定在一起的粘合剂。
- ✧ 可以通过说明代表系统中主要衔接点的接口来分解系统的物理实现，然后提供实现这些接口的构件以及通过这些接口访问其服务的其他构件，这种机制使得能实施那些服务有点独立于位置且可替换的系统。
- ✧ 接口跨越了逻辑和物理边界，同一个接口，如果它被一个构件使用或实现，那么它肯定被该构件所实现的类应用或实现。

1. 符号：

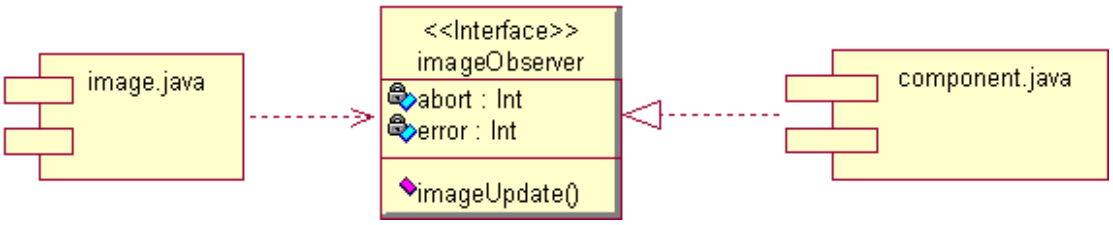


2. 构件和接口关系的表示：

1）用简略的图符形式显示接口，实现接口的构件用一个简略的实现关系连接到接口上



2）用扩展形式显示接口，这种方式可以显示接口的操作，实现接口的构件用一个完整的实现关系连接到接口上。



3. 接口类型：

1）示出接口：

构件实现的接口叫做示出接口，即构件提供的为其他构件服务的接口。一个构件可以提供多个示出接口。

2）引入接口：

构件使用的接口叫做引入接口，即应用它的构件必须遵从这个接口并以此为基础进行构造。一个构件可遵从多个引入接口。

✧ 构件可以既示出接口又引入接口。

✧ 一个给定的接口可以由一个构件示出，也可以被另一个构件引入，因此接口位于两个构件中间而断开了它们之间的直接依赖关系。不管接口是采用什么构件实现的，使用这个给定接口的构件都能正常运行。当然，一个构件当且仅当它所引用的所有引入接口都能由其他构件的示出接口提供时，才能应用于相应语境中。

§ 7.2 构件图建模

一. 对可执行体和库建模：

使用构件的最普遍的目的是对构成系统实现的实施构件建模（如果所建立的系统很小，它的实现由恰好一个可执行程序构成，那么就不需要进行构件建模；但如果要实施的系统由几个可执行体和几个相关对象库构成，则进行构件建模将有助于可视化、详述、构造和文档化对物理系统所做的决策。如果想在系统演化中对这些组成部分进行版本控制和配置管理，则构件建模甚至更为重要），对于大多数系统，这些实施构件来源于划分系统物理实现的决策。

1. 对可执行体和库建模，要遵循如下的策略：

1）确定如何划分你的物理系统。要考虑技术、配置管理及复用问题的影响。

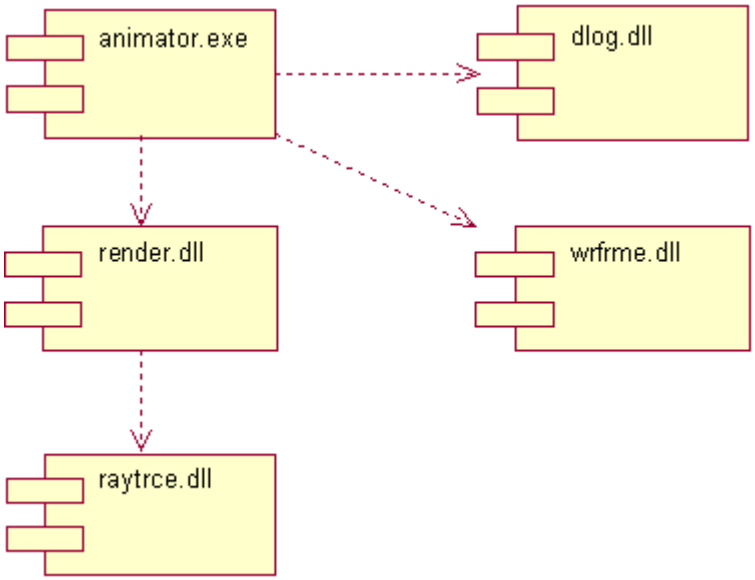
2）使用合适的标准元素，将可执行体和库建模为构件。如果实现中

引入了新的构件类型，则引入相应构造型。

3）如果管理系统中的衔接很重要，就要对由一些构件使用并由另一些构件实现的重要接口建模。

4）按交流意图的需要，对这些可执行体、库及接口之间的关系建模。通常需要对这些构件之间的依赖关系建模，以可视化系统变化的影响范围。

2. 示例：



3. 说明：

直接显示两个构件之间的依赖关系其实是实际构件之间关系的简略视图，一个构件很少直接依赖另一个构件，而通常是引入由其他构件示出的一个或多个接口，为了简单明了，可以忽略这些细节而仅显示构件之间的依赖关系。

二. 对表、文件和文档建模：

对构成系统的物理实现的可执行体和库建模是有用的，但经常会发现还有很多既不是可执行体也不是库的附属实施构件，它们对于系统的物理配置是至关重要的。例如，系统实现中可能包括数据文件、帮助文档、脚本、日志文件、初始化文件及安装/卸载文件等。对这些构件建模是控制系统配置的重要组成部分。

1. 对表、文件和文档进行建模，要遵循如下的策略：

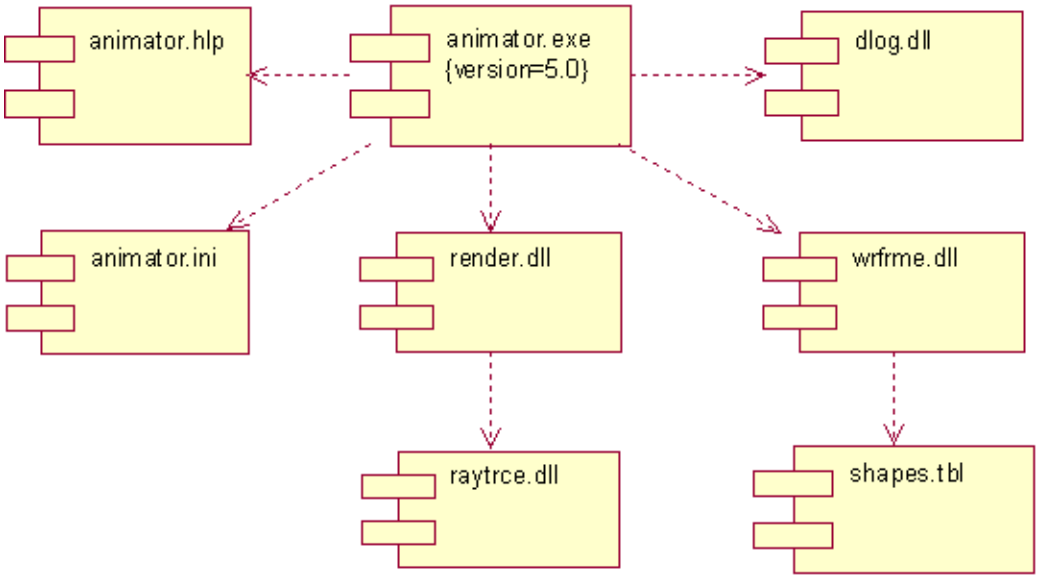
1）识别出作为系统的物理实现部分的附属构件。

2）将这些事物建模为构件。如果实现中引入了新的制品种类，则引入适当的新构造型。

3）按交流的意图的需要，对这些辅助构件与其他可执行体、库及接口之间的关系建模。通常，为了可视化构件变化的影响范围，需

要对这些部分之间的依赖关系建模。

2. 示例：



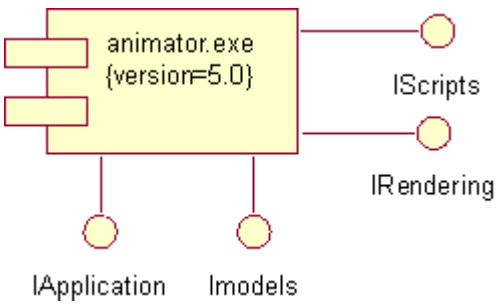
三. 对 API 建模：

API 表示系统中的可编程接缝，可以用接口和构件对它建模。

API 本质上是一个由一个或多个构件实现的接口。作为开发人员，只需要真正地关心接口本身，而不需要了解哪个构件或哪几个构件实现该接口的操作；从系统配置管理角度看，这些 API 的实现是重要的，因为当公布一个 API 时，需要确保存在某些构件能完成 API 的责任。

1. 对 API 建模，要遵循如下的策略：
   1) 识别系统中的可编程接缝，将每个接缝建模为一个接口，并收集形成其边界的属性和操作。
   2) 只显露那些对于在给定语境中对可视化来说时比较重要的那些接口特性，隐藏那些不重要的接口特性，必要时可以将这些特性保持在接口的规格说明中作为参考。
   3) 对每个 API 的实现建模限于这种程度：当它对于展示特定实现的配置重要时才对其建模。

2. 示例：



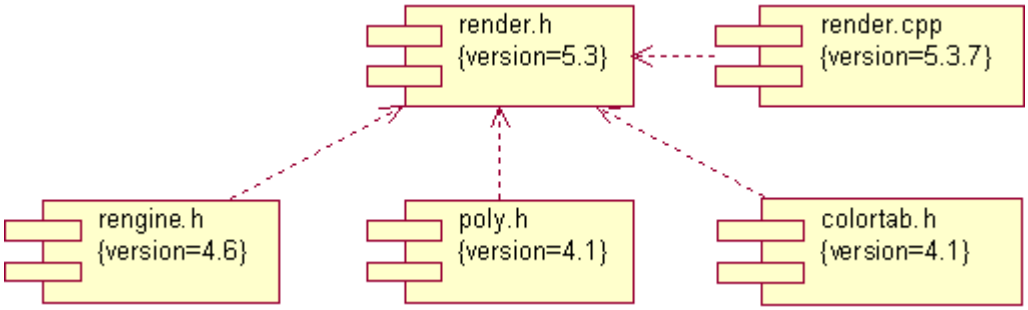
四. 对源代码建模：

使用构件的另一个最普遍的目的是对开发工具用来产生这些构件的所有源代码文件的配置建模，它们表示开发过程中的工作产品构件。

对源代码的图形化建模特别有助于可视化源代码文件之间的编译依赖关系，并在划分或汇合开发路径时管理文件组的分割和合并。采用这种方式，UML 构件可作为到配置管理及版本控制工具的图形接口。

对于大多数系统，源代码文件来源于如何划分开发环境所需文件的决策，源代码文件用来存储类、接口、协作和其他逻辑模型元素的细节，作为由工具产生物理的、二进制的构件的中间环节。

1. 对源代码建模，要遵循如下的策略：
   1) 根据开发工具施加的约束，对存储所有逻辑元素以及它们之间的编译依赖关系细节的文件建模。
   2) 如果对这些模型进行配置管理和版本控制是重要的，则对每一个需要配置管理的文件加进一些标记值，如：版本、作者和检入/检出信息校验等。
   3) 尽可能使用开发工具管理这些文件之间的关系，并只用 UML 来可视化和文档化这些关系。
2. 示例：



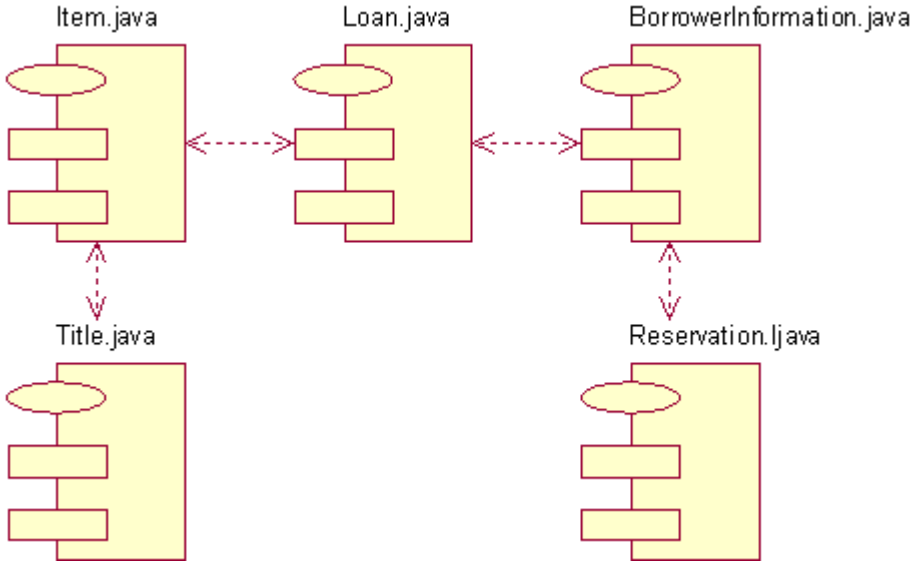
五. 构件图建模的要求：

1. 一个结构良好的构件图，应满足如下的要求：
   1) 侧重于描述系统的静态实现视图的一个方面。
   2) 只包含对理解这一方面是必要的那些模型元素。
   3) 提供与其抽象层次一致的细节，只显露对于理解是必要的那些修饰。
   4) 图形不要过于简化，以至于使读者对重要语义产生误解。
2. 当绘制一个构件图时，要遵循以下的策略：

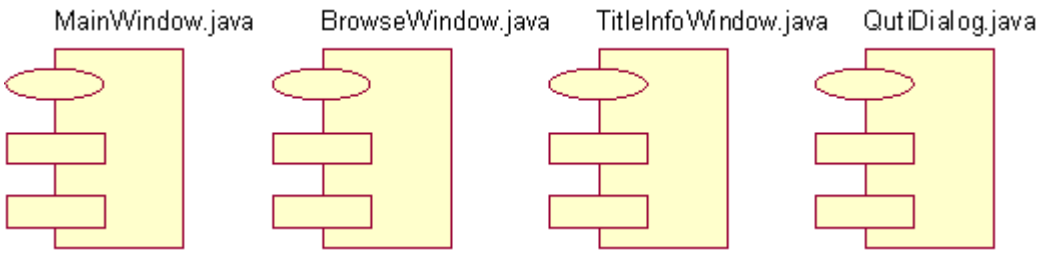
1）为构件图取一个能表示其意图的名称。
2）摆放元素时应尽量避免线的交叉。
3）在空间上合理地组织图的元素，使得语义上接近的事物物理位置上也比较接近。
4）用注解和颜色作为可视化提示，以把注意力吸引到图中的重要特征上。
5）谨慎地采用构造型元素。为项目或组织选择少量通用图标，并在使用它们时保持一致。

§7.3 案例分析：《图书馆管理系统》

一. 业务对象组件图：



二. 用户界面组件图：



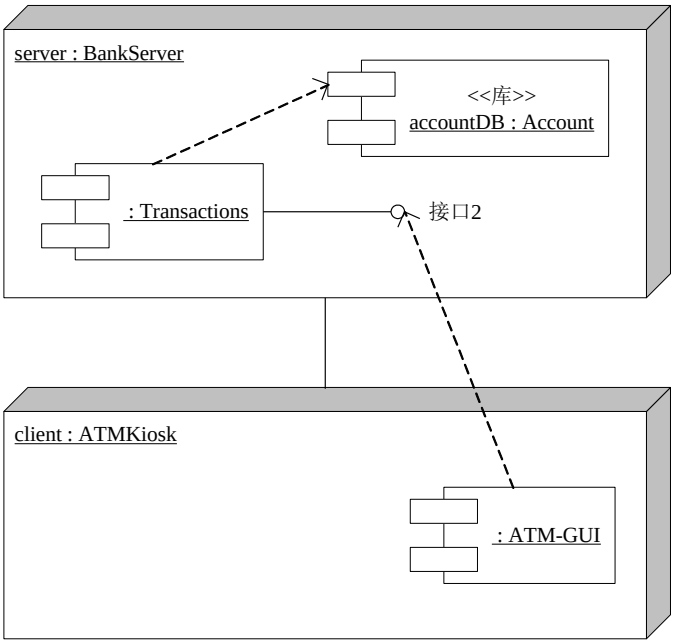
§7.4 小结

第八章　实施图

§8.1 基本概念

一. 实施图：

是一种图，它显示运行时进行处理的节点和在节点上活动的构件的配置



1. 组成：
   1）节点
   2）连接
2. 应用：

   实施图用于对系统的静态实施视图建模，这种视图主要用来描述构成物理系统的各组成部分的分布、提交和安装。

   对系统静态实施视图建模时，通常将以下列三种方式之一使用实施图：

   1）对嵌入式系统建模：

   嵌入式系统是软件密集的硬件集合，其硬件与物理世界连接。嵌入式系统包括控制设备（如：马达、传动装置和显示器）的软件，又包括由外部的刺激（如：传感器输入、运动和温度变化）所控制的软件。可以用实施图对组成一个嵌入式系统的设备

和处理器建模。

2）对客户/服务器系统建模：

客户/服务器系统是一种常用的体系结构，它注重于将系统的用户界面（在客户机上）和系统的永久数据（在服务器上）清晰地分离开。客户/服务器系统是分布式体系的一个极端，它要求对客户/服务器之间的网络连接以及系统中的软件构件在节点上的物理分布作出决策。可以用实施图对这种客户/服务器系统的拓扑结构建模。

3）对全分布式系统建模：

分布式系统的另一个极端是广泛的分布式系统，它通常由多级服务器构成。这种系统中一般存在着多种版本的软件构件，其中有一些版本的软件构件甚至还可以在节点间迁移。精心地构造这样的系统，需要对系统拓扑结构的不断变化作出决策。可以用实施图可视化系统的当前拓扑结构及构件的分布情况，并推断拓扑结构变化的影响。

二. 节点：

是存在于运行时并代表一项计算资源的物理元素，一般至少拥有一些内存，而且通常具有处理能力。

1. 符号：

2. 节点和构件：

节点在许多方面与构件相同：二者都有名称；都可以参与依赖、泛化和关联关系；都可以被嵌套；都可以有实例、都可以参与交互。但是节点和构件之间也有一些显著的差别：

1）构件是参与系统执行的事物，而节点是执行构件的事物。

节点执行构件，构件是被节点执行的事物。

2）构件表示逻辑元素的物理打包，而节点表示构件的物理部署。

一个构件是一组其他逻辑元素（如：类和协作）的物质化实现，而一个节点是构件被部署的地点。一个类可以被一个或多个构件实现，而一个构件可以被部署在一个或多个节点上。节点与它所部署的构件之间的关系可用依赖关系显式地加以表示。

三. 连接：

节点之间使用的最常见的关系是关联关系，在这种语境中，关

联表示节点之间的物理连接。

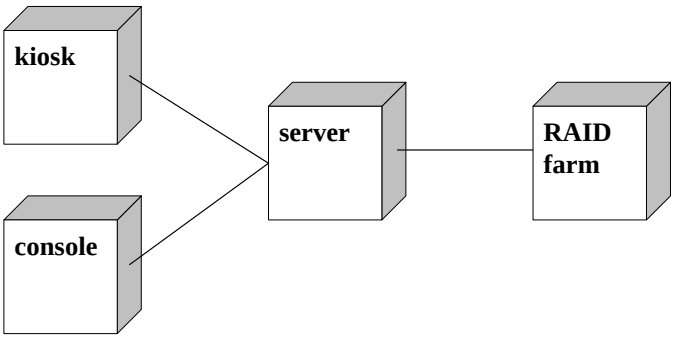
§8.2 实施建模

一. 节点建模：

1. 对处理器和设备建模：

节点的最普遍的用处是对形成单机式、嵌入式、客户/服务器式和分布式系统的拓扑结构的处理器和设备进行建模。

- ✧ 处理器：是一个具有处理能力的节点，即它可以执行构件。
- ✧ 设备：是一个没有处理能力的节点，并通常表示某些与现实世界衔接的事物。

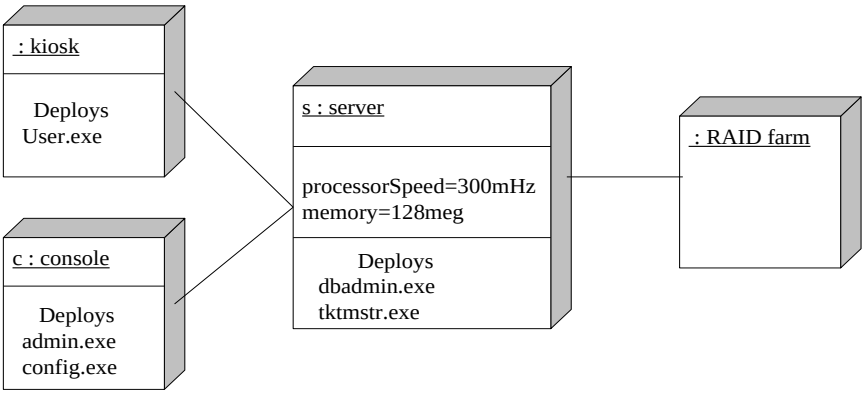


对处理器和设备建模，要遵循如下的策略：

1) 识别系统的实施视图中的计算元素，并将每个计算元素建模为节点。
2) 如果这些模型元素代表一般的处理器和设备，则按照原样将它们构造型化；如果它们是领域的词汇中的某种处理器和设备，则为它们定义相应的带图符的构造型。
3) 与对类建模相似，考虑可以应用于每一个节点的属性和操作。

2. 对构件的分布建模：

当对系统的拓扑结构建模时，可视化或指定其构件在构成系统的处理器或设备上的物理分布通常是有用的。



对构件的分布建模，要遵循如下的策略：

1）对系统中每个有意义的构件，将其分配到一个给定的节点上。

2）考虑构件在节点上的重复放置。同种构件（如：某种可执行体和库）同时存在于多个不同节点上是很常见的。

3）将这种分配用下述三种方式之一表示出来：

- 不使分配成为可见的，但要保留它们作为模型的基架的一部分——即保留在每一个节点的规格说明中。
- 使用依赖关系，将每一个节点和它上面所部署的构件连接起来。
- 在附加栏中列出节点上所部署的构件。

3. 节点建模的要求：

一个结构良好的节点，应满足如下的要求：

1）提供对求解空间硬件词汇中提取的事物的明确抽象。

2）仅需要分解到向读者交流意图所必需的程度即可。

3）仅显露与建模的领域有关的那些属性和操作。

4）直接部署存在于节点上的一组构件。

5）采用反映现实世界系统拓扑结构的方式，将这个节点与其他节点连接。

当在 UML 中绘制一个节点时，要遵循如下的策略：

1）为整个项目或组织定义一组用户容易理解的带图符的构造型，以便向用户提供意义鲜明的可视化提示。

2）仅显示对理解节点在给定语境中的含义是必要的那些属性和操作（如果有）。

二. 实施图建模：

1. 对嵌入式系统建模：

开发一个嵌入式系统远远不只是软件的问题，还必须管理物理世界，对这样的系统建模时，要考虑它与现实世界的接口，这意味着要考虑特殊的设备和节点。

实施图为项目的硬件工程师和软件开发者之间的交流提供了方便，通过使用已被构造型化以使其外观上很像大家熟悉设备的节点，可以建立软硬件工程师都能理解的图，实施图还有助于软硬件之间的折衷。

对嵌入式系统建模，要遵循如下的策略：

1）识别对于系统是唯一的设备和节点。

2) 使用 UML 的扩展机制为系统中的设备，特别是特殊的设备定义带有合适图标的系统专用的构造型，以提供可视化提示。至少要把处理器（其中含有系统构件）和设备区分开来。

3) 在实施图中对处理器和设备之间的关系建模。类似地，说明系统实现视图中的构件和系统实施视图中的节点之间的关系。

4) 如果需要，可以把任何智能设备展开，用更详细的实施图对它的结构建模。

2. 对客户/服务器系统建模：

当开始开发一个其软件要运行在多个处理器上的系统时，将不得不面对许多决策问题：如何将软件构件最佳地分布在各个节点上？它们之间如何通信？以及如何处理失败和噪音问题？作为分布式系统的一个极端，将会遇到客户/服务器系统，其中系统的用户接口（通常由客户机管理）和数据（通常由服务器管理）之间有明显的职责划分。

将系统划分为客户部分和服务器部分都要涉及一些关于在物理上将软件构件放在何处以及如何在这些构件之间到达职责平衡分布的困难决策。

对客户/服务器系统建模，要遵循如下的策略：

1) 识别代表系统中的客户和服务器处理器的节点。

2) 标识出与系统行为有密切关系的设备。

3) 通过进行构造型化，为这些处理器和设备提供可视化提示。

4) 在实施图中为这些节点的拓扑结构建模，并说明系统实现视图中的构件与系统实施视图中的节点之间的关系。

3. 对全分布式系统建模：

可视化、描述和文档化全分布式系统的拓扑结构对于系统管理员来说是很有价值的活动，因为它们必须保持企业计算资源的一个列表，可以利用 UML 中的实施图来为这类系统的拓扑结构建模。当用实施图文档化全分布式系统时，可能要展开系统网络设备的细节，将每台设备表示为构造型化的节点。

对全分布式系统建模，要遵循如下的策略：

1) 像对待较简单的客户/服务器系统那样，识别出系统中的设备和处理器。

2) 如果需要刻画系统网络的性能或者网络变化带来的影响，那么对这些通信设备建模，一定要达到足以做出这些估计的程度。

3）特别注意节点的逻辑分组，它可以通过使用包来描述。

4）用实施图来对设备和处理器建模。尽可能使用工具遍历系统的网络以发现系统的拓扑结构。

5）如果要着眼于系统的动态方面，则引进用况图以描述所感兴趣的行为类型，并利用交互图来展开用况。

4. 实施图建模的要求：

当在 UML 中创建实施图时，记住每一个实施图只是系统静态实施视图的一个图形表示，这意味着每一个实施图都不必捕获系统实施视图的所有内容。

一个结构良好的实施图，应满足如下的要求：

1）侧重于描述系统的静态实施视图的一个方面。

2）只包含对理解这个方面是必要的那些元素。

3）提供与抽象级别一致的细节，只显露对于理解问题是必要的那些修饰。

4）不要过分简化，以免使读者对重要语义产生误解。

当绘制一个实施图时，要遵循如下的策略：

1）取一个能边上其意图的名称。

2）摆放元素是尽量避免线的交叉。

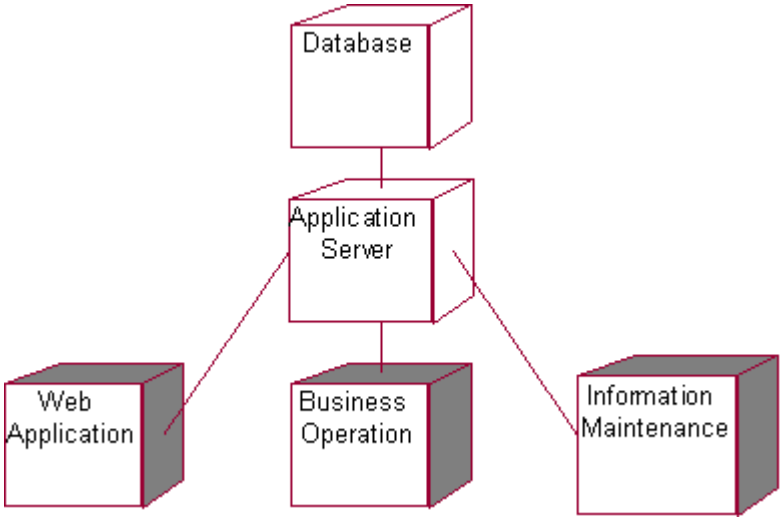
3）从空间上合理组织模型元素，使得语义上接近的事物物理位置上也比较接近。

4）用注解和颜色作为可视化提示，以把注意力吸引到图中的重要特征上。

5）谨慎地使用构造型化元素。为项目或组织选择少量通用图标，并在使用它们时保持一致。

§8.3 案例分析：《图书馆管理系统》

图书馆管理系统中的实施图：


Database
Application Server
Web Application
Business Operation
Information Maintenance

§8.4 小结