软工复习整理（修订补充）——YaHei

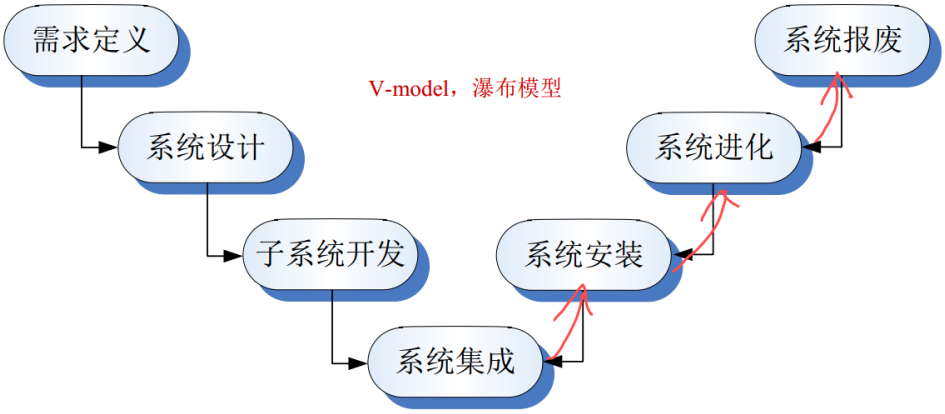
概论：

1. 软件费用
   a) 软件费用经常是系统费用的主要成分
   b) 维护费用超过开发费用
   c) 软件工程对费用敏感
2. **软件 = 程序 + 文档**
3. 软件工程：关于**软件生产**各个方面的一门工程学科
4. 计算机科学：研究构成计算机和软件系统基础的有关**理论和方法**
   软件工程：研究软件制作中的**实际问题**
5. 系统工程：系统开发的方方面面，包括硬件、软件和工艺等；
   软件工程：系统工程的一部分
6. *软件过程的三要素：方法、工具、过程
7. **软件过程：描述、开发、有效性验证、进化**
8. 软件工程方法：结构化方法（不断细分），目的在于提高软件的生产性价比
9. CASE：计算机辅助软件工程
10. 优良软件：可维护性（进化）、可依赖性（可信赖）、有效性（不浪费资源）、有用性（对用户来说）

系统工程

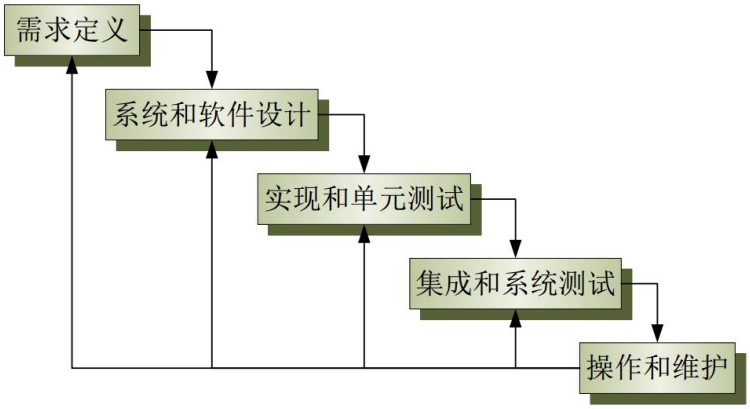
1. 系统工程：设计、实现、配置、操作，包括硬件、软件、人
2. 系统：一组相关联、能一起工作从而达到某个目标的相关组件的集合
3. 软件工程是系统工程的一部分，而且比重在不断增长
4. 几个特性
   （**RAMS，Reliability 可靠、Availability 可用、Maintainability 可维护、Safety 安全**）
   a) 可靠性：从开始使用到第一次出现故障的时间
      i. 影响因素：硬件失效、软件失效、操作员误操作
      ii. 组件失效会传播到整个系统
      iii. 硬件失效产生错误信号以致超出软件预期
      iv. 软件错误报警引起操作员紧张进而导致错误操作
   b) 可用性：从开始使用到第一次出现严重故障以至于无法继续使用的时间
   c) 可维护性：修复系统故障所需时间
5. 系统组件：传感器、执行机构、计算组件、调度组件、通信组件、接口组件

6. 系统工程过程通常采用瀑布模型



软件过程：

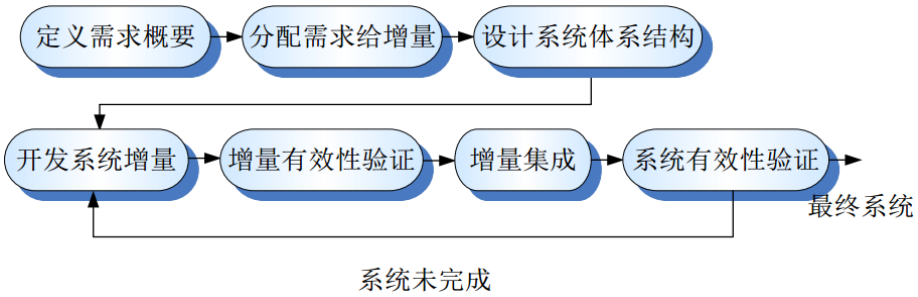
1. 瀑布模型：按部就班
   a) 过程



   b) 缺点：（需求、设计、测试）变更困难
   c) 适合：需求非常明确的情况
2. 进化式开发：描述与开发交叉
   a) 探索式：从比较**明确**的需求开始
      抛弃式：从比较**模糊**的需求开始（建立原型，适合用户界面）
   b) 缺点：过程可见性差、系统结构差
   c) 适合：交互式、生命周期短的系统或组件
3. 形式化系统开发（B 语言、Z 语言）：数学系统转化
   a) 确保正确性，可靠性高
   b) IBM 的净室软件开发（增量式开发、静态检验、统计性测试）
   c) 缺点：技术要求高、某些方面难以形式化描述
   d) 适合：严格、安全性和保密性要求极高的系统
4. 面向复用的开发：已有组件来组建

5. 混合模型

a) 增量式开发



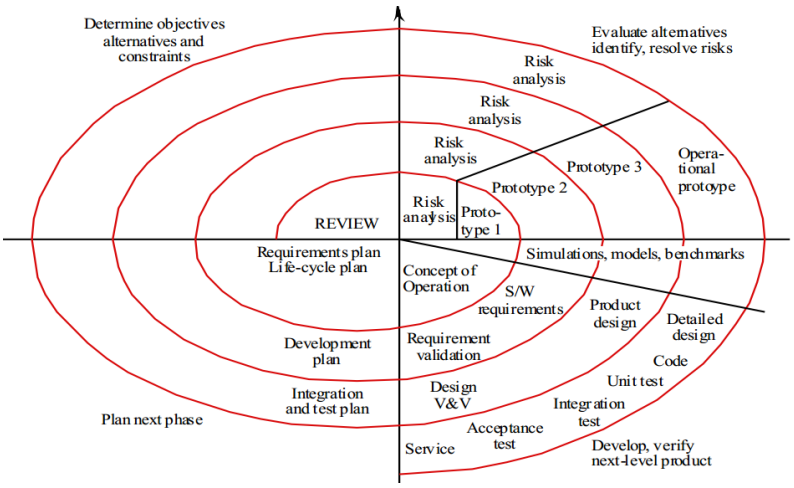
i. 功能划分为多个增量，逐一完成，需求变更只能应用与下一增量

ii. 总体进化式，局部瀑布式：
为每份增量生成需求和描述

iii. 适用：前期宣传、引导需求、总体风险低、测试分布合理（关键服务 – 优先级高 – 早开发 – 多测试）

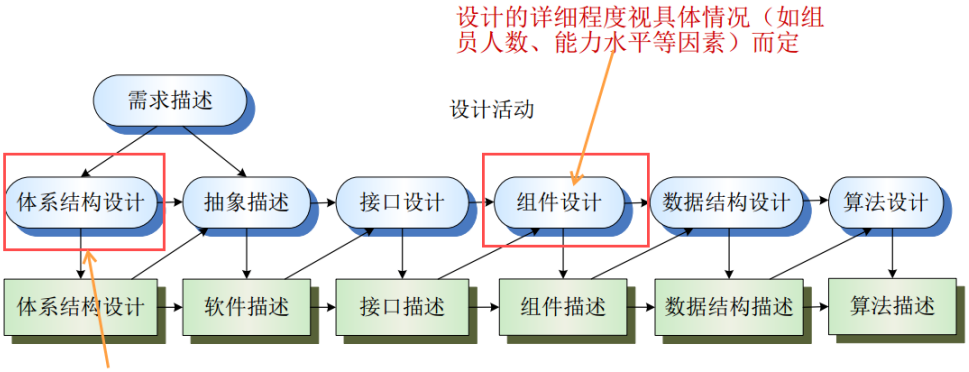
iv. 极限编程：增量很小的增量式开发，要求用户与开发团队联系紧密，通常采用结对编程——一般两人一组共用一个工作台，两人轮流编写代码、思考或检查代码，并且两人不是固定的而是两两轮流交替；

b) *螺旋式开发



6. 软件设计过程

需求描述、体系结构、抽象描述、接口、组件、数据结构、算法



图：体系结构图（组件数量7-20个为宜）设计产品
表：对图中组件的描述
文字：补充描述

7. 调试与测试

	调试	测试
目的	立，使程序能够工作	破，找到让程序不能工作的缺陷
方向	前向	后向
主要工作	缺陷定位与修复	证明存在缺陷
工作基础	程序员对程序的理解	需求
工作人员	程序员	测试人员（有时程序员也会参与）

8. 检验（Verify）和有效性验证（Validate）

a) 检验强调过程，验证强调结果

b) 主要目的

i. 发现系统中的缺陷

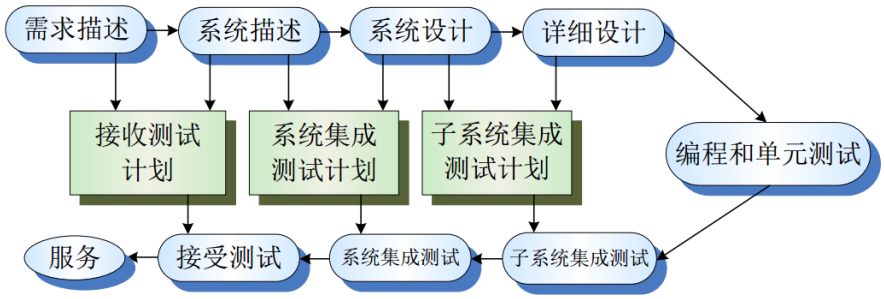
ii. 评估系统在实际操作中是否可用

c) 检查（审查）与测试：V&V 的手段

	检查、审查	测试
动静态	静态	动态
对象	代码、文档	实际运行的程序
实施时间	整个生命周期	原型或最终产品
优点	除了缺陷之外还能发现代码、文档是否规范或符合标准； 一次可以发现许多不同的缺陷	
缺点	复用领域和编程知识，要求评审人有较高的水平和丰富的经验； 无法检验非功能特征	一个缺陷可能屏蔽另一个缺陷
两者相互补充		

d) 检查过程
   i. 计划
   ii. 总体观察
   iii. 将代码和文档分发给检查团队
   iv. 检查并发现错误
   v. 修改错误
   vi. 继续检查（可能需要重新检查）
e) 测试
   i. 测试用例：测试数据和**期望的**输出
   ii. 测试高层代码：造一个空壳的底层代码，跟踪程序流
   测试底层代码：从高层调用，检查输入输出

9. V 模型（瀑布模型）



10. 能力成熟度模型 CMM
它是对于软件组织在定义、实施、度量、控制和改善其软件过程的实践中各个发展阶段的描述。共分为五级——
CMM1：初始级，缺乏管理，过程混乱无序且不可重复；
CMM2：可重复级，基本的管理制度，过程可重复；
CMM3：已定义级，过程文档化、标准化，采用评审保证质量；
CMM4：已管理级，对过程和质量有定量的理解和控制，采集指标、统计分析、改进；
CMM5：优化级，基于统计质量和过程控制工具，持续改进；

软件需求

1. 需求
   a) 只要是用户规定的，就是需求
   b) 功能
      i. 可能是合同标书的基础，必须公开解释
      ii. 可能是合同本身的基础，必须详细定义
      合同必须全是自然语言，不能使用图表（形象但不严谨）
   c) 功能描述有可能既是需求又是设计
2. 需求类型
   a) 用户需求
      系统的**概要**需求（服务和约束），约束的自然语言、表格和方块图
   b) 系统需求
      系统的**详细**需求（服务和约束），结构化的文档；
      作为系统设计的基础，可以作为合同的一部分，可以用系统模型

c) 软件设计描述

详细的软件描述，作为设计或实现的基础

3. 功能需求和非功能需求
   a) 功能需求：系统需要提供的服务的**描述**
   b) 非功能需求：系统提供的服务或功能上的**约束**（产品需求、机构需求、外部需求）
      i. 非功能**目标**：**模糊**的描述（如易用）

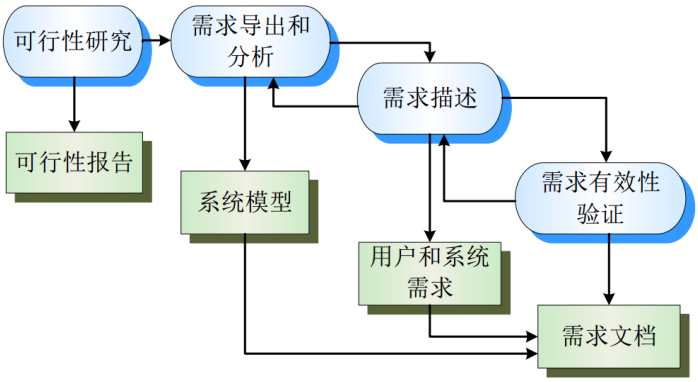
         非功能**需求**：**具体可检验**的描述（如经过 2 小时培训由经验的用户平均错误量少于每天 2 个）
      ii. 复杂系统中不同的非功能需求经常会冲突（如性能和功耗）
4. 实际不可能产生完全完整（包含所有服务）、一致（服务描述上没有冲突）的需求文档
5. 用户需求的结构化表述

> **2.6 栅格工具**
> **2.6.1** 编辑器应该提供栅格工具，包含一个由水平和垂直线阵列作为编辑器窗口的背景。这个栅格是被动的，即实体的对准是用户的任务。栅格是被动的，即实体的对准是用户的任务。
> 基本原理：一个栅格帮助用户产生一个具有间隔的整齐的图表。虽然一个主动栅格能自动将实体与某栅格线对齐，但这种定位也是不精确的。用户是决定实体应该在哪里放置的最好的决定者
> *Specification*: ECLIPSE/WS/Tools/DE/FS Section 5.6

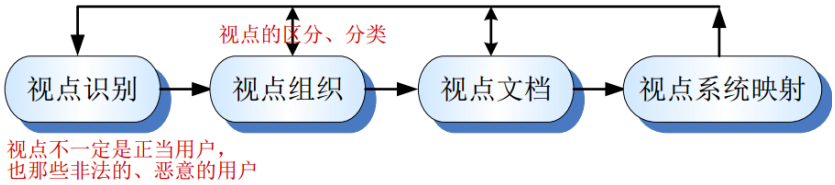
编号方便查找、需求跟踪、讨论时的称谓；
需求删除后编号一并删除，新增的需求编号不能占用已删除的编号（防止其他需求对已删除的需求的引用导致需求混乱）

6. 需求描述
   a) 自然语言的缺点
      i. **二义性**：读者和作者对同一词语可能又不同解释
      ii. **随意性太大**
      iii. **模块化不够**
   b) 结构化的语言（格式模板限制、排除二义性和随意性）
      i. 名称（唯一、确定）
      ii. 描述（介绍、清晰易懂）
      iii. 输入及来源
      iv. 输出及目的地
      v. 前置条件、后置条件
      vi. 决定者（负责人）
      vii. 变更记录
      viii. ……
   c) 基于 PDL 的需求定义
      i. 类似程序设计的伪代码
      ii. 适合：操作可分解为一系列顺序非常关键的动作、定义软硬件接口
      iii. 缺点：难懂、非程序员不易理解
   d) 接口描述
      i. 程序接口（如函数名）
      ii. 交换数据的数据结构（如参数类型）
      iii. 数据的表示（如字符串编码方式）

需求工程过程



1. 可行性分析（该阶段也有系统定义和功能的简要描述）
   a) 机构的总体目标
   b) 现有技术条件、预算、时间
   c) 与其他已存在的系统集成
2. 导出和分析：
   a) 过程：领域理解、需求收集、分类、冲突解决、优先排序、需求检查
   b) 面向视点（保证需求的完整性）
      i. 视点类型：数据源或数据接收者、框架（比较同类型的系统模型）、服务接收者（交互系统）
      ii. 视点不一定是正当用户，也包括那些非法的、恶意的用户
      iii. 视点与服务是多对多的关系
      iv. VORD 方法过程模型



- 视点识别（提供、接收服务）：卡片法
- 视点组织（分类、层次结构）
- 视点文档（视点和服务描述）
- 视点系统映射（转为面向对象的设计）

   c) 场景分析
      i. 开始状态
      ii. 正常的事件流
      iii. 出错与出错处理
      iv. 同时发生的活动信息
      v. 结束状态
3. 需求描述（体系结构图、用例图、序列图等等）
4. 需求检查
   a) 有效性、准确性、一致性、完整性、现实性、可验证性

b) 有效性验证

i. 评审

ii. 原型开发

iii. 测试案例生成

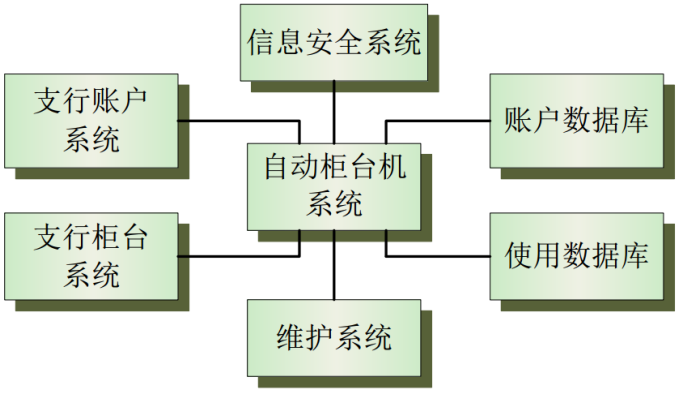
iv. 一致性分析

系统模型

1. 上下文模型

描述系统边界、描述一个系统与其他系统间的关系；

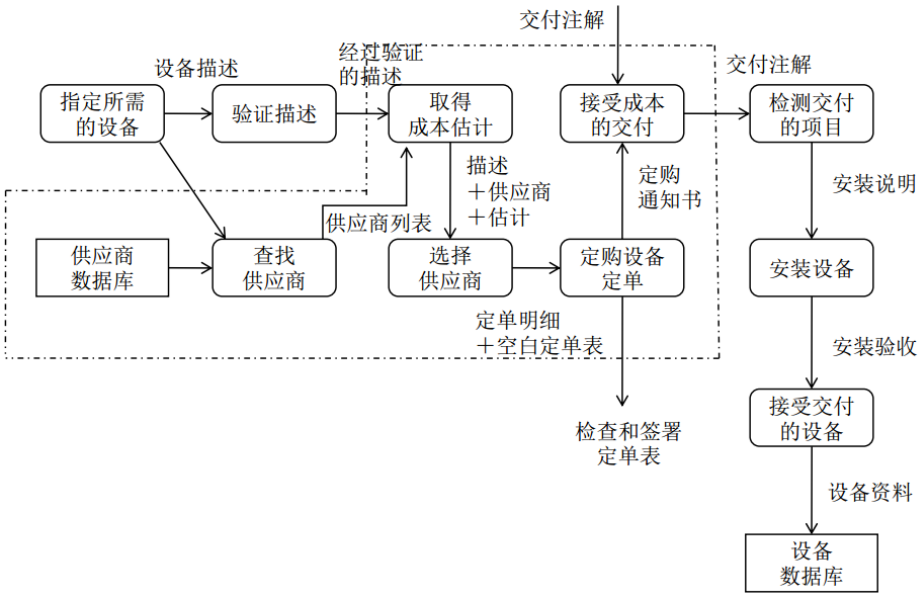
方块内为系统的名称，方块间通过实线连接



2. 过程模型

描述系统中支持的活动过程

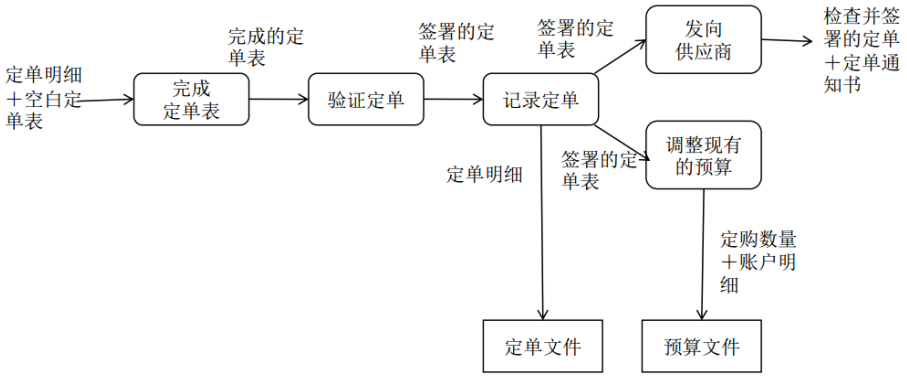
直角方块为实体，圆角方块为活动，箭头表示工作流的方向



3. 行为模型

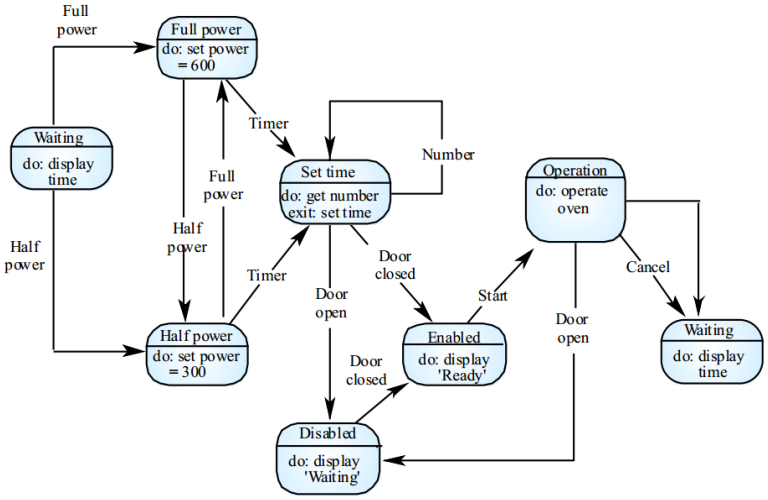
a) 数据流模型

直角方块为实体，圆角方块为活动，箭头表示数据流动的方向（与过程模型的区别）



b) 状态机模型

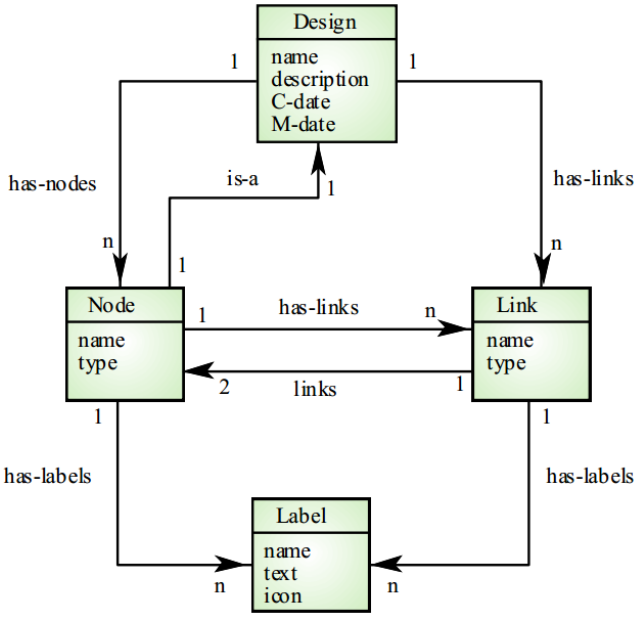
圆角方块为状态，箭头表示某事件发生后状态的改变



4. 语义模型

描述系统数据加工的过程；

方块描述实体及其属性，箭头表示从属、包含等关系



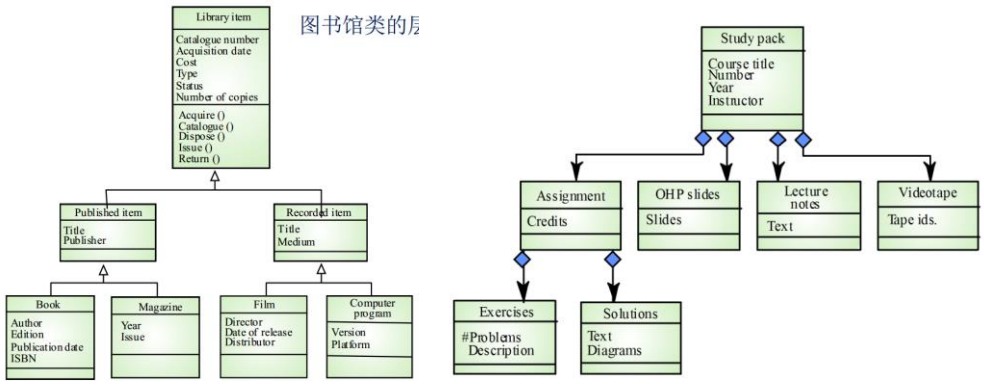
*可以为上述实体、属性、关系建立数据字典

名字	描述	类型	日期
Has-labels	在节点或关联实体和类型标签实体间的1：N关系	关系	5.10.1998
Label	存放节点或关联的结构化或非结构化信息。标签由一个图标和相关的文本表示	实体	8.12.1998
Link	表示设计实体的节点间的1：1关系，关联具有类型和名字	关系	8.12.1998
name(label)	每个标签具有一个说明类型的名字。该名字在设计中的标签类型必须唯一	属性	8.12.1998
name(node)	每个节点名字在整个设计中必须唯一，名字可以长达64个字符	属性	9.11.1998

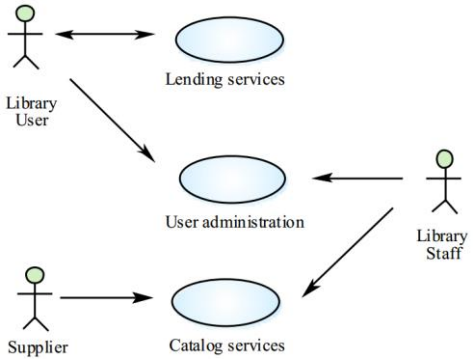
5. 对象继承/聚合模型

方块表示对象类（名称、属性、方法），连线表示对象类间的关系（继承、聚合）；

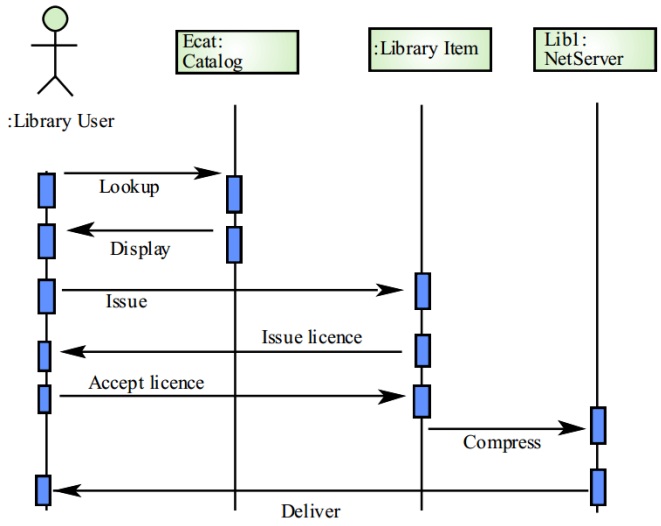
三角形表示继承、菱形表示聚合



6. 对象行为交互建模
   a) 用例图
      描述对象间的交互（概要）
      小人表示对象，椭圆表示服务，箭头表示交互；



   b) 序列图
      描述对象间的交互（强调过程细节）；
      小人表示用户, 方块表示实体, 竖线表示时间轴, 时间轴上的小方块表示生命周期, 箭头表示实体间的交互

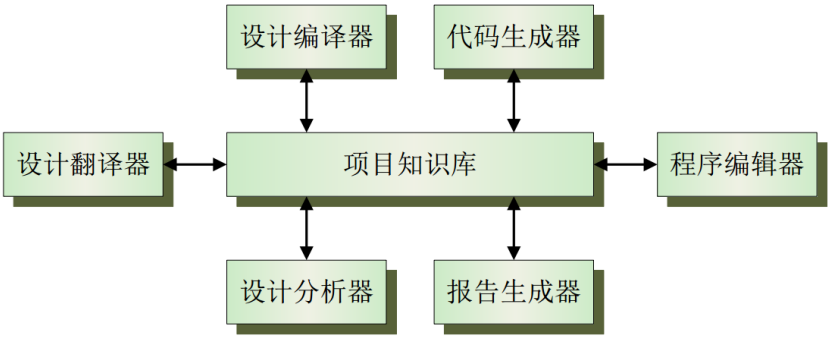


软件原型

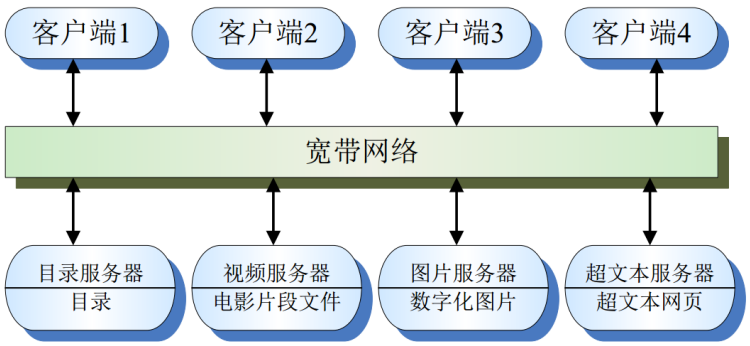
1. 原型的作用
   a) 需求导出
   b) 需求有效性验证（只能验证部分需求）
   c) 降低需求风险
   d) 早期可以得到一个能工作的系统
   e) 可以支持用户培训和系统测试
2. 原型存在的问题
   a) 部分需求无法建立原型
   b) 无法作为法律上的合约
   c) 非功能性需求无法充分测试
3. 进化式：从需求较明确的开始，最终得到交付的系统
   抛弃式：从需求较模糊的开始，最终得到可执行的原型和系统描述，系统需要重做

体系结构的设计

1. 子系统：独立构成的系统，不依赖其他子系统提供的服务
   模块：提供服务给其他组件的组件
2. 设计过程
   a) 系统结构化（分解子系统，识别它们之间的通信）
   b) 控制建模（建立各部分之间的控制关系）
   c) 模块分解（将子系统进一步分解成模块）
3. 容器模型
   多个子系统共享一个中央数据库
   优点：
   a) 共享数据
   b) 子系统无需关心数据如何管理
   c) 数据处理活动集中进行
   缺点：
   a) 数据模型必须统一，不可避免需要妥协
   b) 数据进化比较困难
   c) 数据分布比较困难



4. 客户端 – 服务器
   一组客户机通过网络向服务器请求特定的服务
   优点：
   a) 数据分发简单明了
   b) 硬件成本较低
   c) 容易增加或升级
   缺点：
   a) 子系统间没有统一的数据模型，数据交换效率不高
   b) 各个服务器存在冗余管理
   c) 客户机难以知道具体有哪些服务器和服务



5. 抽象机模型

分层结构

优点：

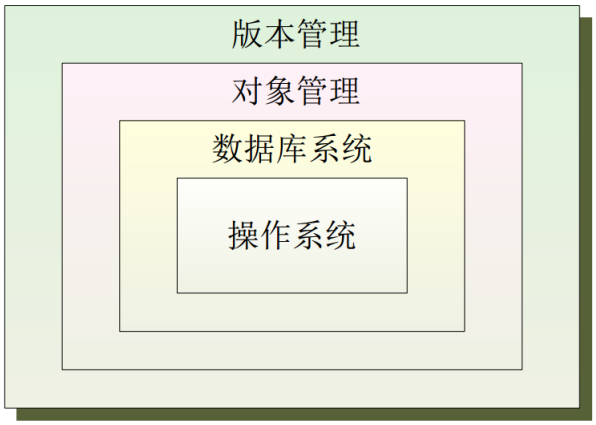
a) 各层子系统独立增量开发

b) 各层接口改变时只有相邻层受到影响

c) 安全性比较高

缺点：

a) 构建系统通常比较困难



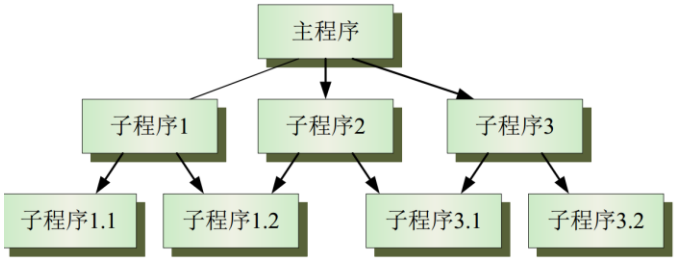
6. 控制模型

关注子系统间的控制流

a) 集中式控制

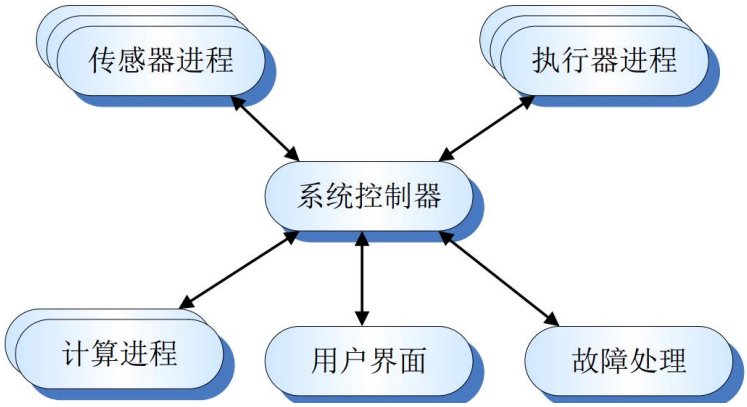
i. 调用 – 返回模型

自上而下的控制过程，适合顺序系统



ii. 管理者模型

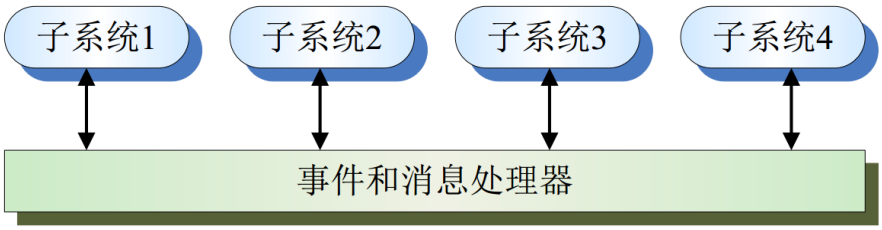
一个系统组件协调、控制其他子系统，适合并发系统



b) 事件驱动

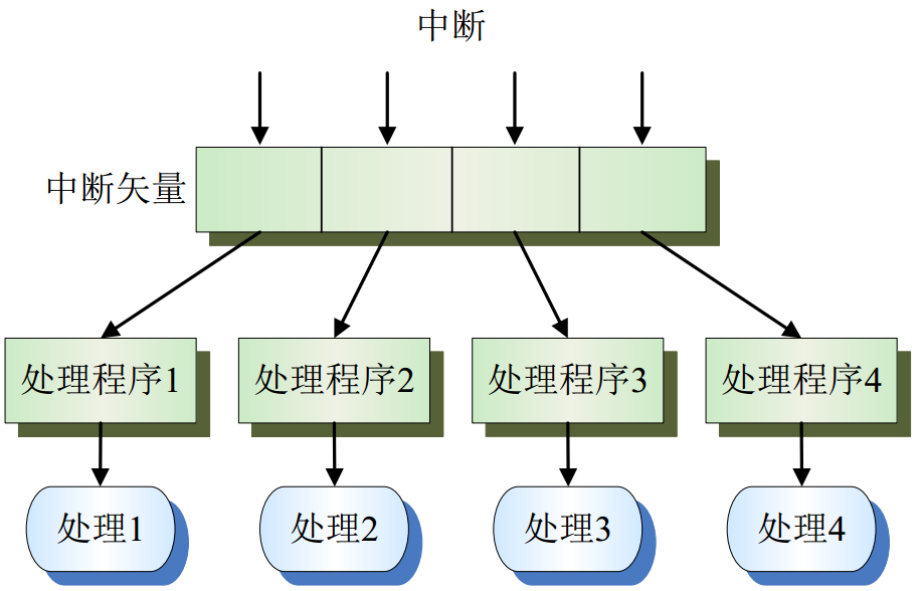
i. 广播模型

子系统注册特定事件，事件发生后控制流传递给能够处理该事件的子系统



ii. 中断驱动

适合实时系统，为中断类型定义专门的中断处理，由硬件将中断转到对应的处理程序，但是编程复杂且难验证

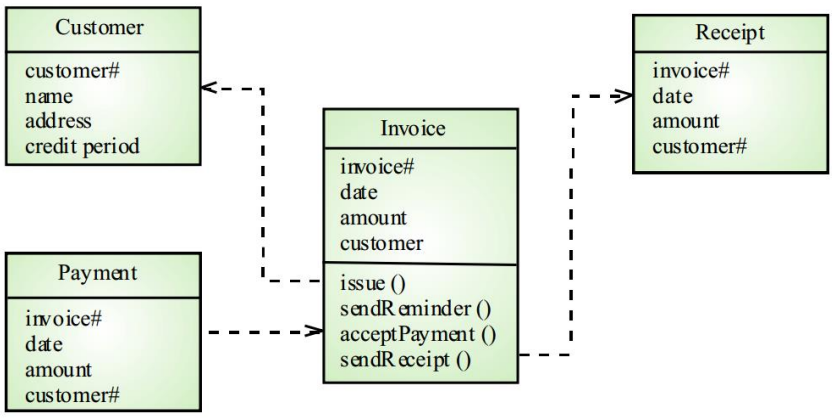


7. 模块分解模型

将子系统分解成模块

a) 对象模型

多个对象类之间通过虚线箭头连接，表示类间的控制关系

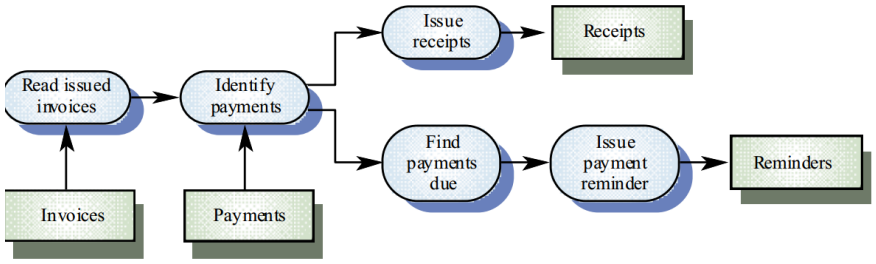


b) 数据流模型（管道模型）

当转换是顺序时称为批处理顺序模型；

不适合交互式系统；

直角方块是数据，圆角方块是数据处理模块，箭头表示数据的流动方向



8. 领域相关的体系结构
   a) 自上而下的参考模型（如 OSI）
   b) 自下而上的类模型（如编译器）

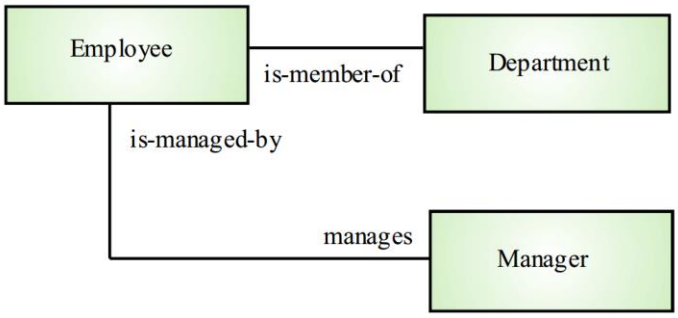
面向对象的设计

1. 对象和对象类

   对象是实体，是对象类的实例；

   对象类是对象的模板，可以从其他对象类继承属性和服务
2. 对象通过消息传递消息，消息通常通过过程调用来实现
3. 继承是 UML 泛化的实现
   a) 优点：实体分类，设计与编程上的重用，明显的层级结构
   b) 问题：对象类不再独立，理解困难
   c) 继承引入了复杂性，对于严格的系统应当避免
4. UML 关联

   方块表示对象，连线及其文字说明表示关联



5. 面向对象设计过程
   a) 定义**上下文**和系统的使用模式（静态的上下文模型或动态的交互模型）
   b) 设计**系统体系**结构（容器模型等）
   c) **识别**出系统中的主要对象
      i. 自然语言描述，名词 -> 对象或属性，动词 -> 操作或服务
      ii. 使用真实实体
      iii. 行为描述，谁参与什么行为
      iv. 情景分析
   d) 开发设计**模型**（子系统、系列、状态机等）

e) 描述对象**接口**（UML 或 JAVA）

用户界面设计

1. 设计原则
   a) 用户熟悉
   b) 一致性（符号、名称等）
   c) 意外最小化
   d) 可恢复性（撤销）
   e) 用户辅助（帮助手册）
   f) 用户差异性（允许字体调整）
2. 交互形式

   直接操作、菜单选择、表格填写、命令语言、自然语言

交互类型	主要优点	主要缺点	应用实例
直接操作	快速和直观的交互 容易学习	较难实现 只适合于任务和对象有视觉隐喻的情况	视频游戏 CAD系统
菜单选择	避免用户错误 只需很少的键盘输入	对有经验用户操作较慢 当菜单选择很多时会变得很复杂	绝大多数一般用途的系统
表格填写	简单的数据入口 容易学习	占据很多屏幕空间	库存控制 个人贷款处理
命令语言	强大灵活	较难学习 差的错误管理	操作系统 图书馆信息检索系统
自然语言	适合偶然用户 容易扩展	需要键入的太多 自然语言理解系统不可靠	时刻表系统 WWW信息检索系统

3. 信息表示
   a) 直接显示（数据、文本等） & 转换成某种方式显示（图形化等）
   b) 静态信息（初始化后不再改变） & 动态信息（随会话过程的改变而改变）
   c) 数字显示（准确的数据） & 模拟显示（图形化等模糊的数据）
   d) MVC 方法支持数据的多种表示
4. 色彩
   a) 颜色不宜过多
   b) 不同颜色，不同任务
   c) 允许用户控制颜色
   d) 先单色设计后增加颜色
   e) 颜色编码保持一致
   f) 颜色配对防止冲突
   g) 颜色变化表示状态变化
5. 错误信息
   a) 礼貌、简明、一致、建设性
   b) 设计因素
      i. 上下文（用户当前状态）
      ii. 用户经验
      iii. 用户技能水平
      iv. 积极、主动地显示消息
      v. 用户所在地区文化

6. 帮助系统
    a) 多个入口
    b) 提示用户当前阅读的位置
    c) 导航、跳转
7. 用户文档：考虑不同的用户

检验和有效性验证（V&V）

1. V&V 是全生命周期的过程
2. V&V 不意味着程序完全没有缺陷，而是表明系统足以满足使用要求，这取决于信任程度；信任程度依赖于设计目标（软件功能）、用户期望和市场环境；
3. 缺陷测试：发现缺陷
   统计性测试：反映用户输入的频度，用于可靠性估计
4. 静态分析（对弱类型检查的语言特别有用）
    a) 控制流（代码段的出入口、条件、循环等）
    b) 数据使用（初始化、赋值、引用等）
    c) 接口（声明与使用一致）
    d) 信息流（输入输出间的关系）
    e) 路径（分析可能的路径）

软件测试

1. 组件测试和集成测试

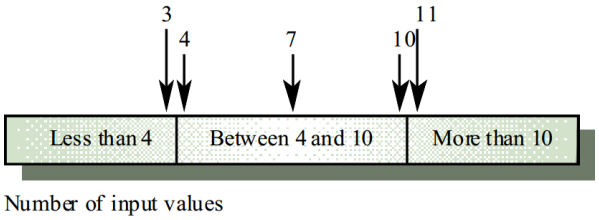
	组件测试	集成测试
对象	个别组件	系统或子系统
负责人	组件开发者	独立测试团队
基于	开发者经验	系统描述

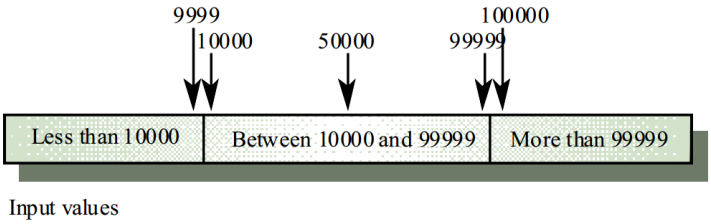
2. 测试优先级：优先关注整体（系统）、旧功能、典型情形
3. 测试数据：测试时输入的数据
   测试用例：测试数据和根据系统描述预期的输出
4. **黑盒测试和白盒测试**
   一般只用黑盒测试，白盒测试成本比较高；
   而且一般只有个别小规模的组件测试会用白盒测试，像大规模的集成测试是不可能用白盒测试的

	黑盒测试（功能测试）	白盒测试（结构化测试）
用例来源	系统描述	程序结构
测试规划	软件过程早期	软件过程后期
目标	发现程序可能出现的缺陷	所有语句都执行一遍

5. 等价划分

测试用例通常选择等价划分的边界情况和典型情况（一般是中间值）





6. 路径测试

a) 目标：所有路径都执行一遍

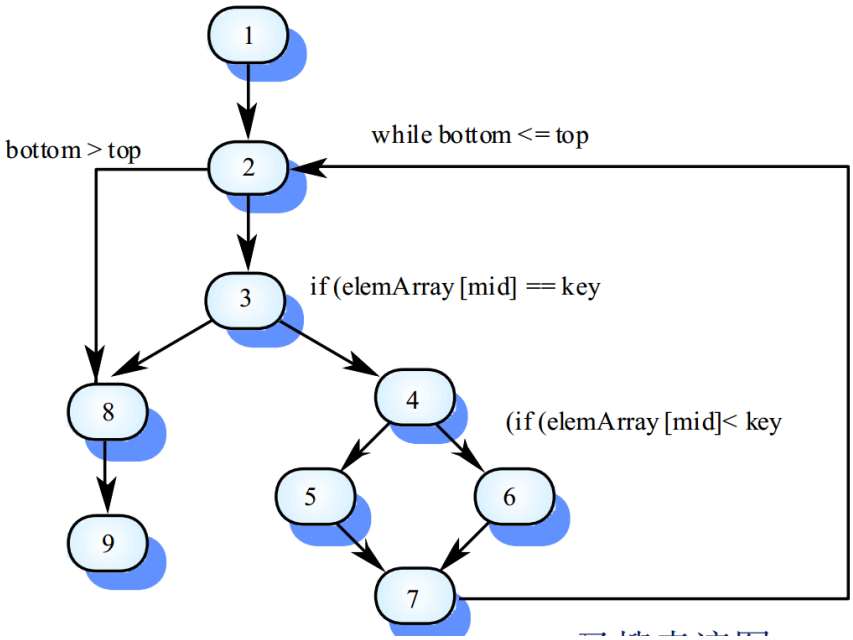
b) 程序流图：边表示控制流

c) 环路复杂性 = 边数 – 节点数 + 2

i. 测试所有语句的最少测试用例数 = 环路复杂性

ii. 程序中的条件数 = 环路复杂性

iii. 只能确保所有路径都执行，但不能保证所有路径组合都执行



边数 11，节点数 9，环路复杂性=11-9+2=4；最少需要 4 个测试用例

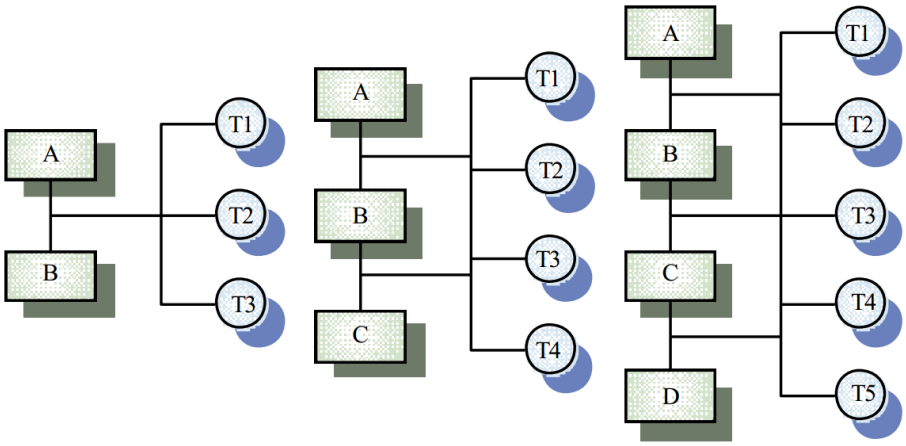
可设计用例序列分别为 12389，1234572，1234672，123467289

7. 集成测试

a) 主要困难是定位错误，可以通过增量式集成测试减轻这个问题；

b) 增量式集成测试

逐个组件集成，每集成一个组件就测试一次，每次测试都包含之前的所有测试内容和新增的测试内容，可以借助一些自动化测试技术；



c) 自顶向下测试和自底向上测试

	自顶向下测试	自底向上测试
含义	从高层开始，需要程序“桩”，即一些空的底层函数，用来打印相关的程序轨迹	一层层集成组件并逐一测试
体系结构	更容易发现错误	
系统演示	早期能有一个功能有限的演示系统	
测试过程		通常比较容易
测试观察	都需要额外代码来观察结果	

d) 接口测试

i. 接口类型：参数、共享内存、过程、信息传递

ii. 接口错误：误用、误解、时机错误

e) 强度测试

8. 面向对象测试

a) 测试对象关联的单个操作、单个对象类、对象集群、面向对象系统

b) 对象继承使得测试更加困难

c) 面向对象系统中集成的层次不明显，通常根据对象操作和系统特征来识别对象集群进而进行集群测试

软件成本估算

1. 基本问题：工作量（人·天）、时间（小时、天等）、总成本（元）
2. 项目估算和进度安排是交叉进行的管理活动
3. 软件成本构成：

a) 软硬件

b) 差旅费和培训费用

c) 工作成本（最主要）

d) 其他（网络、办公场地等）

4. 成本和报价之间没有一个简单的关系

影响报价的因素：

a) 市场机遇（进入一个新的市场）

b) 成本估算的不准确性（没有把握，增加意外费用）

c) 合同条款（源代码的所有权）
d) 需求易变性（需求变更可能引起的价格纠纷）
e) 经济状况（行情不好，没啥好生意，但不能让员工闲着）

5. 生产率

生产软件和相关文档的效率的度量；
但不是面向质量的

a) 面向**规模**的度量：根据活动输出的量（代码行数、文档页数等）
存在的问题：编程语言越低级，代码越冗长，计算出来的生产率越高
b) 面向**功能**的度量（功能的多少）
i. **功能点**方法
功能点分类 – 为每个类别赋予权值 – 计算乘积和
根据功能点估算代码行数：
LOC = AVC * FPs （AVC 是基于语言的系数）
功能点计数是非常主观的，自动化也是不可能的
ii. **对象点**方法
根据以下三个内容进行加权估算：
独立的**显示屏幕数**
生成的**报表数**
为辅助 4GL 代码而开发的 **3GL 模块数**
对象点估算比功能点容易，可在开发过程很早期使用；
c) 影响因素
i. 应用领域经验（估算者先前的经验）
ii. 过程质量（开发过程的质量要求）
iii. 项目规模（主要是考虑合作成本）
iv. 技术支持（额外的支持技术或工具）
v. 工作环境
d) 生产率与质量的关系复杂，通常估算成本只考虑生产率

6. 估算技术

从根本上讲，估算是基于经验的；
只有系统完成后才能精确知道软件的规模（开发过程越到后期，规模估计越准确）

a) 算法成本建模（利用历史信息来估计）
工作量 = $A * Size^B * M$
A 是反映机构情况的常数因子；
B 是反映大型项目工作量的非线性因子，Size 是项目规模；
M 是反映不同过程、产品、开发特征的混合因素的因子
b) 专家判定（相对廉价）
c) 类比估计（与另一个相同应用领域的项目类比）
d) 帕金森定律（工作占满所有可用时间？？？）
e) 根据客户预算报价（有时是唯一可用的方法）

7. 上述估算技术既可以自上而下也可以自下而上

自上而下：
适合体系结构、组件识别未知的情况；
容易低估解决底层技术难题的成本；

自下而上：

适合体系结构、组件识别尤其是详细设计已知的情况；

容易低估系统级的集成、配置管理、文档的成本

8. COCOMO 模型：基于项目经验的经验模型

   COCOMO2 模型的分层（三层模型）：

   a) 早期原型层（基于对象点）
   b) 早期设计层（基于功能点）
   c) 后体系结构层（基于源代码行数）

9. 要求的人员不能单纯用所需工作量除以开发时间来计算（额外的合作成本、管理成本）