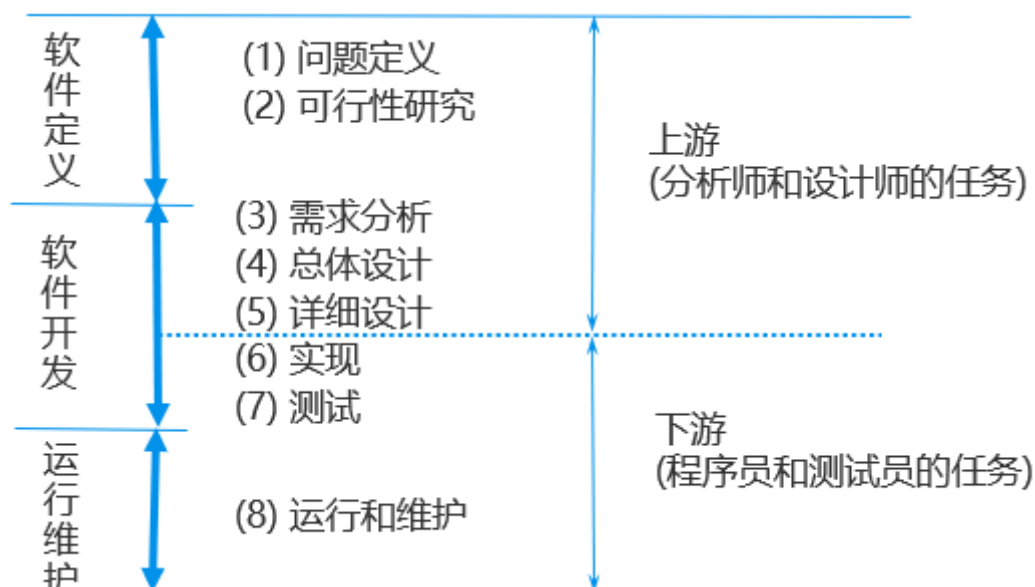


* 概述

- 什么是软件工程？就是用工程化的方法来开发软件
- 软件工程的目标：创造足够好的软件
- 软件工程定义：软件工程是指导计算机软件开发和维护的一门工程学科。采用工程的概念、原理、技术和方法来开发与维护软件，把经过时间考验而证明正确的管理技术和当前能够得到的最好的技术方法结合起来，以经济地开发出高质量的软件并有效地维护它。
- 软件工程过程 是指在生命周期内，为了实现特定目标而进行的一系列相关活动
- 软件工程方法 包含软件开发方法、软件度量方法、软件管理方法和软件环境方法
- 软件=程序+数据+文档
- 软件的本质特征：复杂性、易变性、不可见性、一致性
- 软件危机：在计算机软件的开发和维护过程中所遇到的一系列严重问题
- 软件开发过程：问题定义、需求开发、软件设计、软件构造、软件测试
- 软件生存期：可行性研究和项目开发计划、需求分析、概要设计、详细设计、软件构造、测试、维护



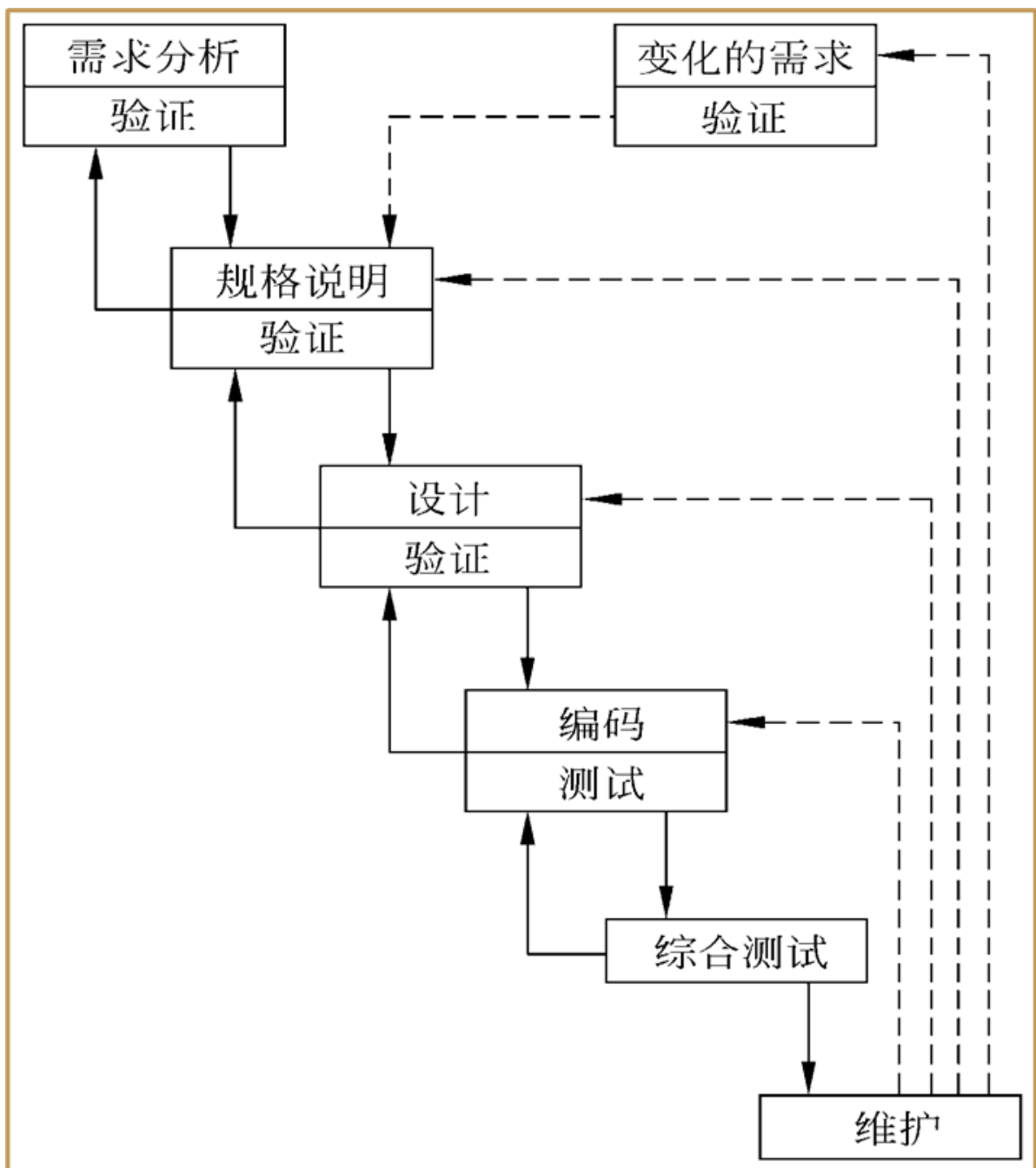
❖ 瀑布模型

适合于用户需求明确、完整、无重大变化的软件项目开发

i 是一种以文档驱动模型，每阶段任务完成之后，产生相应的文档

每项活动实施的工作进行评审，强调开发的阶段性、早期计划和需求调查以及产品测试等环节

缺点：由于是一种理想的线性开发模式，无法解决需求不明确或者变动的问题



* 快速原型模型

适用于用户不能给出完整、准确的需求说明，或者开发者不能确定算法的有效性、操作系统的适应性或人机交互的形式等许多情况下，可以根据用户的一组基本需求，快速建造一个原型

4步：快速分析

构造原型

运行和评价原型

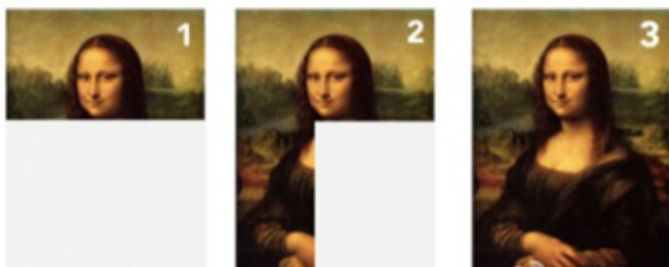
修改与改进

* 增量模型

渐进地开发逐步完善的软件版本的模型。把待开发的软件系统模块化，将每个模块作为一个增量组件，从而分批次地分析、设计、编码和测试这些增量组件

i 在每一个新的发布中逐步增加功能直到构造全部功能

增量模型



增量模型

优点：


- 整个产品被分解成许多个增量构件，开发人员可逐步开发。

- 在较短时间内向用户提交可完成部分工作的产品，并分批、逐步地提交。从第一个构件交付之日起，用户就能做一些有用的工作。
- 以组件为单位进行开发降低了软件开发的風險。一个开发周期内的错误不会影响到整个软件系统。
- 开发顺序灵活。开发人员可以对组件的实现顺序进行优先级排序，先完成需求稳定的核心组件。当组件的优先级发生变化时，还能及时地对实现顺序进行调整。
- 逐步增加产品的功能可以使用户有较充裕的时间学习和适应新产品，从而减少一个全新的软件可能给客户组织带来的冲击。

困难：

- 在把每个新的增量构件集成到现有软件体系结构中时，必须不破坏原来已经开发出的产品。此外，必须把软件的体系结构设计得便于按这种方式进行扩充，向现有产品中加入新构件的过程必须简单、方便，也就是说，软件体系结构必须是开放的。
- 开发人员既要把软件系统看作整体，又要看成可独立的构件，相互矛盾。多个构件并行开发，具有无法集成的风险。
- 采用增量模型比采用瀑布模型和快速原型模型需要更精心的设计，但在设计阶段多付出的劳动将在维护阶段获得回报。

✧ 螺旋模型

- 适用于规模较大的复杂系统、
-  风险驱动
- 螺旋模型将瀑布模型和增量模型结合起来，加入了风险分析
- 螺旋模型的基本思想是降低风险

可以看成在每个阶段之前都增加了风险分析过程的快速原型模型

螺旋模型把开发过程分为制定计划、风险分析、实施工程和客户评估

* 喷泉模型

主要用于支持面向对象开发过程、迭代

i 以用户需求为动力，以对象作为驱动的模式

* 迭代模型

一开始提交一个完整系统，在后续发布中补充完善各子系统功能



* RUP (Rational Unified Process, 统一过程模型)

- RUP重复一系列周期，每个周期由一个交付给用户的产品结束。
- 每个周期划分为 初始、细化、构造和移交 四个阶段，每个阶段围绕着九个核心 workflow 分别迭代。
- 核心支持 workflow：环境、项目管理、管理与变更管理
- 核心过程 workflow：部署、测试、实现、分析与设计、业务建模

① 初始阶段：进行问题定义，确定目标系统范围，评估其可行性，降低关键风险。******相当于三段生命周期模型中的计划时期。

- ② 细化阶段：主要完成领域问题分析和软件的设计，配置各类资源、建立系统架构（包括各类视图）。
- ③ 构造阶段：该阶段是产品的制造过程，以系统实现和测试为主，重点是管理资源及控制运作以优化成本、进度和质量。
- ④ 移交阶段：**产品发布、安装、用户培训。**重点是确保软件对最终用户是可用的。



特点：风险驱动，基于UseCase技术的、以架构为中心的、迭代的、可配置的软件开发流程

✧ 敏捷开发

以人为核心、循环迭代、响应变化的特点，着眼于高质量的快速交付令客户满意的工作软件



开发过程以代码为核心，而不是以文档为核心

敏捷开发将整个软件生命周期分为多个小的迭代（2-4周）每次迭代就是一个小的瀑布模型，结束时都要生成一个稳定和被验证过的软件版本

✧ 极限编程

以编码为核心任务的灵活软件开发方法

XP的目标：在最短的时间内将较为模糊、变化较大的用户需求转化为符合用户要求的软件产品

RUP和XP的区别：



RUP面向管理层面，XP面向实施层面。两者是相辅相成、互相补充的。

在RUP管理框架之下，才能在迭代实施过程中采用XP，发挥XP的“快速”作用

✧ 可行性研究

技术可行性、社会可行性、经济可行性

三种结论：可行、基本可行、不可行

✧ 软件需求分析

- 业务需求
- 用户需求
- 系统需求
- 功能需求

过程：需求获取—>分析建模—>文档编写—>需求验证—>需求变更

✧ UML

构成

视图：用例视图、设计视图、过程视图、实现视图、部署视图

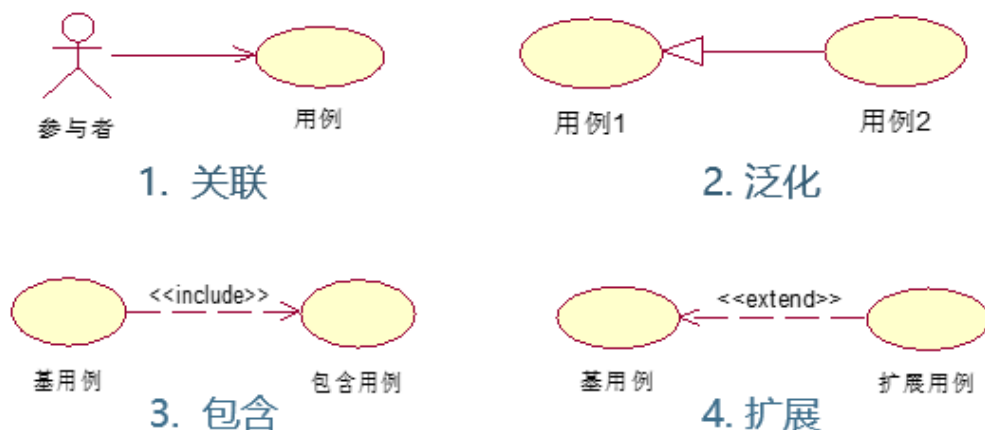
图：

- 用例图
- 静态图：类图、对象图、包图
- 行为图：状态图、活动图
- 交互图：顺序图、合作图
- 实现图：组件图、部署图

模型元素：基元素、构造型元素

✧ 用例图

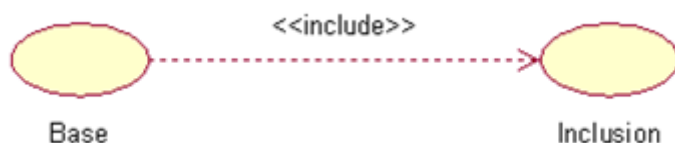
🌀 用例图中有以下几种关系：



用例之间的关系

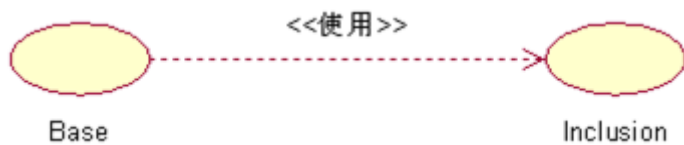
① 包含

包含关系指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。在UML中，包含关系是通过带箭头的虚线段加<>字样来表示，箭头由基础用例(Base)指向被包含用例(Inclusion)。



在处理包含关系时，具体的做法就是把几个用例的公共部分单独的抽象出来成为一个新的用例。主要有两种情况需要用到包含关系：

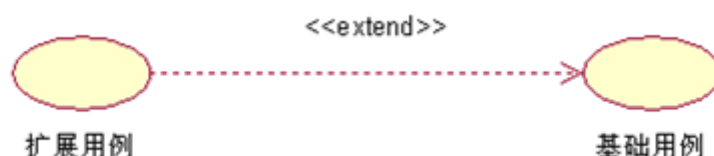
- 第一，多个用例用到同一段的行为，则可以把这段共同的行为单独抽象成为一个用例，然后让其他用例来包含这一用例。
- 第二，某一个用例的功能过多、事件流过于复杂时，我们也可以把某一段事件流抽象成为一个被包含的用例，以达到简化描述的目的。



1 扩展

在一定条件下，把新的行为加入到已有的用例中，获得的新用例叫做扩展用例(Extension)，原有的用例叫做基础用例(Base)，从扩展用例到基础用例的关系就是扩展关系。

一个基础用例可以拥有一个或者多个扩展用例，这些扩展用例可以一起使用。



3.泛化

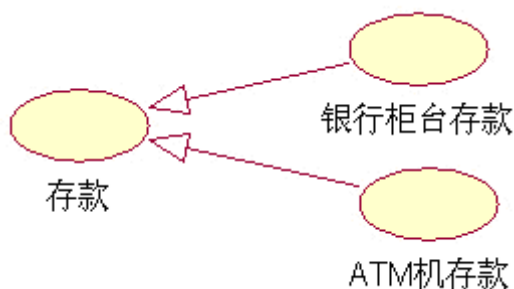
用例的泛化指的是一个父用例可以被特化形成多个子用例，而父用例和子用例之间的关系就是泛化关系。

在用例的泛化关系中，子用例继承了父用例所有的结构、行为和关系，子用例是父用例的一种特殊形式。

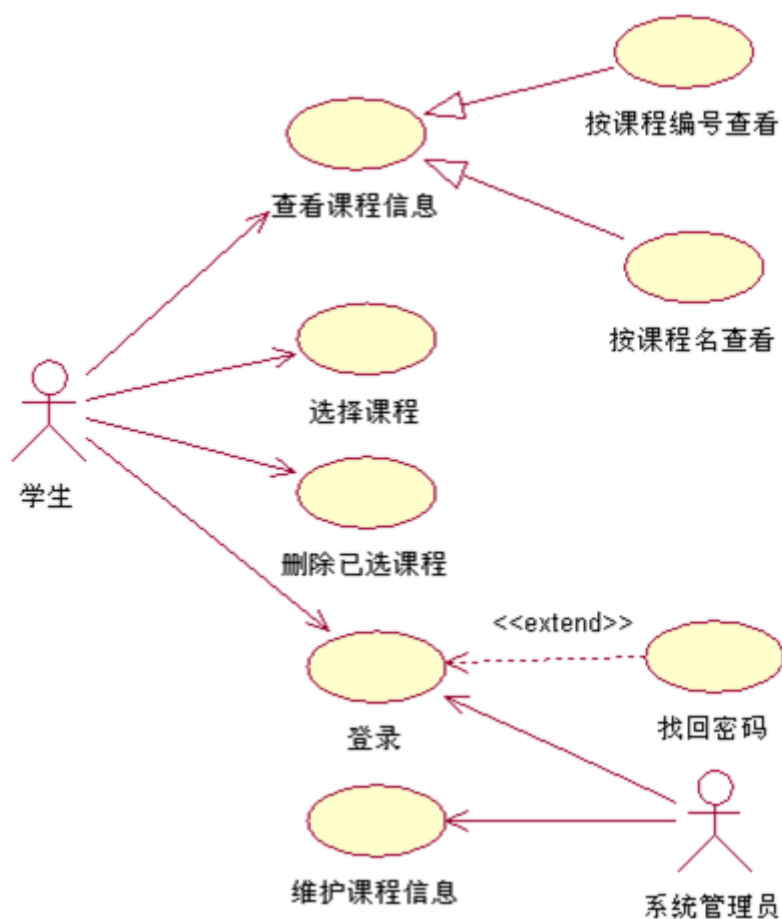
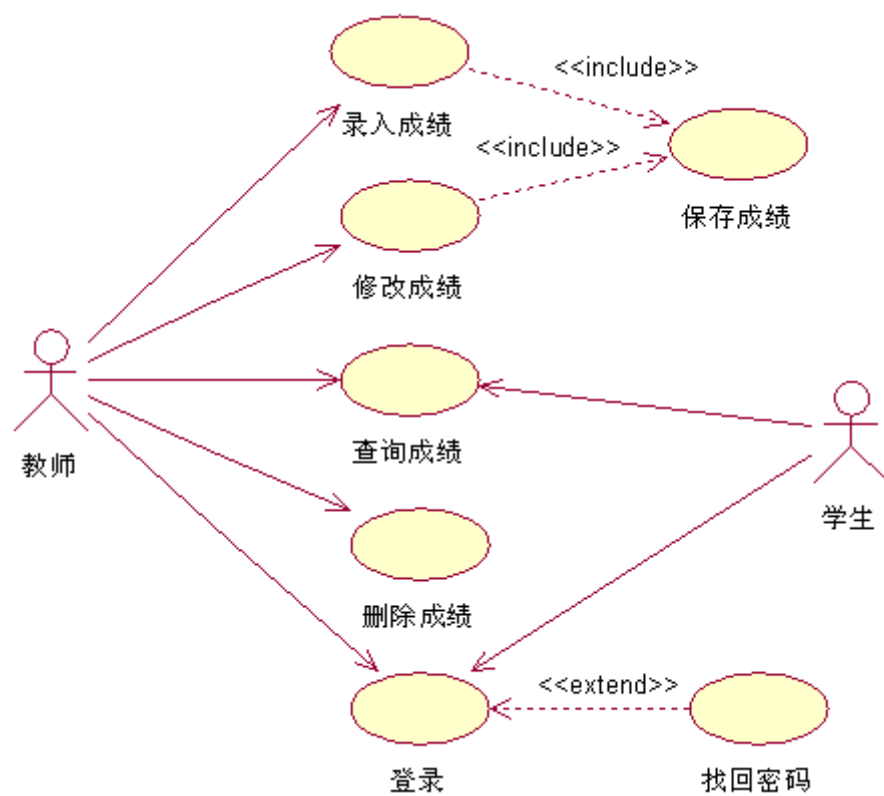
子用例还可以添加、覆盖、改变继承的行为。在UML中，用例的泛化关系通过一个三角箭头从子用例指向父用例来表示。

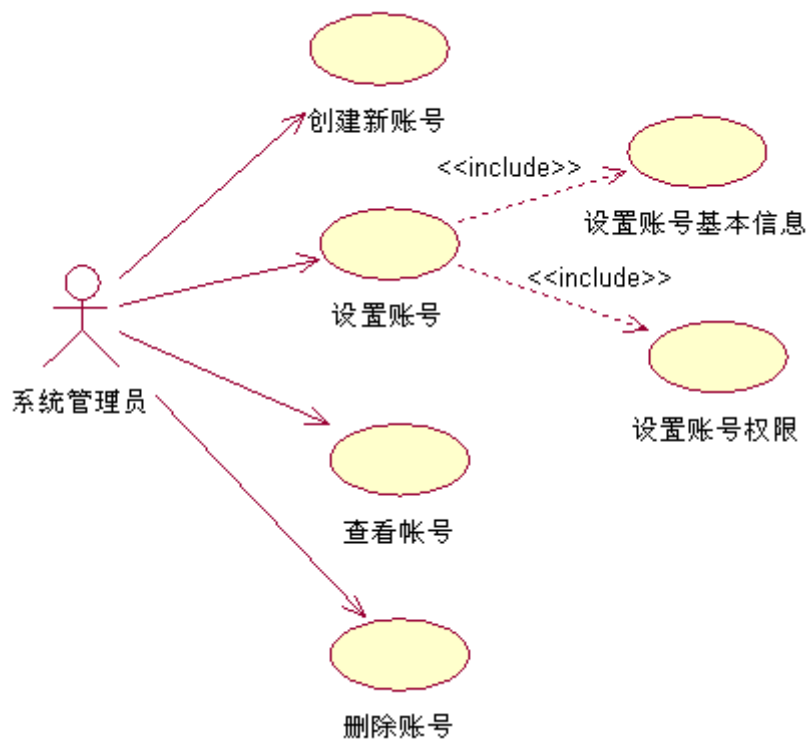


泛化的示例：银行存款有两种方式，一种是银行柜台存款，一种是ATM机存款。在这里，银行柜台存款和ATM机存款都是存款的一种特殊方式，因此“存款”为父用例，“银行柜台存款”和“ATM机存款”为子用例。



例子：





✧ 类图

类图以反映类的结构(属性、操作)以及类之间的关系为主要目的，描述了软件系统的结构，是一种静态建模方法

类图中的“类”与面向对象语言中的“类”的概念是对应的，是对现实世界中的事物的抽象

规范

- 1.单个单词的属性名小写。
- 2.如果属性名是由多个单词组成，那么将多个单词合并，除了第一个单词外。其它单词首字母大写。
- 3.属性的语法：可见性 名称:类型=默认值[约束特性]
 - 可见性表示该属性对类外的元素是否可见。常用的有公有(+)、受保护(＃)和私有(-)三种。
 - 名称表示属性的名称，是一个字符串。
 - 类型定义属性的种类（基本类型或自定义类型）

•默认值表示属性的初始值。

约束特性表示描述对属性的约束。

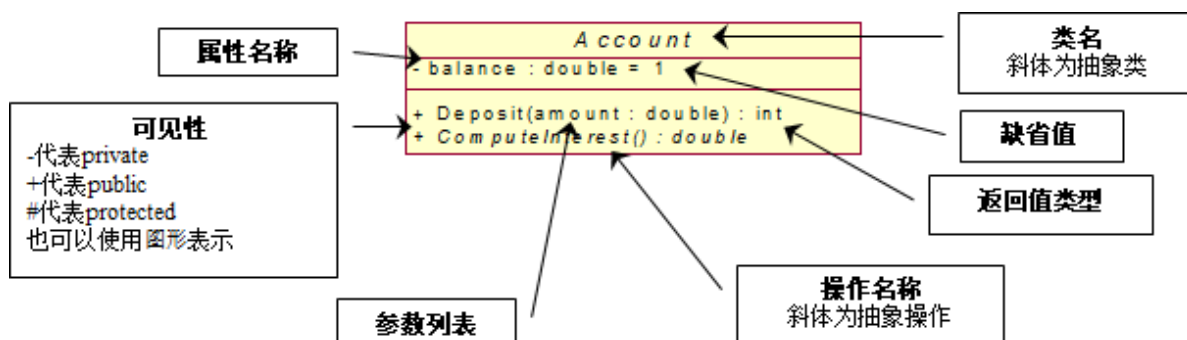
类图中事物

(一) 类

1 从上到下分为三部分，分别是类名、属性和操作。类名是必须有的

2.类如果有属性，则每一个属性必须有一个名字，另外还可以有其他的描述信息，如可见性、数据类型、缺省值等

3.类如果有操作，则每一个操作也都有一个名字，其他可选的信息包括可见性、参数的名、参数类型、参数缺省值和操作的返回值的类型等



(二) 接口

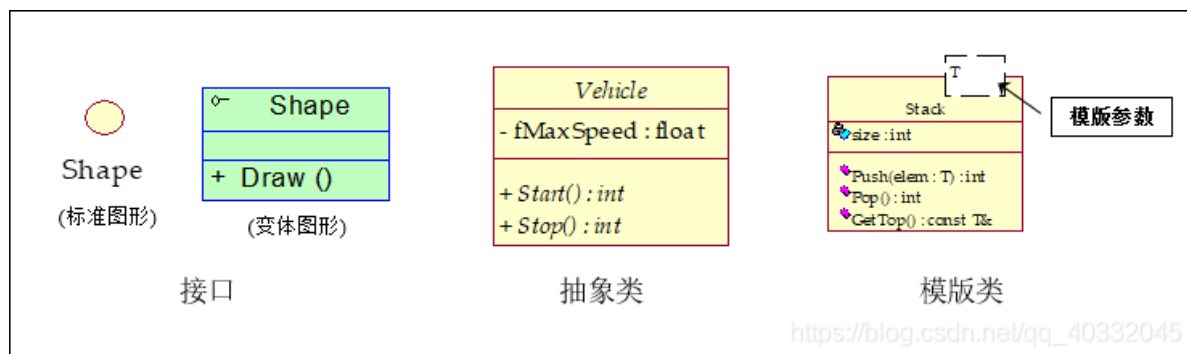
一组操作的集合，只有操作的声明而没有实现

(三) 抽象类

不能被实例化的类，一般至少包含一个抽象操作

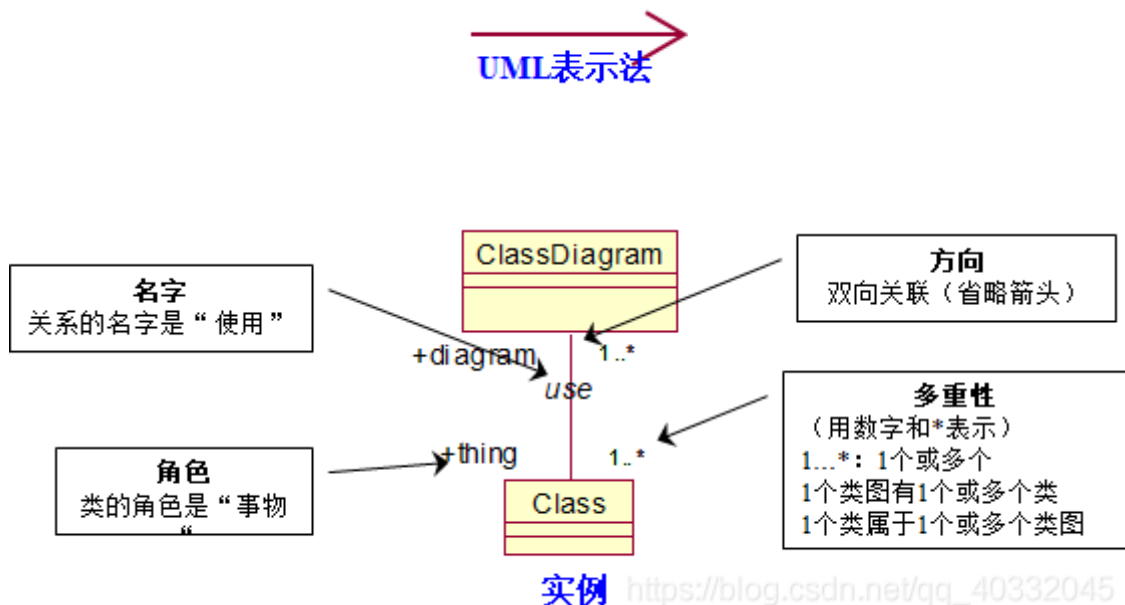
(四) 模板类

一种参数化的类，在编译时把模板参数绑定到不同的数据类型，从而产生不同的类



三、类图中的关系及解释

（一）关联关系——描述了类的结构之间的关系。具有方向、名字、角色和多重性等信息。一般的关联关系语义较弱。

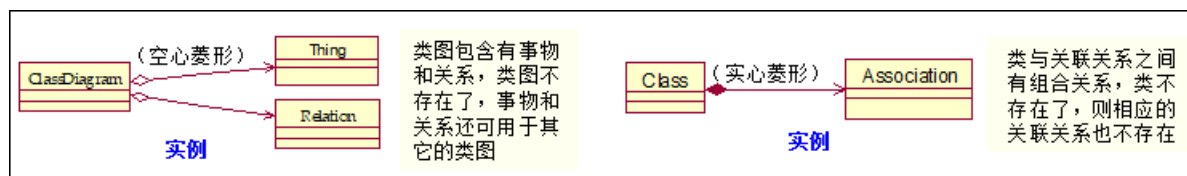


也有两种语义较强，分别是聚合与组合

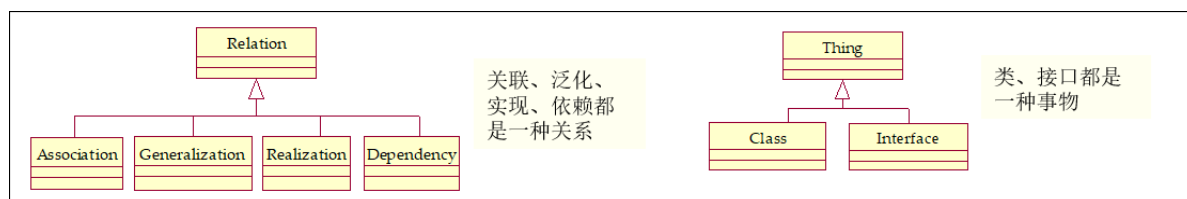
- ① 聚合关系——特殊关联关系，指明一个聚集（整体）和组成部分之间的关系



- 2.组合关系——语义更强的聚合，部分和整体具有相同的生命周期

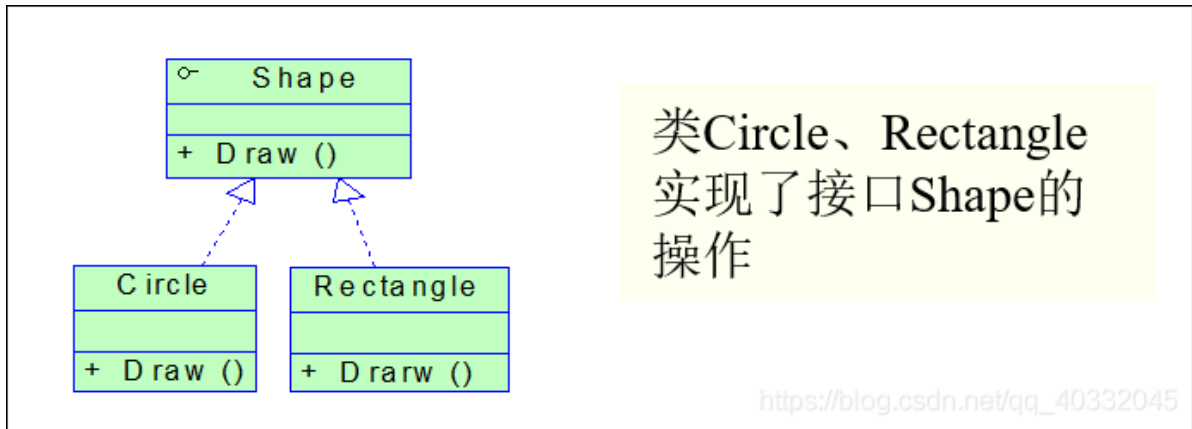


（二）泛化关系——※在面向对象中一般称为继承关系，存在于父类与子类、父接口与子接口之间



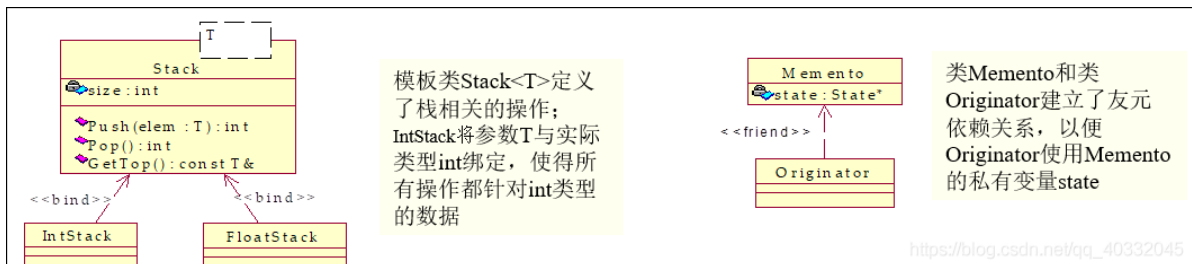
（三）实现关系——对应于类和接口之间的关系

UML表示法

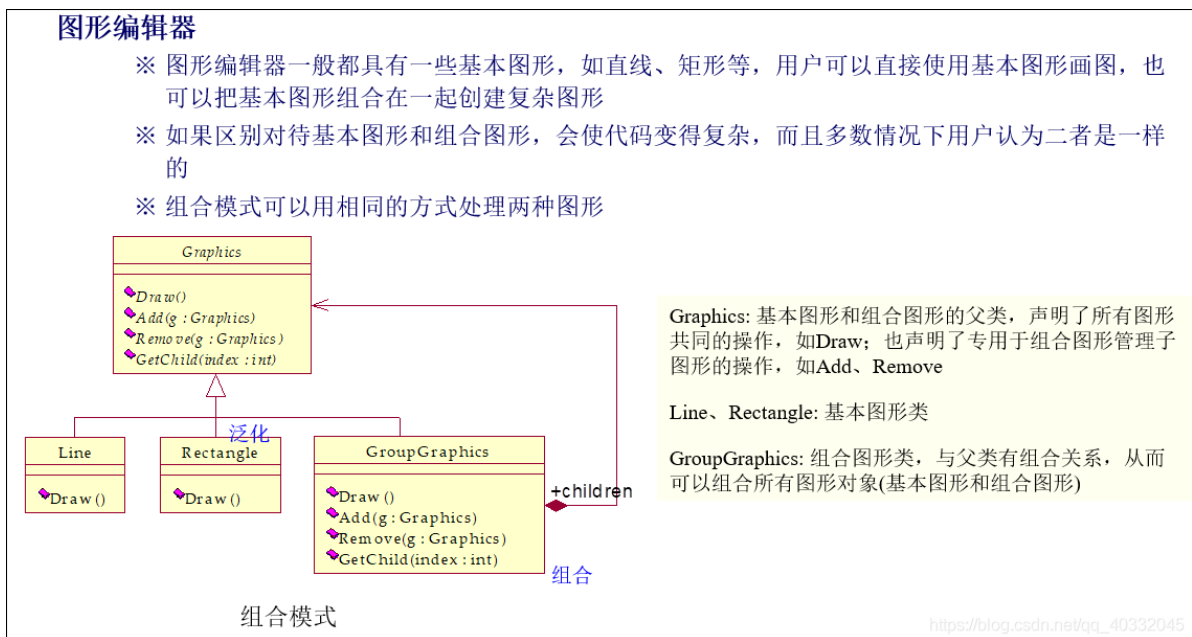


(四) 依赖关系——※描述了一个类的变化对依赖于它的类产生影响的情况。有多种表现形式，例如绑定(bind)、友元(friend)等

UML表示法

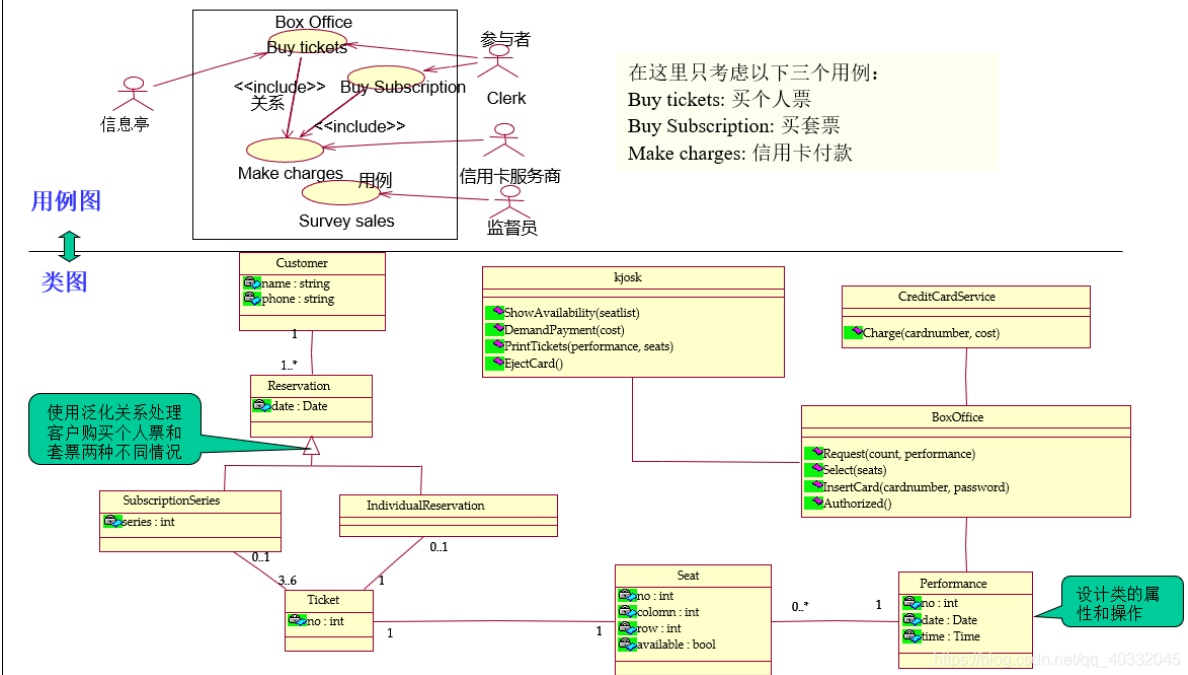


例子：



演出售票系统

在用例驱动的开发过程中，通过分析各个用例及参与者得到类图。分析用例图的过程中需要根据面向对象的原则设计类和关系，根据用例的细节设计类的属性和操作



状态图

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。

状态	上格放置名称，下格说明处于该状态时，系统或对象要做的工作(见可选活动表)	<div>Enter Password</div> <div>entry / set echo to star; password.reset()</div> <div>exit / set echo normal</div> <div>digit / handle character</div> <div>clear / password.reset()</div> <div>help / display help</div>
转移	转移上标出触发转移的事件表达式。如果转移上未标明事件，则表示在源状态的内部活动执行完毕后自动触发转移	消息(属性)[条件]/动作
开始	初始状态(一个)	
结束	终态(可以多个)	



图中包含以下状态

初始状态

Available状态

Locked状态

Sold状态

状态间的转移

初始状态→Available状态

票被预订(lock): Available→Locked

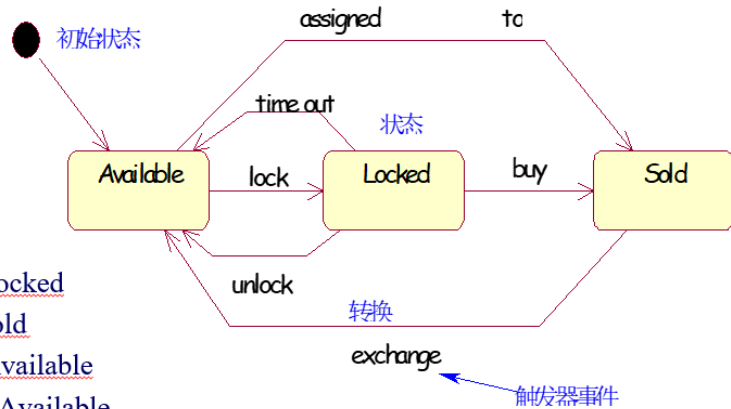
预定后付款(buy): Locked→Sold

预定解除(unlock): Locked→Available

预定过期(time out): Locked→Available

直接购买(assigned to): Available→Sold

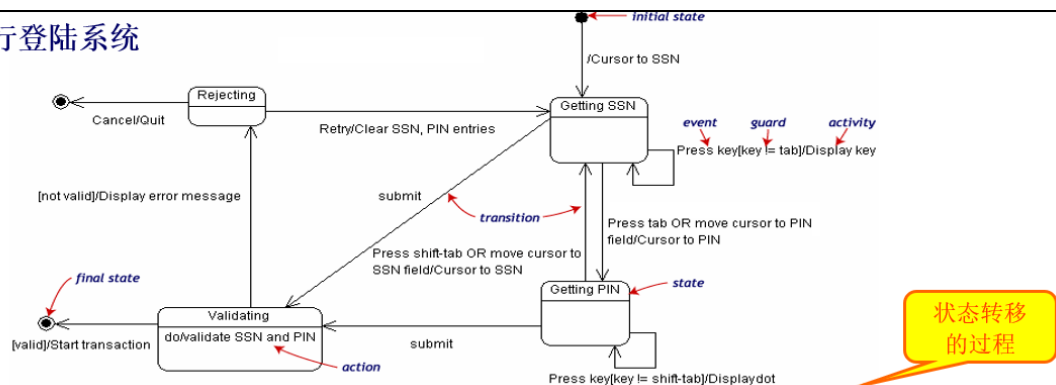
换其它票(exchang), 该票重有效: Sold→Available



https://blog.csdn.net/qq_40332045



网上银行登陆系统



登陆要求提交个人社会保险号(SSN)和密码(PIN)经验证有效后登陆成功。

登陆过程包括以下状态:

- ※ 初态(Initial state)
- ※ 获取社会保险号状态(Getting SSN)
- ※ 获取密码状态(Getting PIN)
- ※ 验证状态(Validating)
- ※ 拒绝状态(Rejecting)
- ※ 终态(Final state)

出发状态	动作	到达状态
Initial state	移动鼠标到 SSN	Getting SSN
Getting SSN	键入非tab键, 显示键入内容	Getting SSN
	键入tab键, 或移动鼠标到BIN	Getting PIN
Getting PIN	提交	Validating
	键入非shift-tab键, 显示 “ * ”	Getting PIN
	键入shift-tab键, 或移动鼠标到SSN	Getting SSN
Validating	提交	Validating
	验证提交信息有效, 状态转移	Final state
Rejecting	验证提交信息无效, 显示错误信息	Rejecting
	退出	Final state
Rejecting	重试, 清除无效的SSN, PIN	Getting SSN

有两个不同的终态



顺序图

顺序图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事件。

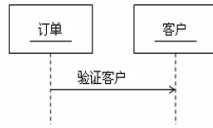
简单的例子

消息格式:

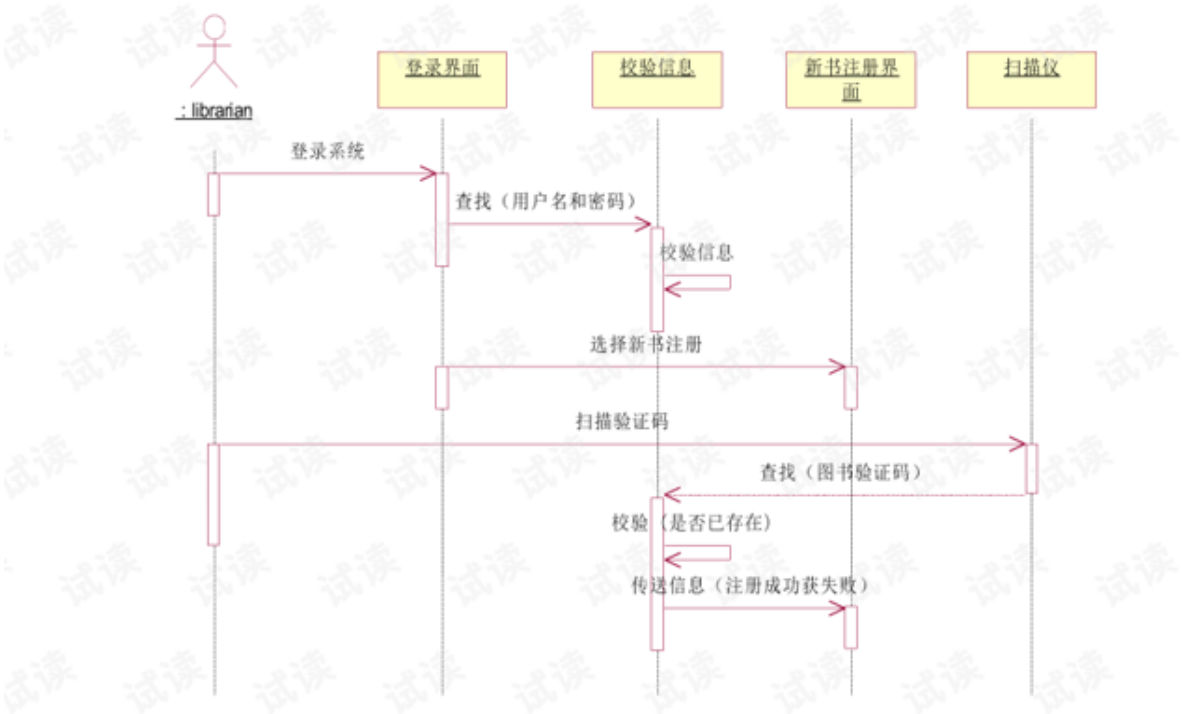
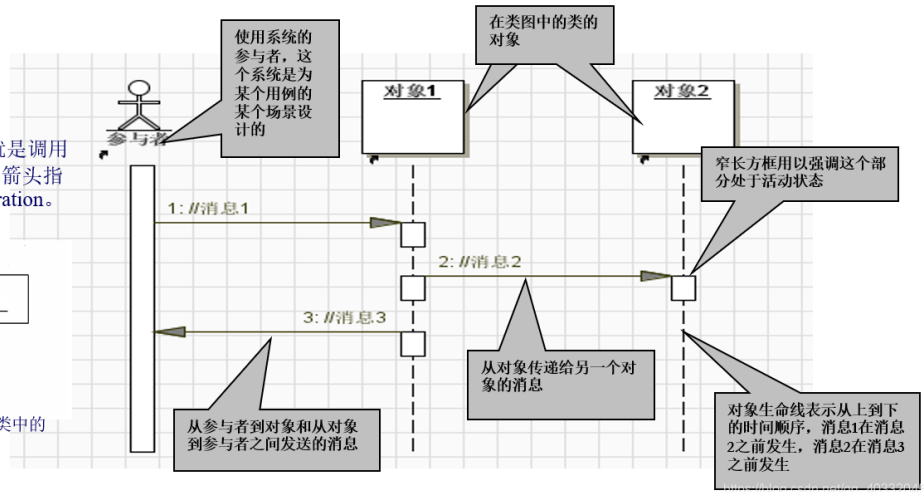
operation (parameter list)

向哪个对象发消息实际上就是调用它的类中的操作，就是调用箭头指向的对象所在类的一个operation。

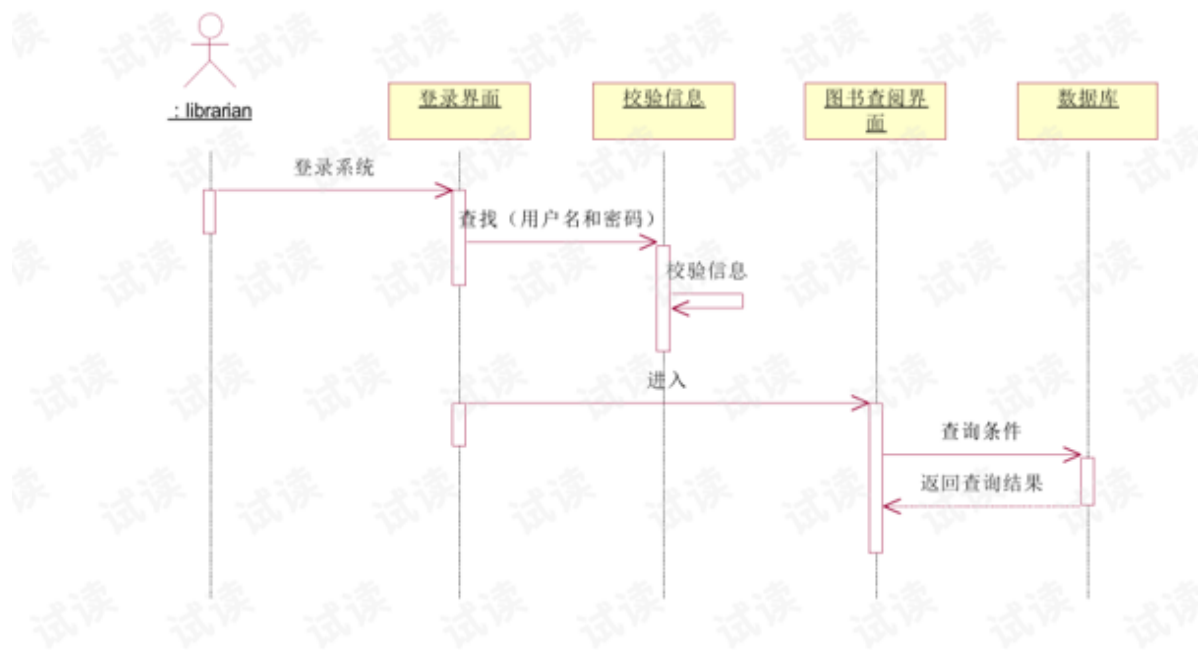
例:



订单类发消息给客户类调用客户类中的“验证客户”操作



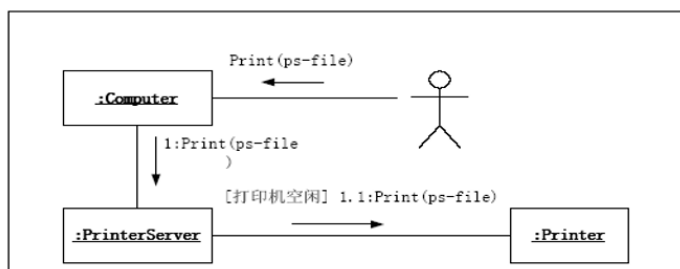
● 图书查询



* 协作图

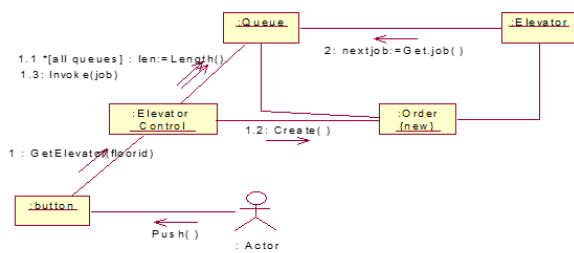
协作图是一种交互图，强调的是发送和接收消息的对象之间的组织结构，使用协作图来说明系统的动态情况。

1. 打印操作的协作图



actor发送Print消息给Computer，
Computer发送Print消息给
PrinterServer，如果打印机空闲，
PrinterServer发送Print消息给printer

2. 乘坐电梯的协作图



图中存在的事物有：

参与者
按钮对象
电梯控制对象
命令对象
工作队列
电梯对象

图中存在的关系有：
链接

参与者需要乘坐电梯，他从系统外部按下按钮，让电梯到达他想去的楼层。此时，电梯系统的操作被启动，电梯控制对象以循环的方式检查所有的电梯，从中选择一个工作队列长度最短的。然后，它创建一个作业命令，并将该命令放入对应电梯的工作队列，接着激活队列。电梯对象并发运行，从它的队列中选择一个作业并执行。电梯是一个活动对象，它与它的控制线程并发执行。

https://blog.csdn.net/aq_40332045

* 活动图

活动 (ActionState)	动作的执行	
起点 (InitialState)	活动图的开始	
终点(FinalState)	活动图的终点	
对象流(ObjectFlowState)	活动之间的交换的信息	
发送信号(signalSending)	活动过程中发送事件，触发另一活动流程	
接收信号(SignalReceipt)	活动过程中接收事件，接收到信号的活动流程开始执行	
泳道(SwimLane)	活动的负责者（模型中存在多个对象时候使用比较适合分为水平和垂直）	

迁移(transition)	活动的完成与新活动的开始	
分支(junction point)	根据条件，控制执行方向	
分叉(fork)	以下的活动可并发执行	
结合(join)	以上的并发活动再此结合	

例子：



一般的活动图

本活动图描述一个处理订单的用例执行过

(1)执行setup order

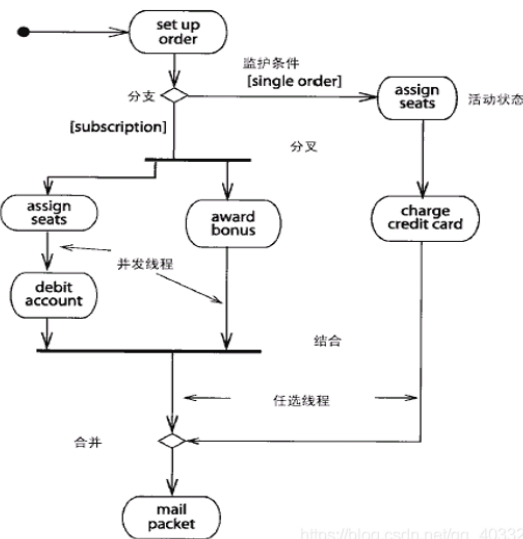
(2)根据order的类型是执行不同的分支：

single order: 执行assign seat、charge credit card

subscription: 同时执行assignseats、debit account或
award bonus

single order与subscription两步可同时进行

(3) 最后mail packet。



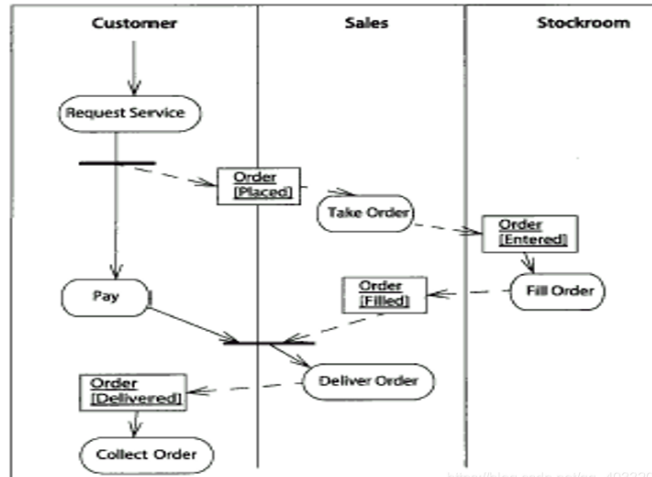


带泳道的活动图

本例为一个按活动职责(带泳道)组织的处理订单用例的活动图(模型中的活动按职责组织)。活动被按职责分配到用线分开的不同区域(泳道):

Customer
Sales
Stockroom

- (1) 顾客要求服务, Sales负责接收订单, 并提交到Stockroom
- (2) Stockroom处理订单, 与此同时, Customer付款, 并由Sales处Deliver order至Customer。



面向对象软件工程方法

- 面向对象分析 (OOA)
- 面向对象设计 (OOD)
- 面向对象编程 (OOP)
- 面向对象测试 (OOT)

用面向对象方法开发软件, 通常需要建立3种模型:

- 描述系统功能的 功能模型
- 描述系统数据结构的 对象模型
- 描述系统控制结构的 动态模型

软件设计

软件设计包括两大部分, 一个是结构设计, 另一个是详细设计。

- 系统结构设计(architectural design)也叫总体设计或概要设计。(包括数据/类设计、软件体系结构设计、接口设计)

- 系统的过程设计(procedural design)也叫详细设计（包括部件级设计）

概要设计（总体设计、结构设计）：

包括系统的总体设计文档、各模块的概要设计文档。在需求规格说明书的基础上描述系统的架构、功能模块的划分、模块接口的定义、用户界面设计、数据库设计等。

详细设计（过程设计）：

根据概要设计进行的模块划分，实现各模块的算法设计，实现用户界面设计，表单，需要的数据，数据结构设计的细化等。

面向对象设计

主要特点是：容易维护

数据持久化

两种方式：

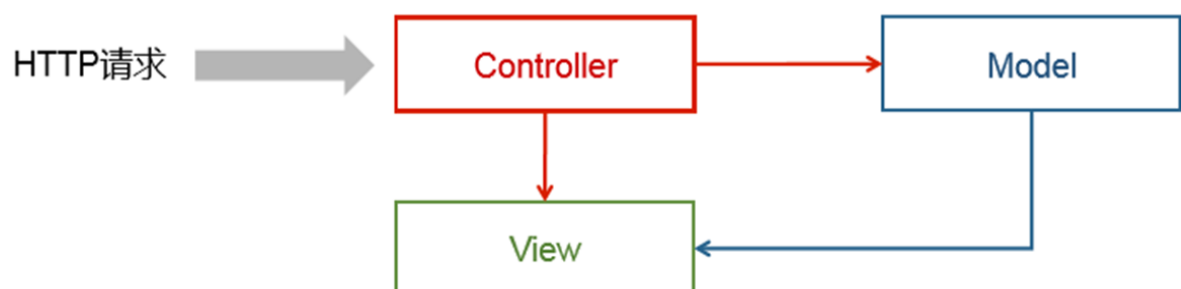
- 数据库管理系统（DBMS）
- 文件存储模式

MVC结构

模型（Model）对象

视图（View）对象

控制器（Control）对象

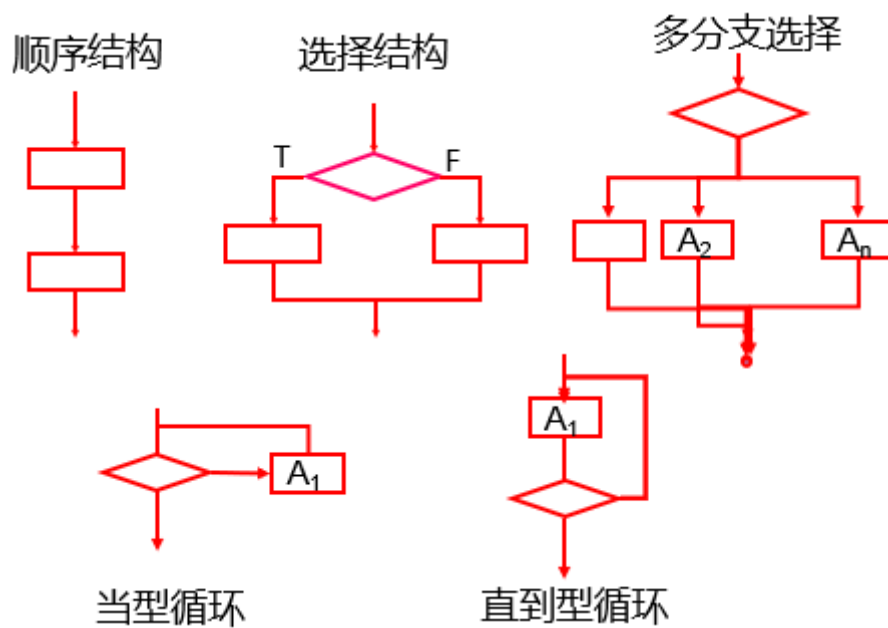


✧ 面向对象的详细设计

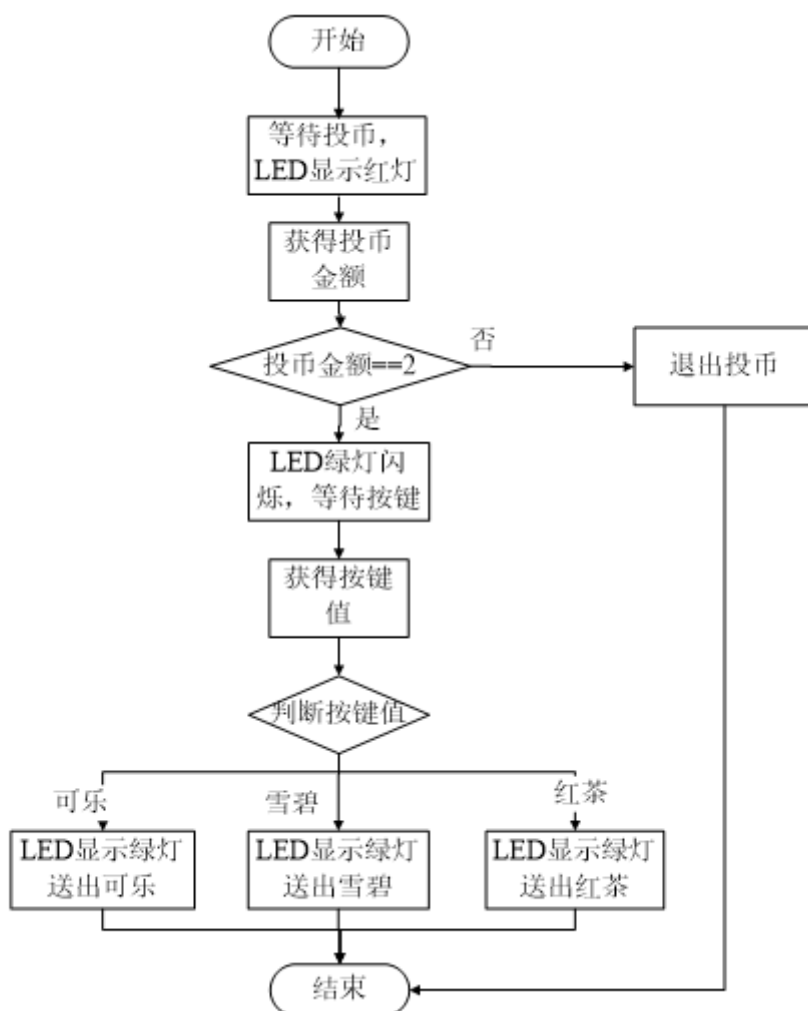
主要是对构件中的每个类进行详细描述，包括：

- 属性的数据结构设计
- 方法的设计
- 实现接口所需机制的设计

程序流程图

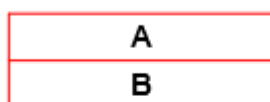


例子：

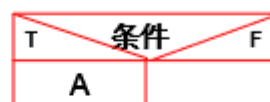
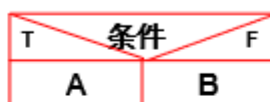


N-S图(看懂)

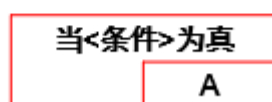
顺序结构



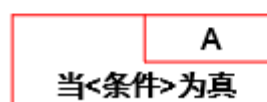
选择结构



While型循环结构



Do-While型循环结构



判定表

	1	2	3	4	5	6	7	8	9
国内乘客		T	T	T	T	F	F	F	F
头等仓		T	F	T	F	T	F	T	F
残疾乘客		F	F	T	T	F	F	T	T
行李重量 $W \leq 30$	T	F	F	F	F	F	F	F	F
免费	×								
$(W-30) \times 2$				×					
$(W-30) \times 3$					×				
$(W-30) \times 4$		×						×	
$(W-30) \times 6$			×						×
$(W-30) \times 8$						×			
$(W-30) \times 12$							×		

判定树



* 注释

分为：

○ 序言性注释（程序开头部分），包括：

- ① 程序标题；
- ② 有关本模块功能和目的的说明；
- ③ 主要算法；
- ④ 接口说明：包括调用形式，参数描述，子程序清单；

- ⑤ 有关数据描述：重要的变量及其用途，约束或限制条件，以及其它有关信息；
 - ⑥ 模块位置：在哪个源文件中，或隶属于哪一个软件包；
 - ⑦ 开发简历：模块设计者，复审者，复审日期
- 功能性注释（程序体中）

解释程序或者语句的作用

✧ 数据说明

数据说明的次序应该标准化。有次序易查阅，能加速测试、调试和维护的过程。

→ 例如：数据说明

- ① 常量说明
- ② 简单变量类型说明
- ③ 数组说明
- ④ 公用数据块说明
- ⑤ 所有的文件说明

数据类型说明

- ① 整型量说明
- ② 实型量说明
- ③ 字符型量说明
- ④ 逻辑量说明

当多个变量名在一个语句中说明时，应该按字母顺序排列这些变量。

✧ 软件测试

软件测试贯穿软件开发的整个生命周期

测试只能找出程序中的错误，但在未发现错误时，并不能证明程序中没有错误。

发现错误并不是软件测试的最终目标。测试阶段的根本目标是尽可能多的发现软件中潜藏的错误，最终把一个高质量的软件系统交给用户使用。

软件测试的目的是发现软件中的错误和缺陷，并加以纠正，设计合适的测试用例，用尽可能少的测试用例，来发现尽可能多的软件错误

分类：

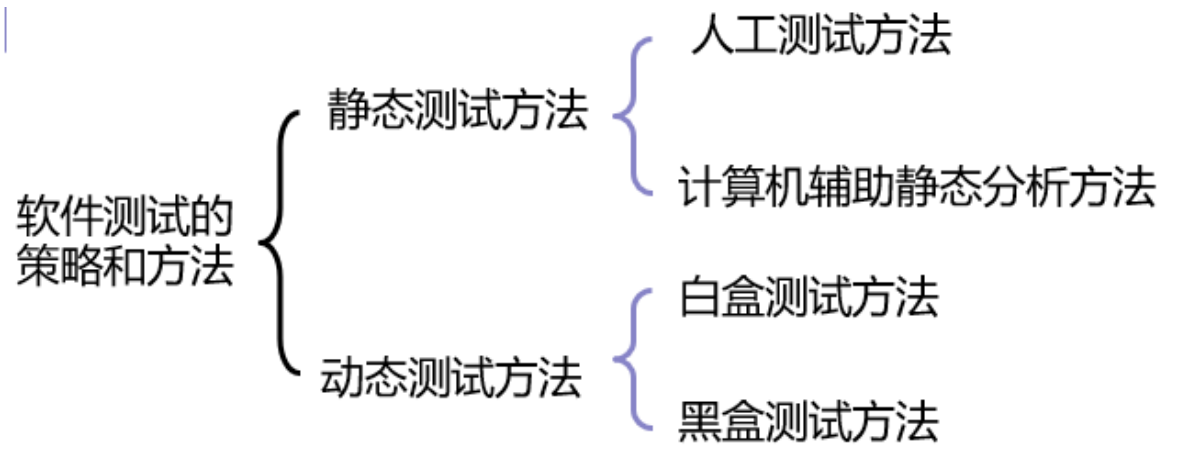
- 单元测试（程序模块进行正确性检验的测试工作）（单元）

- 集成测试（有增量集成法和非增量集成法之分）（局部）
- 系统测试（整体）
- 验收测试（必须有用户积极参与，或以用户为主测试）（用户参与）

测试阶段	主要依据	测试人员 测试方式	主要测试内容
单元测试	详细设计文档	由程序员执行黑盒+白盒测试	接口测试、边界条件测试、局部数据结构与算法测试、路径测试、错误处理路径的测试
集成测试	概要设计文档和软件需求	由程序员执行黑盒测试	接口测试、路径测试、功能测试、性能测试
系统测试	软件需求	由独立测试小组执行黑盒测试	功能测试、健壮性测试、性能测试、UI测试、安全性测试、压力测试、可靠性测试、安装/反安装测试等
验收测试	软件需求	由用户执行黑盒测试	

i 分析软件产品的过程称为静态测试，运行软件的测试过程称为动态测试。

验证保证产品的正确性，确认保证生产了正确的产品。



静态测试

通过人工分析或程序正确性证明的方式来确认程序正确性。

桌前检查，代码会审，步行检查

动态测试

通过输入数据，运行软件，观察执行状态和结果，来检验软件的动态行为和运行结果的正确性

白盒测试（已知产品内部工作过程）：

发现错误的 能力	标 准	含 义
1(弱)	语句覆盖	每条语句至少执行一次
2	判定覆盖	每一判定的每个分支至少执行一次
3	条件覆盖	每一判定中的每个条件，分别按“真”、“假”至少各执行一次
4	判定/条件覆盖	同时满足判定覆盖和条件覆盖的要求
5 (强)	条件组合覆盖	求出判定中所有条件的各种可能组合值，每一可能的条件组合至少执行一次

黑盒测试（已知产品功能）

等价分类法：不用考虑程序内部结构

边界值分析法（选取正好等于、刚刚大于或小于边界值作为测试数据）

错误推测法：凭经验进行的

因果图法（描述对于多种输入条件的组合，相应产生多个动作的形式来设计测试用例，最终生成的是判定表）

集成测试

非渐增性测试：

采用一步到位的方法来集成系统，在对每个模块进行单元测试的基础上，把所有模块按设计要求组装在一起

渐增性测试：

自顶向下结合：不需要编写驱动模块，只需要编写桩模块

自底向上结合：只需要编写驱动模块

α测试和β测试

- **Alpha 测试**：由开发者模拟用户对即将面市的软件产品进行测试。开发者需要记录发现的错误和使用中遇到的问题，该测试是在受控的环境中进行的。
- **Beta 测试**：由软件的最终用户在一个或多个客户场所进行。**Beta**测试是软件在开发者不能控制的环境中的“真实”应用。用户记录在**Beta**测试过程中遇到的一切问题（真实的或想像的），并且定期把这些问题报告给开发者。接收到在**Beta**测试期间报告的问题之后，开发者对软件产品进行必要的修改，并准备向全体客户发布最终的软件产品


回归测试

- 回归测试就是用于保证由于调试或其他原因引起的变化，不会导致非预期的软件行为或额外错误的测试活动。
- 回归测试是指重新执行已经做过的测试的某个子集，以保证修改变化没有带来非预期的副作用。

面向对象测试

策略：

- **类内测试**：相当于单元测试，包括类内的方法测试和类的行为测试。
- **类间测试**：相当于集成测试，要根据类之间的关系进行组装，有独立类和依赖类等。
- **基于场景的测试**：相当于确认测试和系统测试。

 白盒测试和黑盒测试方法也适用于面向对象测试

✧ 软件调试

调试是在测试发现错误之后排除错误的过程

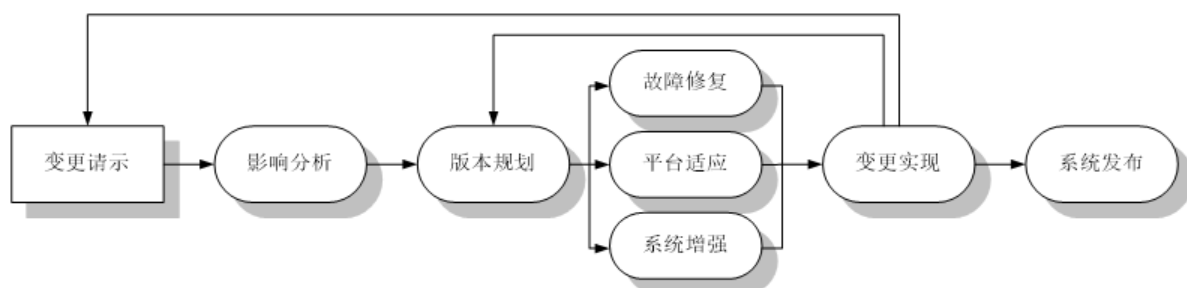
方法：

- 蛮干法 --- 逐点（单步）跟踪
- 回溯法 --- 从出错处沿控制流向上追溯
- 原因排除法 --- 对分查找法、归纳法和演绎法

✧ 软件维护

i 文档是影响软件可维护性的决定因素

维护过程：



类型：

- 改正性维护(为了识别和纠正软件错误、改正软件性能上的缺陷、排除实施中的误使用)
- 适应性维护(为使软件适应这种变化，而去修改软件)
- 扩充与完整性维护(用户提出新的功能与性能要求)
- 预防性维护（为了提高软件的可维护性、可靠性等，为以后进一步改进软件打下良好基础）（又称软件再工程）

非结构性维护

- 软件的配置中只有源代码。

- 由于没有分析和设计文档，无法对程序的功能进行反向追踪，理解别人的代码是很痛苦的事情。
- 由于配置中没有测试文档，所以维护后的代码无法进行回归测试。因而导致程序的结构化被不断的破坏，维护的质量无法得到保证。

结构性维护

- 待维护的软件配置是完整的。
- 用户提出的维护申请用正向追踪很容易从分析设计文档追踪直至代码中，从而使维护人员很容易定位代码的维护点。所以这种维护不会破坏软件的结构。
- 结构化维护不仅能减少维护的工作量，还能提高维护的质量。

软件系统的文档类型

- 用户文档：主要描述系统功能和使用方法，并不关心这些功能是怎样实现的。
- 系统文档：描述系统设计、实现和测试等各方面的内容

软件再工程

预防性维护又称为软件再工程。指的是重新构造或编写现有系统的一部分或全部，但不改变其功能

目的：

- 努力使系统更易于维护，系统需要被再构造和再文档化。
- 减少风险：重新开发一个在用的系统具有很高的风险，可能会有开发问题、人员问题和规格说明问题。
- 低成本：再工程的成本比重新开发软件的成本要小得多

重构

在不改变软件现有功能的基础上，通过调整程序代码改善软件的质量、性能，使其程序的设计模式和架构更趋合理，提高软件的扩展性和维护性

重构要注意两点：

① 保持系统的核心价值不变

② 注意风险