考试范围等合并

1-3特供

1软工概述

0 软件

软件

软件 = 程序 + 数据 + 文档

- 程序: 是能够完成预定功能和性能的可执行指令序列
- 数据: 是使程序能够适当处理信息的数据结构
- 文档: 是开发、使用和维护过程程序所需要的图文资料

1 软件危机

软件危机是 软件开发和维护过程中遇到的一系列严重问题

两个方面

- **如何开发**软件，以满足日益增长的软件需求
- **如何维护**数量不断膨胀的已有软件

表现

1. 开发成本和进度估不准
2. 用户不满意
3. 质量靠不住
4. 可维护性差
5. 没有适当文档资料
6. 软件成本在计算机系统总成本中所占的比例逐年上升
7. 开发生产率低

原因

- 客观原因
  - ① 软件是计算机系统中的逻辑部件，缺乏“可见性”，因此管理和控制软件开发过程相当困难
  - ② 软件维护通常意味着改正或修改原来的设计，因此软件较难维护

    - ③ 软件规模庞大，而程序复杂性将随着程序规模的增加而呈指数上升
- 主观原因
    - ① 存在与软件开发和维护有关的许多错误认识和做法
    - ② 对用户要求没有完整准确的认识就匆忙着手编写程序
    - ③ 开发人员只重视程序而忽视软件配置的其余成分（文档和数据等）
    - ④ 软件开发人员轻视维护

2 软件工程

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

基本原理

- 用分阶段的生命周期计划严格管理
- 坚持进行阶段评审
- 实行严格的产品控制
- 采用现代程序设计技术
- 结果应能清楚地审查
- 开发人员少而精
- 承认不断改进软件工程实践的必要性

软件工程方法学

三要素

方法、工具、过程

传统方法学（生命周期方法学）

- 采用**结构化技术**完成软件开发各项任务
- 把软件生命周期的全过程依次划分为**若干阶段**
- 每个阶段开始和结束都有**严格标准**
- 每个阶段结束后要有**严格审查**

面向对象方法学

- 把对象作为融合了**数据及在数据上的操作行为**的统一的软件构件
- 把所有对象划分为**类**

- 按照父类与子类的关系，把若干类组成**层次结构**的系统
- 对象彼此间仅能通过发送**消息**互相联系

3 软件生命周期

软件生命周期由 软件定义、软件开发、软件维护组成

- 定义
- 开发
- 维护

定义

问题定义

可行性研究

需求分析

开发

总体设计

详细设计

编码和单元测试

综合测试

维护

软件维护

4 软件过程

是为了获得高质量软件所需要完成的一系列任务框架，它规定了完成任务的工作步骤。通常用软件生命周期模型来描述软件过程。

常用软件过程模型

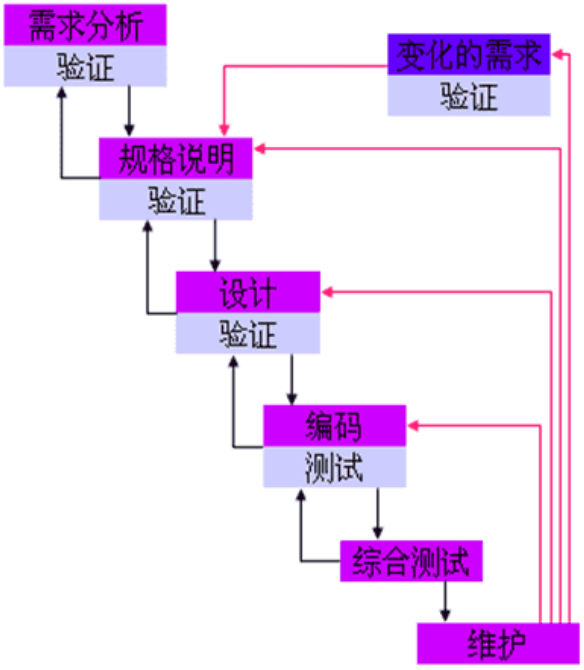
- 瀑布模型
- 快速原型模型
- 增量模型

- 螺旋模型
- 喷泉模型
- Rational统一过程

瀑布模型

使用最早应用最广

实际瀑布模型带**反馈环**



特点

1. 阶段具有**顺序性**和**依赖性**

   前一阶段结束后一阶段开始，前一阶段输出文档，后一个阶段输入文档。

2. **推迟实现**观点

   瀑布模型在编码前设置系统分析、系统设计，推迟程序物理实现，保证前期工作扎实。

3. **质量保证**观点

   1. 每阶段都必须完成完整、准确的文档。该文档是软件开发时人员间通信、运行时维护的重要依据。
   2. 每阶段结束前对文档评审。

评价

优点

- **强迫开发人员使用规范的方法**
- **严格规定了每个阶段提交的文档**
- **要求每个阶段交出的所有产品都必须经过质量保证小组的仔细验证**
- **对文档的约束，使软件维护变得容易一些，且能降低软件预算**

缺点

- **在软件开发的初期阶段就要求作出正确的、全面的、完整的需求分析对许多应用软件来说是极其困难的**
- **在需求分析阶段，当需求确定后，无法及时验证需求是否正确、完整**
- **作为整体开发的瀑布模型，由于不支持产品的演化，缺乏灵活性**

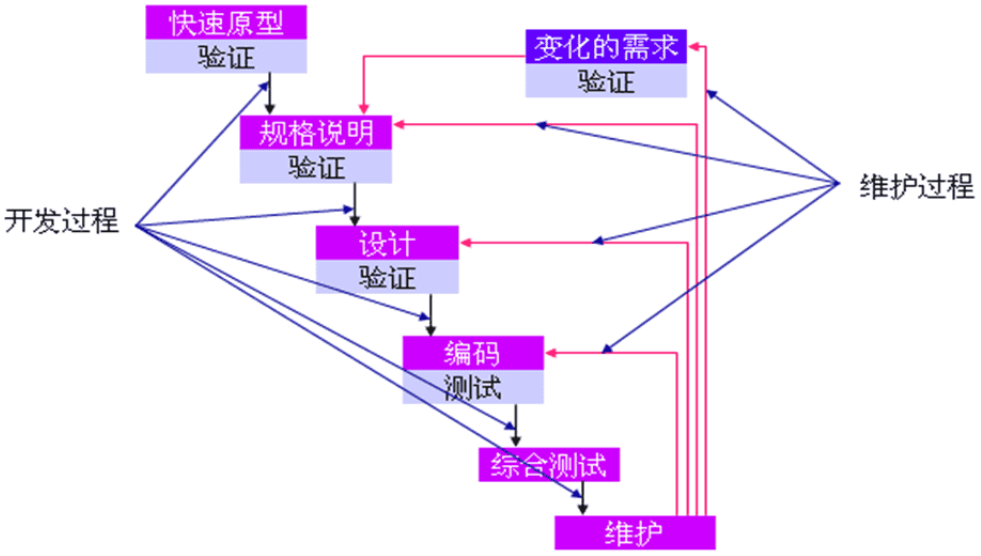
（可以使用改进的带反馈环的）

适用范围

- **用户的需求非常清楚全面，且在开发过程中没有或很少变化**
- **开发人员对软件的应用领域很熟悉**
- **用户的使用环境非常稳定**
- **开发工作对用户参与的要求很低**

快速原型模型

快速原型是快速建立起来的可以在计算机上运行的程序，它所能完成的功能往往是最终产品能完成的功能的一个子集。



特点

快速原型模型是**不带反馈环**的，开发基本上是线性顺序进行的

评价

优点

- **开发的软件产品通常满足用户需求**
- **软件产品开发基本是线性过程**

缺点

- **准确原型设计困难**
- **原型理解可能不同**
- **不利于开发人员创新**

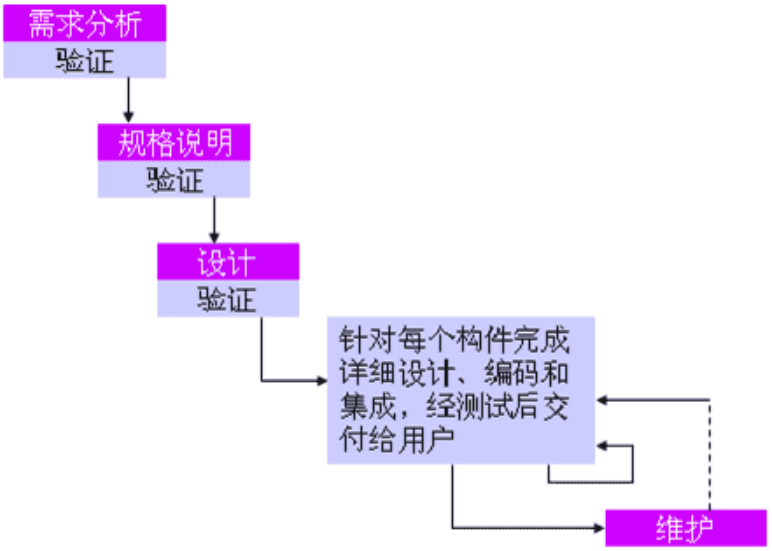
适用范围

- **对所开发的领域比较熟悉而且有快速的原型开发工具**
- **项目招投标时，可以以原型模型作为软件的开发模型**
- **进行产品移植或升级时，或对已有产品原型进行客户化工作时**

增量模型

也称为渐增模型

先完成一个系统子集的开发，再按同样的开发步骤增加功能，如此递增下去直至满足全部系统需求



特点

每个构件由多个相互作用的模块构成，并且能够完成特定的功能

> **区别于瀑布模型和快速原型模型：**
>
> 瀑布模型和快速原型模型是一次把满足所有需求产品提交给用户。
>
> 增量模型是分批向用户提交产品

第一个增量构件往往实现软件的基本需求，提供最核心的功能。把软件产品分解成增量构件时，应该使构件的规模适中。

分解时唯一必须遵守的约束条件是：当把新构件集成到现有软件中时，所形成的产品必须是可测试的

评价

优点

- **短时间内可提交完成部分功能**
- **逐渐增加产品功能，用户适应产品快**

缺点

- **要求软件体系结构必须是开放的**
- **增量构件划分以及集成困难**
- **容易退化为边做边改模型**

适用范围

- 进行已有产品升级或新版本开发
- 对完成期限严格要求的产品
- 对所开发的领域比较熟悉而且已有原型系统

螺旋模型

把它看作在每个阶段之前都增加了**风险分析过程**的快速原型模型

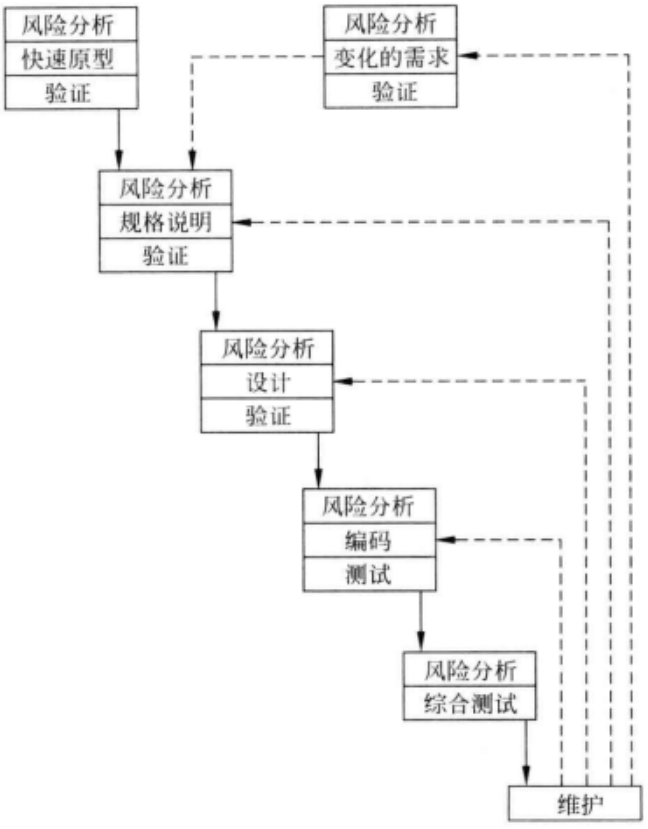
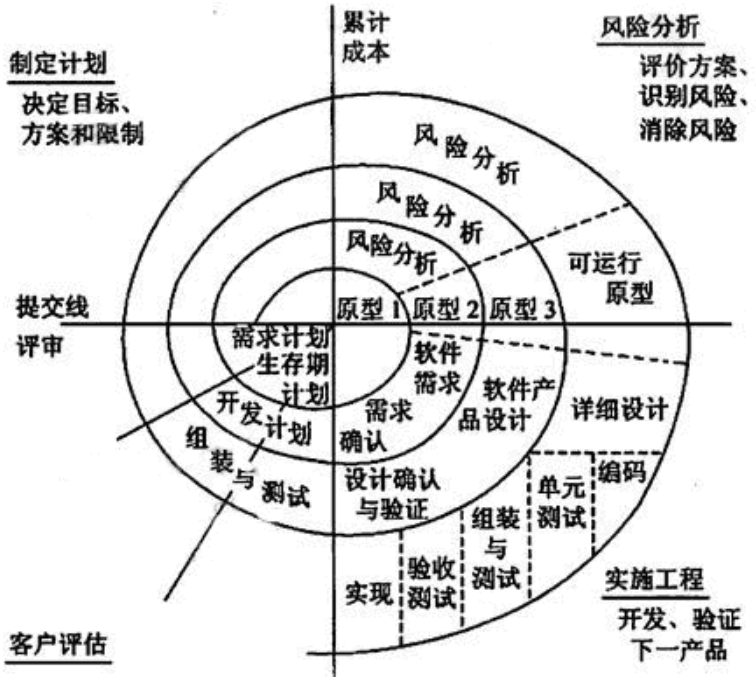


图 1.7　简化的螺旋模型



笛卡尔坐标四象限表达四方面活动：

- 制定计划：确定目标、选定方案、设定约束条件。
- 风险分析：评估方案，识别和消除风险。
- 实施工程：软件开发。
- 客户评估：评价开发工作，计划下一阶段工作。

沿螺线自内向外每旋转一圈开发出更完善新版本。

评价

优点

- **利于把软件质量作为软件的开发目标**
- **减少测试**
- **维护和开发不分开**

缺点

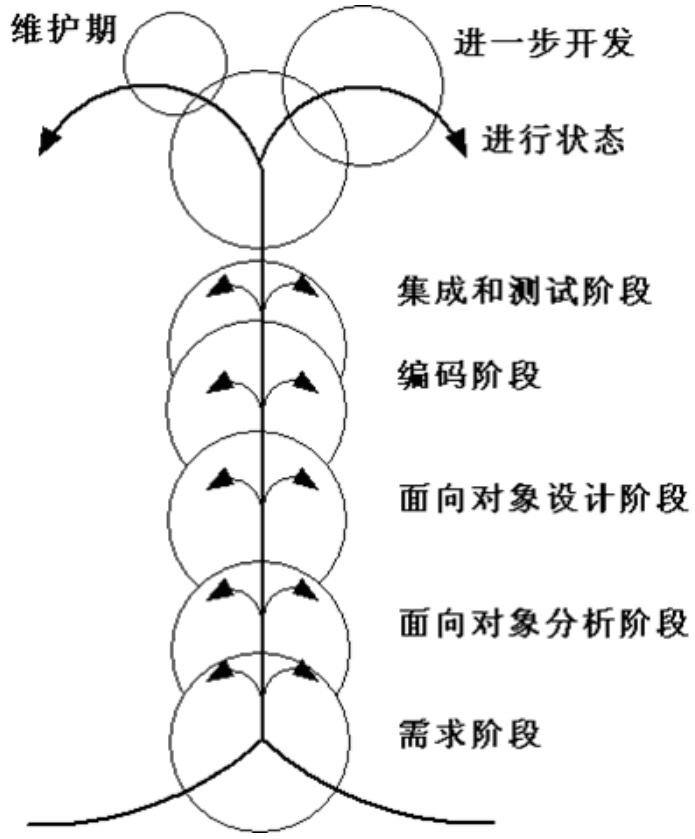
- **风险估计困难**
- **过多的迭代次数会增加开发成本，延迟提交时间**

适用范围

适用于内部开发的大规模软件项目

喷泉模型

面向对象生命周期模型，体现迭代（求精，系统某部分常被重复工作多次，相关功能在每次迭代中逐渐加入演进系统）和无缝（分析、设计、编码各阶段间不存在明显边界）特性。



特点

- 喷泉模型是一种以用户需求为动力，以对象为驱动的模型，主要用于描述面向对象的软件开发过程
- 体现了面向对象软件开发过程迭代和无缝的特性

评价

优点

无缝，可同步开发，提高开发效率，节省开发时间，适应面向对象软件。

缺点

可能随时加各种信息、需求与资料，需严格管理文档，审核的难度加大。

适用范围

RUP

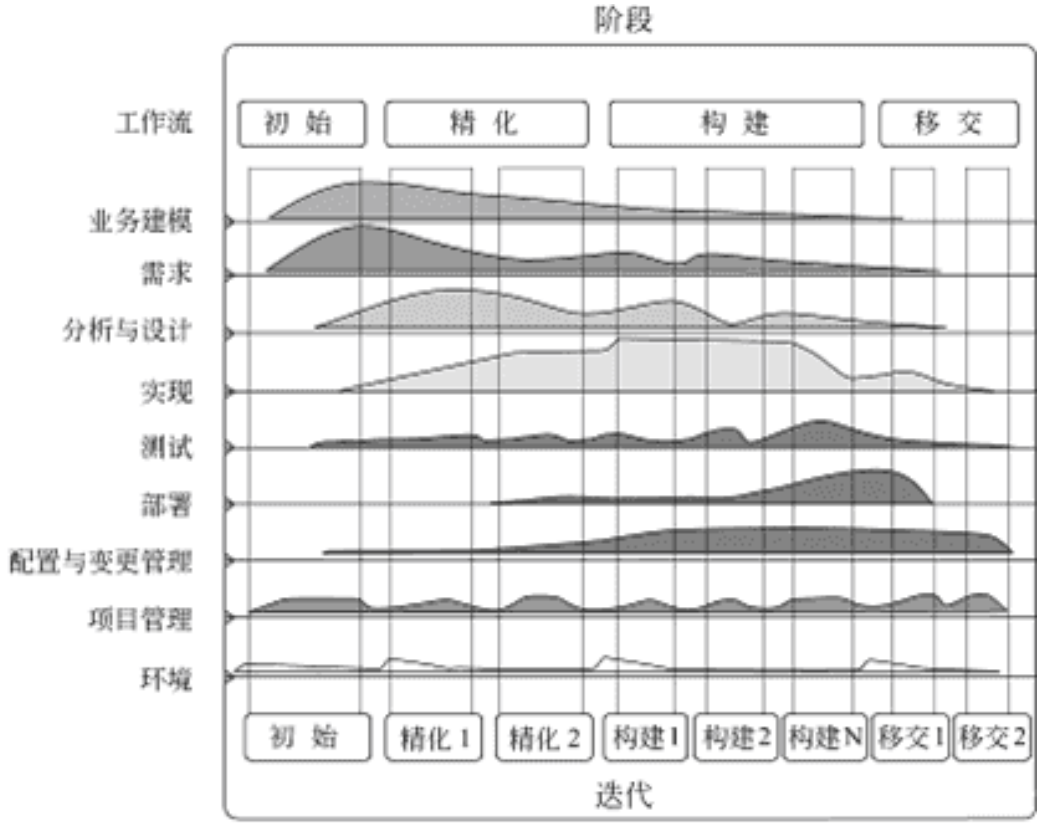
Rational统一过程

由Rational软件公司推出的一种软件过程，该过程强调以迭代和渐增方式开发软件

RUP软件开发生命周期是一个二维的面向对象的生命周期模型

阶段

- 初始
- 精化
- 构建
- 移交



特点

- 采用迭代和渐增的方式开发软件
- 具有多功能性和广泛适用性

评价

优点

不断的版本发布成为一种团队日常工作的真正驱动力

将发现问题、制定方案和解决过程集成到下一代迭代

迭代开发，降低风险

更好地安排产品开发的辅助过程

2可行性研究

1 可行性研究概念

可行性研究的目的不是解决问题，而是确定问题是否值得去解决

具体任务

1.**分析和澄清问题的定义**

2.**导出系统的逻辑模型**（数据流图+数据字典）

3.**根据逻辑模型探索若干种可供选择的解法**（并判断可行性）

可行性

- **经济可行性**

  ?效益>成本

- **技术可行性**

  开发风险，资源

- **操作可行性**

  运行，时间进度，法律

2 可行性研究过程

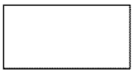
1. 复查系统规模和目标
2. 研究目前正在使用的系统
3. 导出新系统的高层逻辑模型
4. 进一步定义问题
5. 导出和评价供选择的解法
6. 推荐行动方针
7. 草拟开发计划书
8. 写文档提交审查

3 系统流程图

系统流程图是概括地描绘物理系统的传统工具

系统流程图表达的是数据在系统各部件之间流动的情况，而**不是对数据进行加工处理的控制过程**，因此尽管系统流程图的某些符号和程序流程图的符号形式相同，但是它却是物理数据流图而不是程序流程图

符号

符　号	名　称	说　　明
	处理	能改变数据值或数据位置的加工或部件，例如程序、处理机、人工加工等都是处理
	输入输出	表示输入或输出（或既输入又输出），是一个广义的不指明具体设备的符号
	连接	指出转到图的另一部分或从图的另一部分转来，通常在同一页上
	换页连接	指出转到另一页图上或由另一页图转来
	数据流	用来连接其他符号，指明数据流动方向

可以使用分层绘制的方法，先给出大致的路径，然后细化后绘制

4 数据流图

数据流图(DFD)描绘信息流和数据从输入移动到输出的过程中所经受的变换

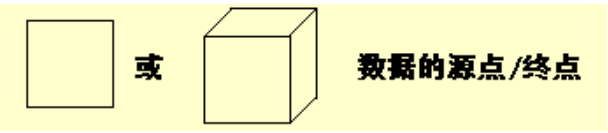
没有任何具体物理部件，只是描绘数据在软件中流动和被处理的逻辑过程

特点

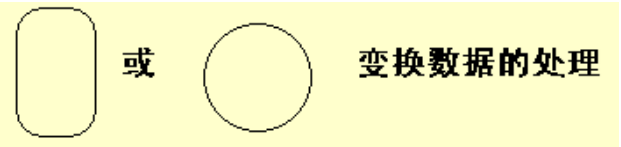
- 数据流图中**没有具体的物理部件**，只是描绘数据在软件中流动和被处理的**逻辑过程**
- 数据流图是**系统逻辑功能的图形表示**，是分析员与用户之间极好的通信工具
- 设计时只需考虑系统**必须完成的基本逻辑功能**，不考虑怎样**具体地实现**这些功能

符号

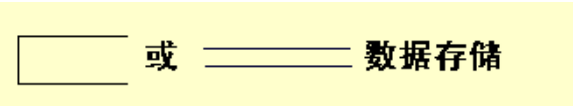
- 四种基本符号：
  - 正方形（或立方体）：表示数据的**源点或终点**

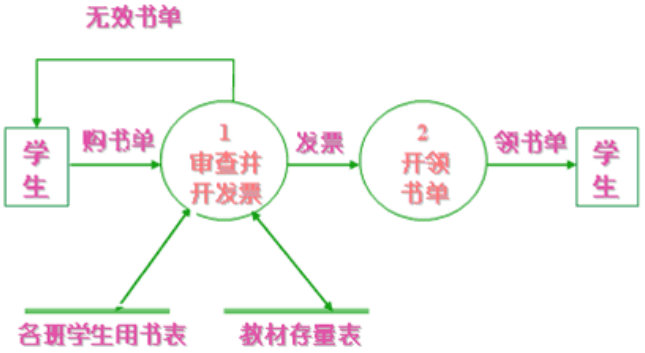
  - 圆角矩形（或圆形）：代表**变换数据的处理**



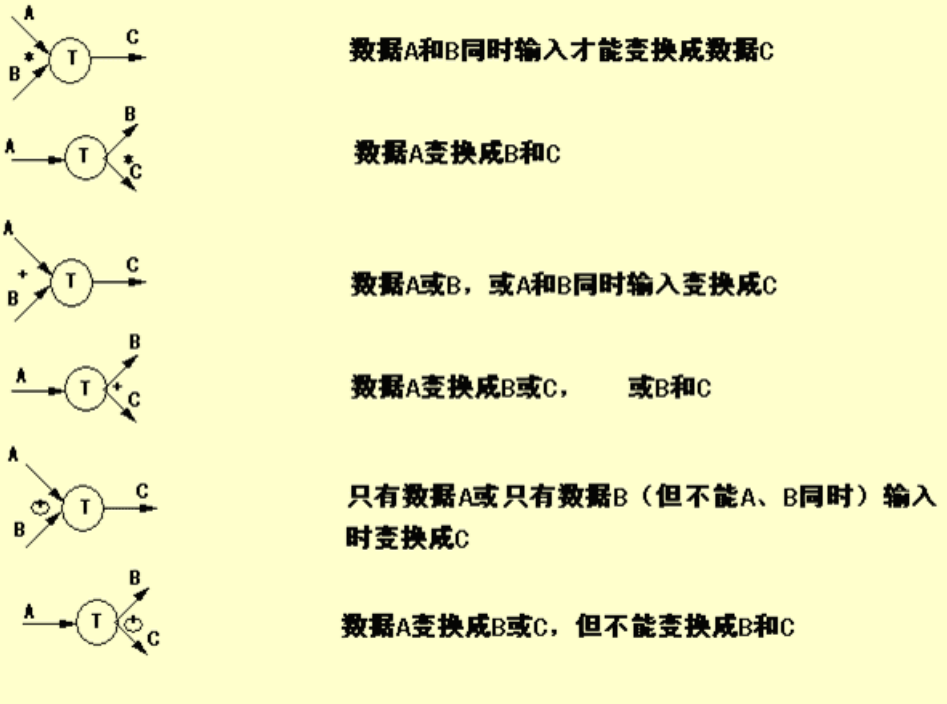
- 开口矩形（两条平行横线）：代表**数据存储**



- 箭头：表示**数据流、即特定数据的流动方向**



- 附加符号



思路

需要绘制时候

1. **考虑数据的源点和终点**
2. 如何处理
3. 考虑数据流的方向，从哪里送哪里
4. 数据存储到哪里

基础模板

- 基本系统模型

  ```
  源点 -> 处理 -> 终点
  ```

- 细化模型

  ```
  加入逻辑等
  ```

注意

- 在数据流图中应该描绘所有可能的数据流向，而不应该描绘出现某个数据流的条件
- 一个处理框可以代表一系列程序、单个程序或者程序的一个模块
- 一个数据存储可以表示一个文件、文件的一部分、数据库的元素或记录的一部分等
- 数据存储是处于静止状态的数据，数据流是处于运动中的数据

- 通常在数据流图中忽略出错处理
- 表示数据的源点和终点相同的方法是再重复画一个同样的符号表示数据的终点
- 代表同一事物的符号出现在n个地方，在这个符号的角上画(n-1）条短斜线做标记
- (绘图)功能进一步分解涉及如何具体实现功能时，不应再分解

命名

1. 数据流（数据存储）命名
    1. 用名词，区别于控制流
    2. 代表整个数据流（数据存储）内容，不仅仅反映某些成分
    3. 不要用缺乏具体含义名字，如“数据”、“信息”
2. 处理命名
    1. 用动宾词组，避免使用“加工”，“处理”等
    2. 应反映整个处理的功能，不是一部分功能
    3. 通常仅包括一个动词，否则分解
3. 数据源点/终点

    不属于数据流图的核心内容

5 数据字典

数据字典是关于数据的信息的集合，也就是对数据流图中包含的所有元素的定义的集合

对象

4类元素

- **数据流**
- **数据元素**
- **数据存储**
- **处理**

1. 数据流

```
数据流名:
    说明:简要介绍作用即它产生的原因和结果
    数据流来源:即该数据流来自何方
    数据流去向:去向何处
    数据流组成:数据结构
    每个数据量流通量:数据量、流通量
```

2. 数据元素

```
数据元素名:

    类型: 数字（离散值、连续值), 文字（编码类型）
    长度:
    取值范围:
    相关的数据元素及数据结构:
```

3. 数据存储

```
数据存储名:

    简述: 存放的是什么数据
    输入数据:
    输出数据:
    数据文件组成: 数据结构
    存储方式: 顺序, 直接,关键码
    存取频率:
```

4. 处理

```
处理名:
    处理编号:反映该处理的层次
    简要描述:加工逻辑及功能简述
    输入数据流:
    输出数据流:
    加工逻辑:简述加工程序、加工顺序
    ...
```

描述

典型的情况是，在数据字典中记录数据元素的下列信息:

1. **一般信息**，名字、别名、描述等等
2. **定义**，数据类型、长度、结构等等
3. **使用特点**，值的范围、使用频率、使用方式(输入、输出、本地)、条件值等等
4. **控制信息**，来源、用户、使用它的程序、改变权、使用权等等

5. **分组信息**，父结构、从属结构、物理位置记录、文件和数据库等等

组成

由数据元素组成数据的方式

- **顺序**
- **选择**
- **重复**
- **可选**

符号

符号	含义
=	等价于（或定义为）
+	顺序连接（确定次序连接）
[|]	或（方括弧中的分量选一个）
{ }n	重复（重复花括弧中的分量）
( )	可选（圆括弧里的分量可有可无）

符　号	含　义	举　例
=	等价于 / 定义为	$x = a$
+	与 / 连接	$x = a + b$
[|]	或 / 选择	$x = [a \mid b]$
m{ }n	重复 m..n 次	$x = 1\{a\}5$
( )	可选	$x = a + (b)$
" "	基本数据元素	$x =$ "0"
..	范围	$x =$ "1".."9"

示例

北京某高校可用的电话号码有以下几类：校内电话号码由4位数字组成，第一位数字不是0。校外电话又分为本市电话和外地电话两类。拨校外电话需要先拨0，若是本市电话则接着拨8位数字（第一位不是0），若是外地电话则拨3位区码后再拨8位电话号码（第一位不是0）

- **电话号码**=[校内电话 | 校外电话]

- **校内电话**=非零数字+三位数字
- **非零数字**=[1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9]
- **三位数字**=3{数字}3
- **数字**=[0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9]
- **校外电话**=[本市电话 | 外地电话]
- **本市电话**=0 + 八位非零开头数字
- **八位非零开头数字**=非零数字+七位数字
- **七位数字**=7{数字}7
- **外地电话**=0 + 三位区码 + 八位非零开头数字
- **三位区码**=三位数字

例如:
学生成绩通知={ {{学号+姓名+{ {{课程名称+成绩} }}+(补考课程名称+补考时间+补考地点)} }}所有注册学生
学生奖励通知={ {{学号+姓名+[一等奖 | 二等奖 | 三等奖]} }}所有获奖学生

用途

(简单)

1.作为分析阶段的工具

2.数据字典中包含的数据元素的控制信息是很有价值的

3.数据字典是开发数据库的第一步，而且是很有价值的一步

3需求分析

1 需求分析的任务

1. 建立分析模型
2. 编写需求说明

4 实体联系图（E-R图）

数据模型中包含3种相互关联的信息：数据对象、数据对象的属性及数据对象彼此间相互连接的关系

数据对象

对软件必须理解的复合信息（复合信息是指具有一系列不同性质或属性的事物，仅有单个值的事物不是数据对象）的抽象

> 可以由一组属性来定义的实体都可以被认为是数据对象

实体联系图

矩形方框:实体
菱形框:联系
圆角矩形:属性

> 见MySQL – 绘制ER

评价

- **E-R模型比较接近人的思维习惯方式**
- **E-R模型使用简单的图形符号表达，便于用户理解**

6 状态转换图

状态转换图(简称为状态图)通过描绘系统的状态及引起系统状态转换的事件，来表示系统的行为。此外，状态图还指明了作为特定事件的结果系统将做哪些动作

概念

状态 ：状态是任何可以被观察到的系统行为模式，一个状态代表系统的一种行为模式。

状态规定了系统对事件的响应方式。系统对事件的响应，既可以是做一个(或一系列)动作，也可以是仅仅改变系统本身的状态，还可以是既改变状态，又做动作

- 状态有初态、终态和中间状态
- 一张状态图只能有一个初态，而终态可以没有也可以有多个

事件：事件是在某个特定时刻发生的事情，它是对引起系统做动作或(和)从一个状态转换到另一个状态的外界事件的抽象。简而言之，事件就是引起系统做动作或(和)转换状态的控制信息

事件

3种标准事件：`entry`，`exit`和`do`

- `entry`事件指定进入该状态的动作
- `exit`事件指定退出该状态的动作
- `do`事件则指定在该状态下的动作

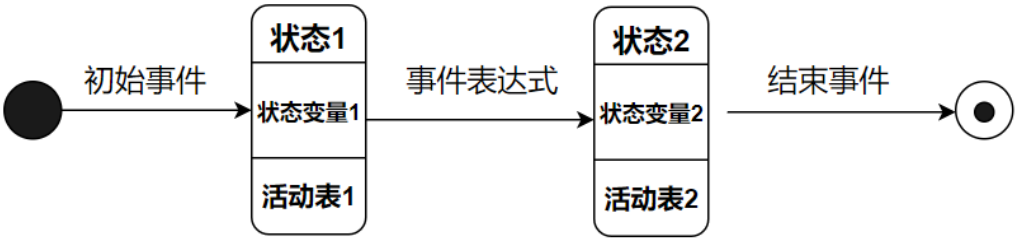
符号

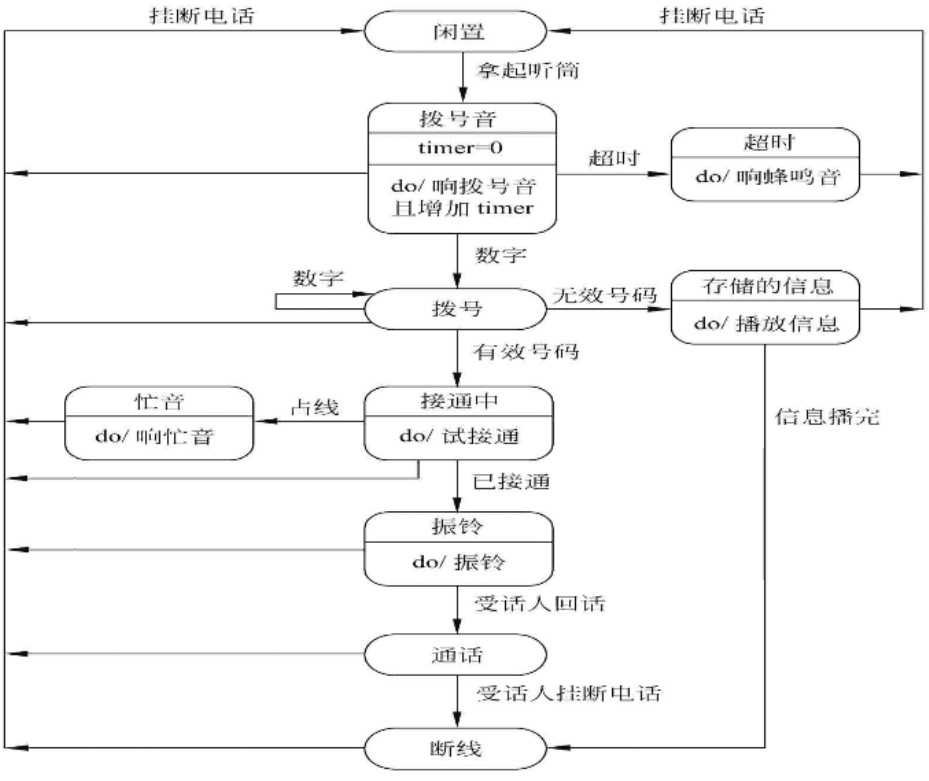
箭头表示，箭头上标事件名。后跟[条件] 、表状态转换条件

状态图中两个状态之间带箭头的连线称为状态转换，箭头指明了转换方向。状态变迁通常是由事件触发的，在这种情况下应在表示状态转换的箭头线上标出触发转换的事件表达式；如果在箭头线上未标明事件，则表示在源状态的内部活动执行完之后自动触发转换

行为：进入某状态所作动作。状态框内do：行为名

对于圆角矩形，可以将其分为上、中、下三部分：上部分是状态的名称；中部分是状态变量的名字和值；下部分是活动表





8 验证软件需求

一致性、完整性、现实性和有效性

4-7特供

```
4 已经全部略过
```

4总体设计

1 设计过程

(1)系统设计阶段

1：设想供选择的方案

2：选取合理的方案

3：推荐最佳方案

(2)结构设计阶段

4：功能分解

5：设计软件结构

6：设计数据库

7：制定测试计划

8：书写文档

9：审查和复查

2 设计原理

结构化设计的概念与原理

- 模块化
- 抽象
- 逐步求精
- 信息隐蔽和局部化
- 模块独立

模块独立

高内聚、低耦合是设计目标

耦合

对一个软件结构内不同模块间互连程序的度量。耦合强度取决于模块接口的复杂程度、通过接口的数据等。耦合度越高，模块独立性越弱

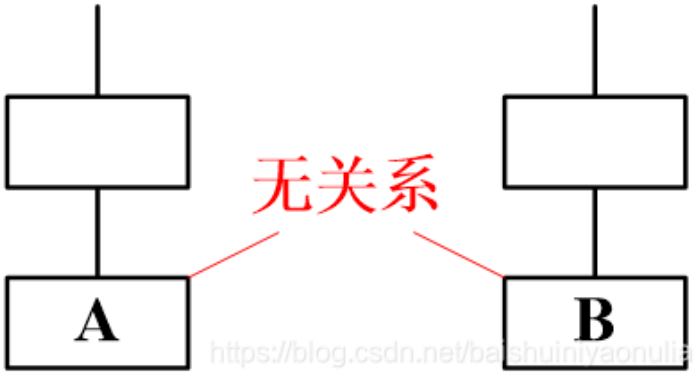
耦合度从低到高

- 完全独立
- 数据耦合
- 特征耦合
- 控制耦合
- 外部耦合
- 公共耦合
- 内容耦合

完全独立

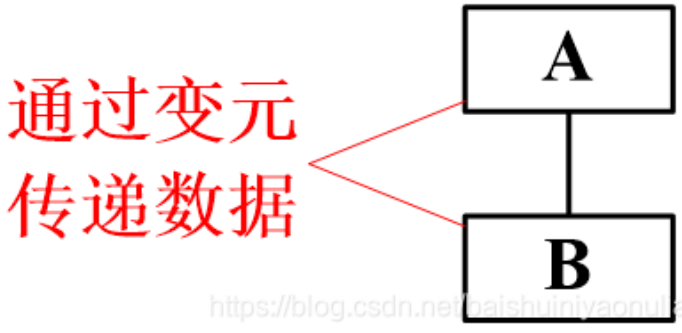
（非直接耦合）

每一个都能独立地工作而不需要另一个模块的存在



数据耦合

参数交换信息，而且交换的信息仅仅是数据



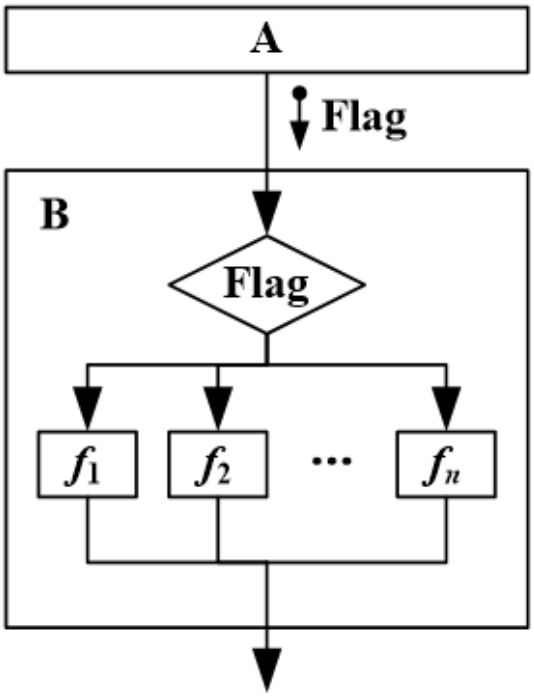
特征耦合

整个数据结构作为参数传递而被调用的模块只需要使用其中一部分数据元素

被调用的模块可以使用的数据多于它确实需要的数据，这将导致对数据的访问失去控制，从而给计算机犯罪提供了机会

控制耦合

参数交换信息，并且传递的信息中包含控制信息(这种控制信息可以以数据的形式出现)，则称它们是控制耦合。控制耦合是中等程度的耦合，它增加了系统的复杂程度。控制耦合往往是多余的，可用数据耦合代替它



外部耦合

都访问同一全局简单变量，而且不通过参数表传递该全局变量的信息，则称之为外部耦合。外部耦合和公共耦合很像，区别就是一个是简单变量，一个是复杂数据结构

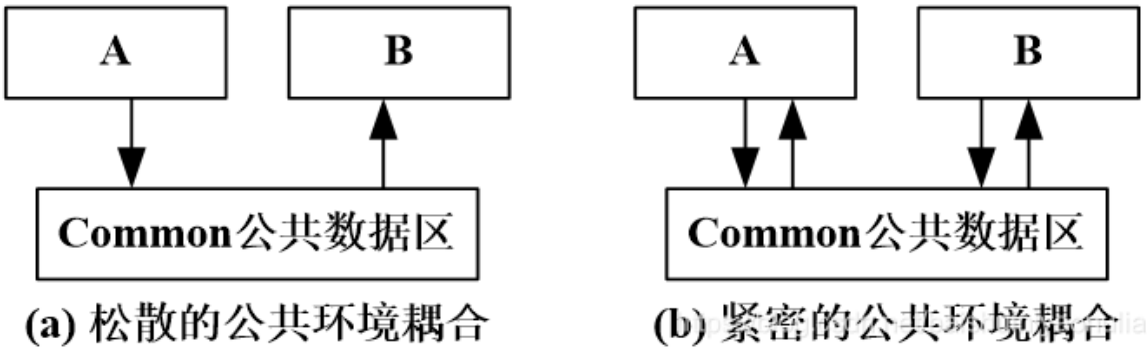
公共耦合

两个或多个模块通过一个公共数据环境相互作用,则称它们是公共环境耦合。公共环境耦合的复杂程度随耦合的模块个数增加而增加

公共环境耦合有两种类型:

(1)一个模块往公共环境送数据，另一个模块从公共环境取数据。数据耦合的一种形式，是比较松散的耦合

(2)两个模块都既往公共环境送数据又从里面取数据，这种耦合比较紧密，介于数据耦合和控制耦合之间

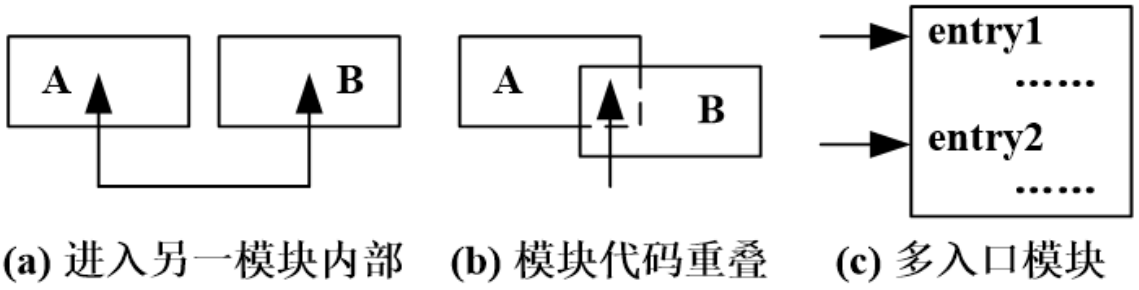


内容耦合

最高程度的耦合

内容耦合情况:

- 一个模块直接访问另一模块的内容
- 一个模块不通过正常入口而转到另一个模块的内部
- 两个模块有一部分程序代码重叠(只可能出现在汇编程序中)
- 一个模块有多个入口(这意味着一个模块有几种功能)



内聚

用来度量一个模块内部各个元素彼此结合的紧密程度。内聚度越高，紧密程度越高

内聚和耦合是密切相关的，模块内的高内聚往往意味着模块间的松耦合。实践表明内聚更重要，应该把更多注意力集中到提高模块的内聚程度上

内聚分为三大类低内聚、中内聚和高内聚（了解）

内聚度从低到高

- 偶然内聚
- 逻辑内聚
- 时间内聚

- 过程内聚
- 通信内聚
- 顺序内聚
- 功能内聚

（低内聚）

偶然内聚

即使有关系，关系也是很松散的

逻辑内聚

逻辑上属于相同或相似的一类

时间内聚

任务必须在同一段时间内执行

（中内聚）

过程内聚

处理元素是相关的，而且必须以特定次序执行

通信内聚

元素都使用同一个输入数据和(或)产生同一个输出数据

（高内聚）

顺序内聚

处理元素和同一个功能密切相关，而且这些处理必须顺序执行

功能内聚

所有处理元素属于一个整体，完成一个单一的功能

3 启发规则

启发规则有

- 改进软件结构提高模块独立性
- 模块规模应该适中

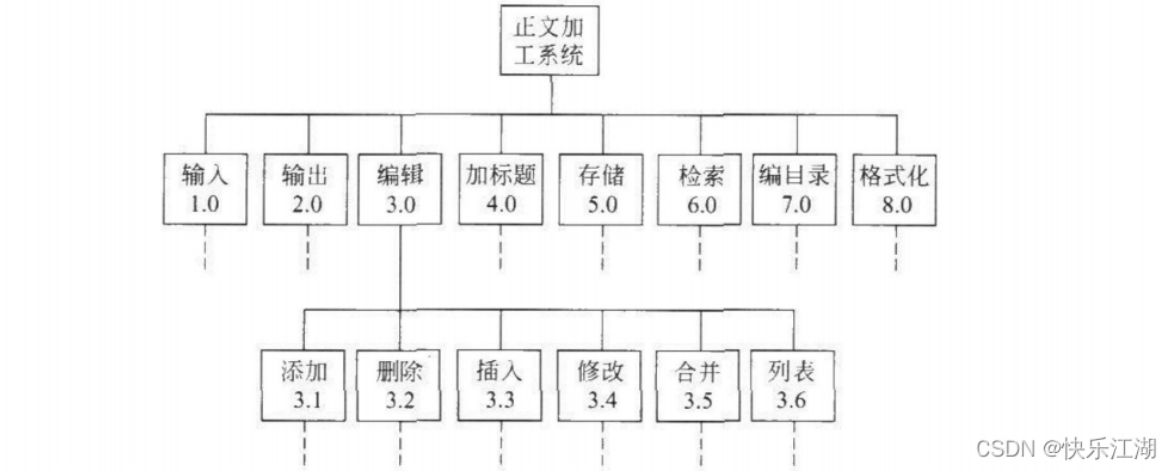
- 深度、宽度、扇入和扇出应适当
- 模块的作用域应该在控制域之内
- 力争降低模块接口的复杂程度
- 设计单入口单出口的模块
- 模块功能应该可以预测但要防止过分局限

4 描绘软件结构的图形工具

层次图和HIPO图

一个矩形框代表一个模块，方框间的连线表示调用关系而不像层次方框图那样表示组成关系

HIPO图本质就是层次图加编号

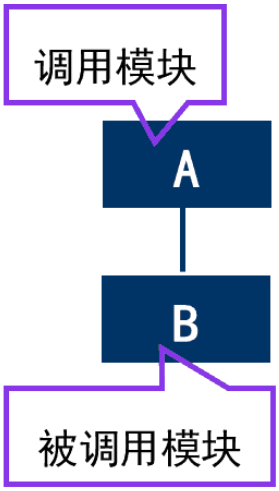


结构图

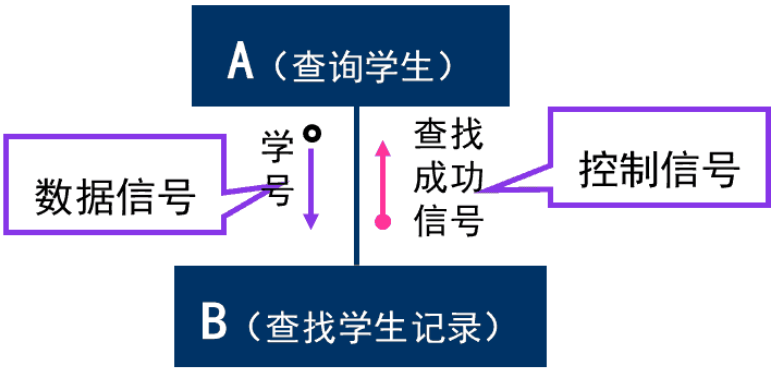
结构图不仅描述调用关系，还描述**传递的信息和调用方式**

基本符号

- 方框代表模块、框内注明模块的名字或主要功能
- 箭头或直线表示调用关系
- 尾部是空心圆表示传递的是数据；若是实心圆则表示传递的是控制信息

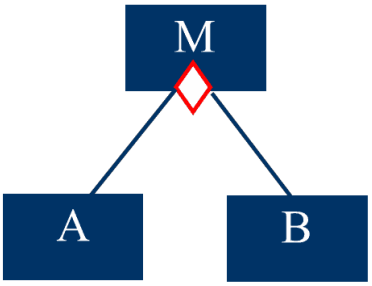


-- 模块调用关系

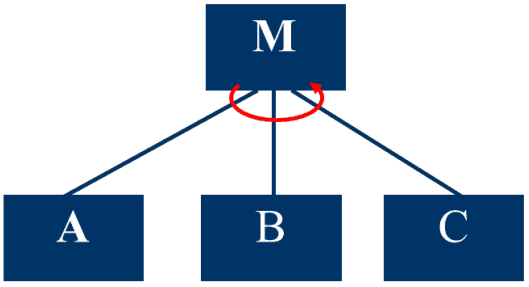


-- 模块间接口的表示

特殊符号



模块M有选择地调用模块A与B。判定为真调用A，为假调用B



模块M循环调用模块A、B、C。

5详细设计

1 结构程序设计

如果一个程序的代码块仅仅通过**顺序、选择和循环**这3种基本控制结构进行连接，并且每个代码块只有**一个入口和一个出口**，则称这个程序是结构化的。

类型

- 只允许使用顺序、IF-THEN-ELSE型分支和DO-WHILE型循环这3种基本控制结构，则称为 **经典的结构程序设计**
- 还允许使用DO-CASE型多分支结构和DO-UNTIL型循环结构，则称为 **扩展的结构程序设计**
- 再允许使用LEAVE(或BREAK)结构，则称为 **修正的结构程序设计**

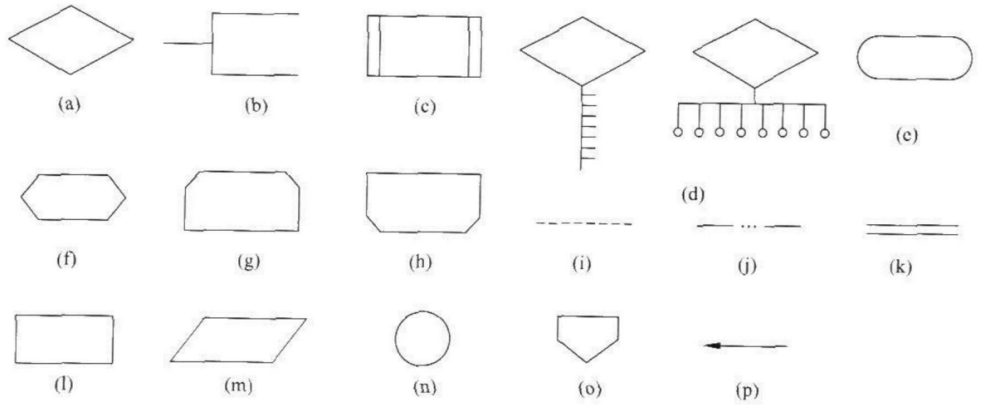
3 过程设计的工具

- 程序流程图
- 盒图（N-S图）
- PAD图
- 判定表
- 判定树
- 过程设计语言（PDL）

程序流程图

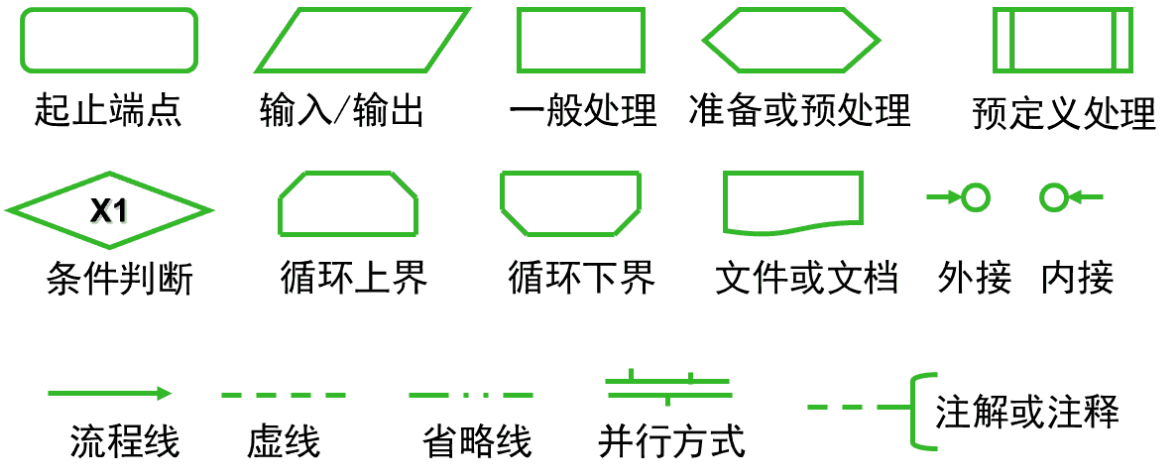
又称为程序框图，是历史最悠久，使用最广泛的描述过程设计的方法，然而它也是用得最混乱的一种方法

程序设计流图标准化图符



- a：**选择**
- b：注释
- c：预先定义的处理
- d：多分支
- e：**开始或停止**
- f：准备
- g：循环上界限
- h：循环下界限
- i：虚线
- j：省略符

- k：并行方式
- l：**处理**
- m：**输入输出**
- n：连接
- o：换页连接
- p：**控制流**



评价

优点：

对控制流程描绘直观，便于初学者掌握

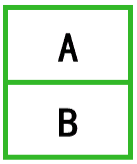
缺点：

1. 不是逐步求精的好工具，过早考虑控制流程，非整体结构
2. 用箭头代表控制流，程序员随意转移控制
3. 不易表示数据结构和调用关系

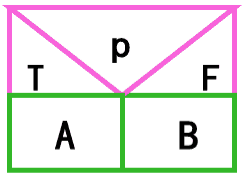
盒图（N-S）

出于要有一种不允许违背**结构程序设计精神**的图形工具的考虑，提出了盒图，又称为N-S图

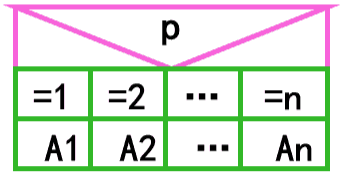
子程序大概不画



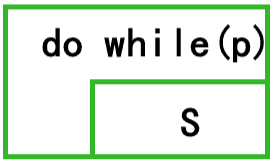
顺序型



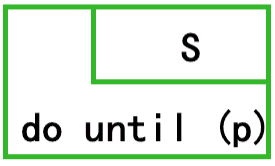
选择型



多分支选择型



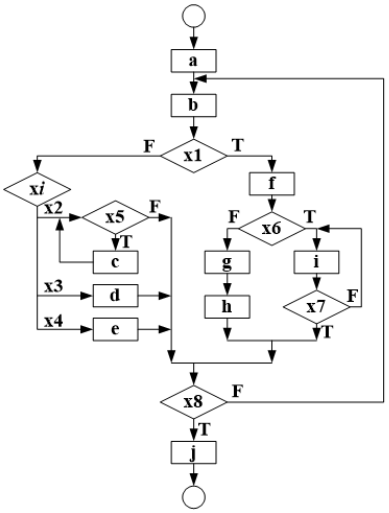
当型循环型

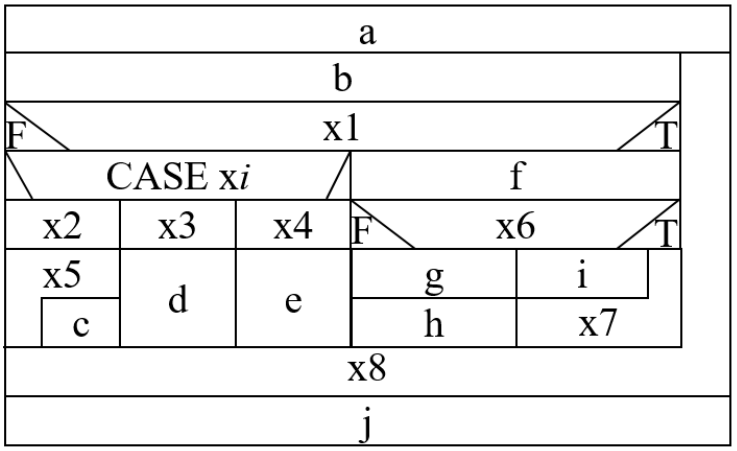


直到型循环型

调用子程序

示例





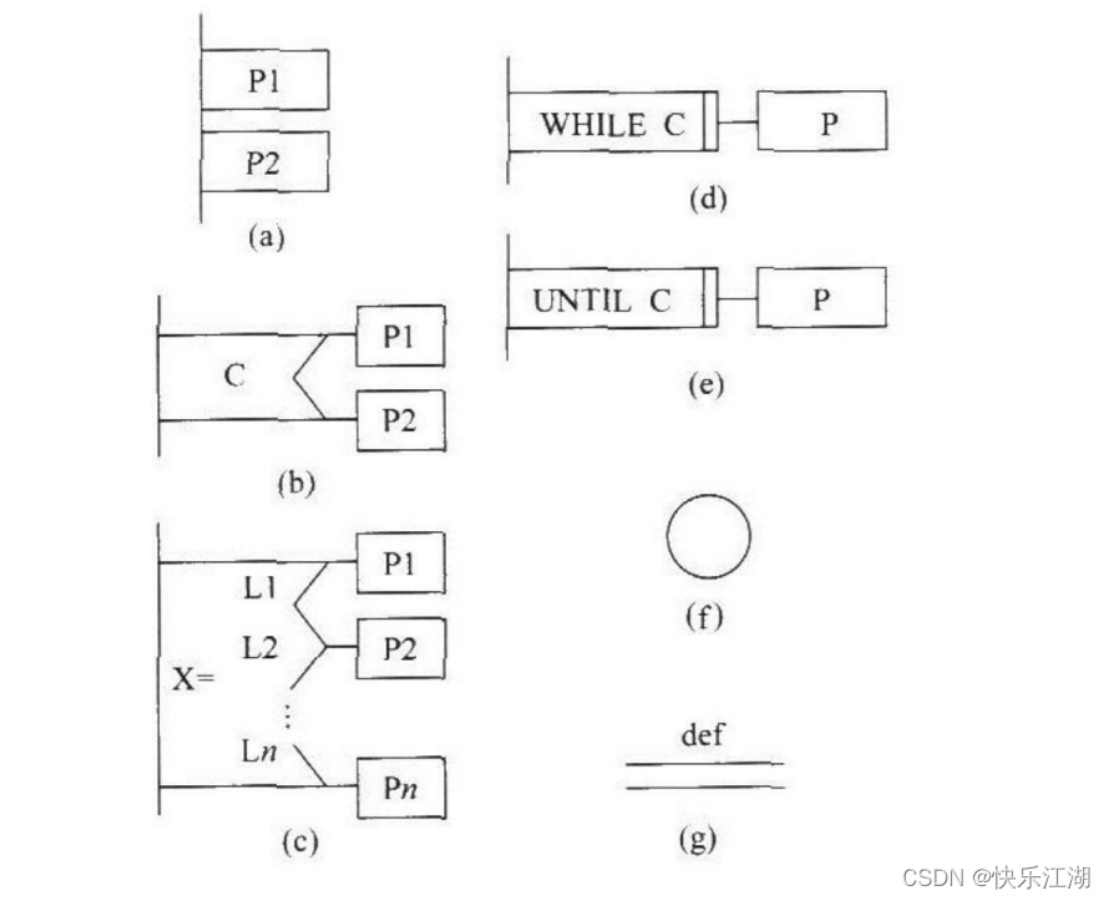
特点

1. 功能域（特定控制结构的作用域）明确
2. 不可能任意转移控制
3. 容易确定局部和全程数据的作用域

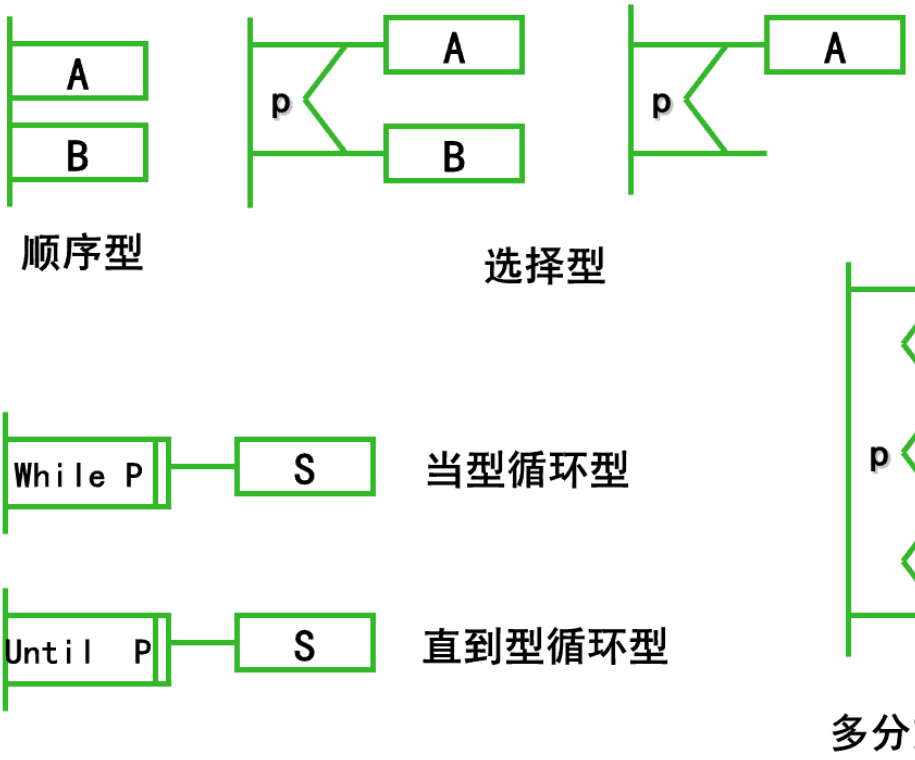
4. 容易表现嵌套关系，也可表示模块的层次结构

PAD图

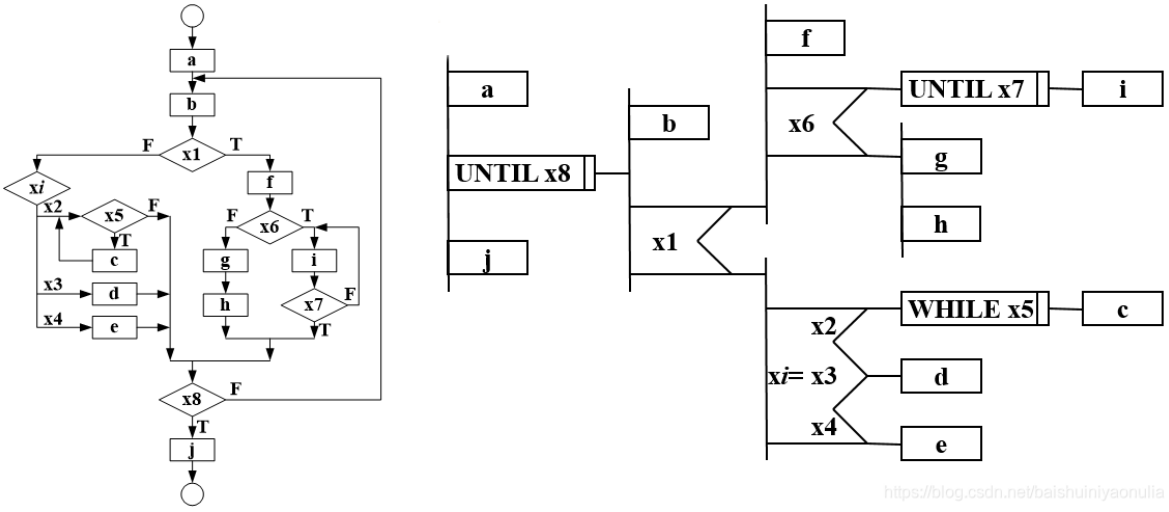
问题分析图（problem analysis is diagram）的英文缩写，是使用二维树形结构的图来表示程序的控制流，这种图翻译为程序代码比较容易



- a：顺序
- b：选择（`IF C THEN P1 ELSE P2`）
- c：`CASE`型多分支
- d：`WHILE`型循环（`WHILE C DO P`）
- e：`UNTIL`型循环（`REPEAT P UNTIL C`）
- f：语句符号
- g：定义



示例



评价

优点

1. 使用PAD图设计的程序必然是结构化程序
2. PAD图描绘的程序结构十分清晰

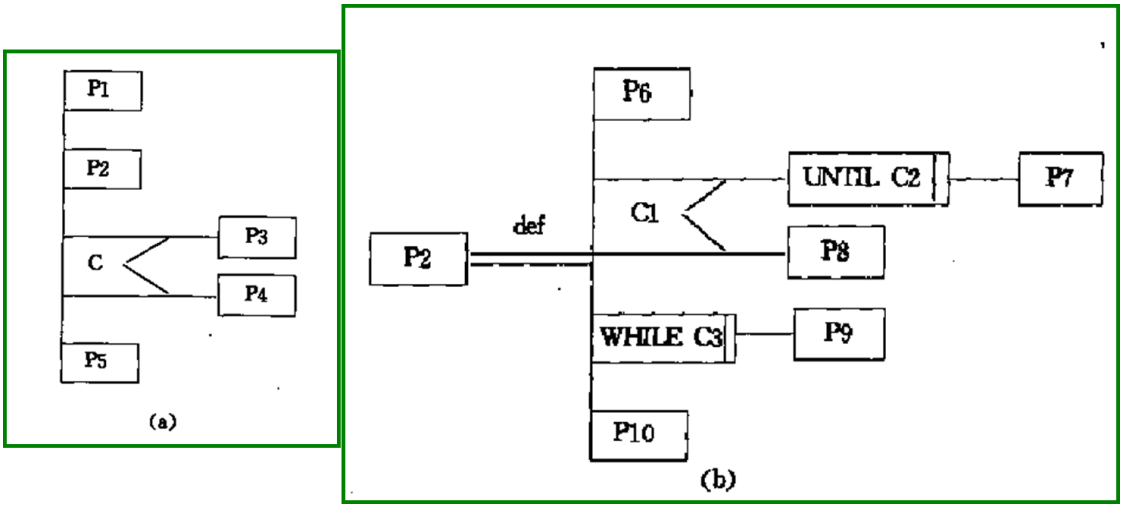
> 最左面的竖线是**程序的主线**，即**第一层结构**。随着程序层次的增加，PAD图逐渐**向右延伸**，每增加一个层次，图形向右扩展一条竖线。PAD图中竖线的总条数就是**程序的层次数**

3. 用PAD图表现程序逻辑，易读、易懂、易记

   > PAD图是二维树形结构的图形，程序从图中最左竖线上端的结点开始执行 ，**自上而下，从左向右**顺序执行，遍历所有结点

4. 容易将PAD图转换成高级语言源程序
5. 既可用于表示程序逻辑，也可用于描绘数据结构
6. 支持自顶向下逐步求精

   > 开始时设计者可以定义一个抽象的程序，随着设计工作的深入而使用`def`符号逐步增加细节，直至完成详细设计



判定表

当算法中包含**多重嵌套的条件选择**时，用程序流程图、盒图、PAD图或后面即将介绍的过程设计语言(PDL)都不易清楚地描述

能清晰表示复杂的条件组合与应做动作间对应关系

四部分：

- 左上部列出所有条件；
- 左下部所有可能做的动作；
- 右上部表示各种条件组合的矩阵；
- 右下部是和每种条件组合相对应的动作。

判定表右半部的每一列实质上是一条规则，规定了与特定的条件组合相对应的动作

右上部分中**T表示它左边那个条件成立，F表示条件不成立，空白表示这个条件成立与否并不影响对动作的选择。**判定表右下部分中**画X表示做它左边的那项动作，空白表示不做这项动作**

		Rules								
	Rule numbers →	1	2	3	4	5	6	7	8	9
Condition rows	国内乘客		T	T	T	T	F	F	F	F
	头等舱		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 ≤ 30	T	F	F	F	F	F	F	F	F
Action rows	免费	×								
	(W-30) × 2				×					
	(W-30) × 3					×				
	(W-30) × 4		×						×	
	(W-30) × 6			×						×
	(W-30) × 8						×			
	(W-30) × 12							×		

评价

优点

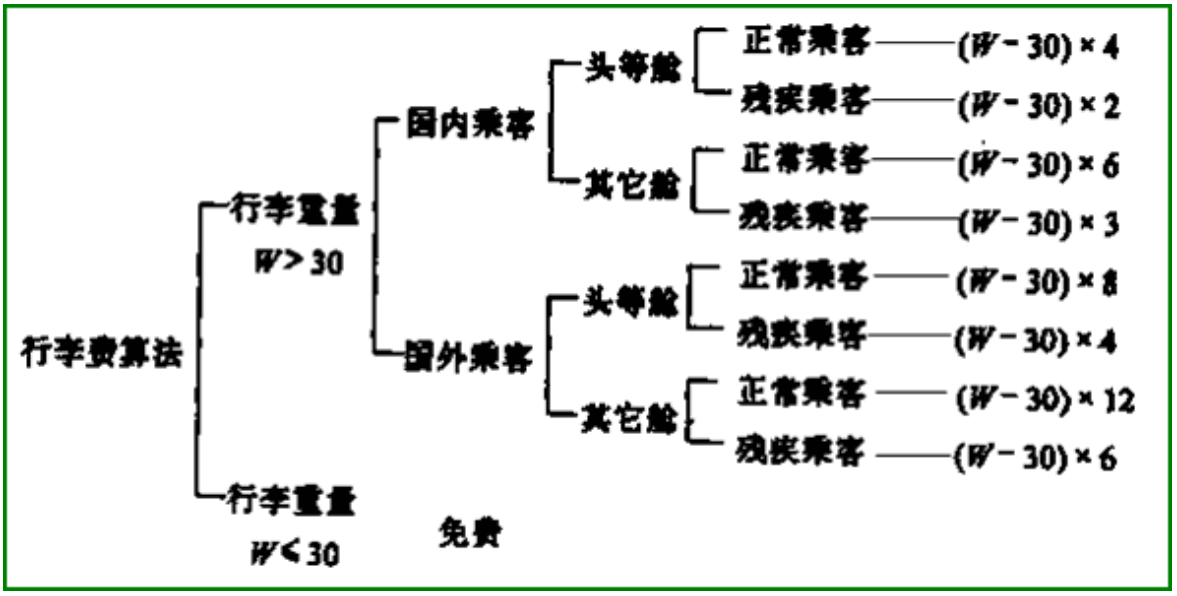
- 判定表能够简洁而又无歧义地描述处理**规则**
- 判定表和布尔代数或卡诺图结合起来使用，可以更加直观、简洁、清晰的描述规则

缺点

- 不能同时清晰地表示出问题的**顺序性**和**重复性**
- 初次接触这种工具的人理解它需要有一个学习过程
- 数据元素增多时，判定表的简洁程度大幅下降，此时建议使用判定树

判定树

判定表变种，表示复杂条件组合与应做动作间对应关系



评价

优点：形式简单，易看出含义，易于掌握和使用。

缺点：简洁性不如判定表，相同数据元素重复写多遍，越接近叶端重复次数越多。

4 面向数据结构的设计方法

Jackson法

面向数据结构的设计方法

Jackson方法是在软件开发过程中常用的方法，使用Jackson方法时可以实现从数据结构导出程序结构

5 程序复杂程度的定量度量

McCabe方法

McCabe方法根据程序控制流的复杂程度定量度量程序的复杂程度，这样度量出的结果称为程序的环形复杂度

1. 根据过程设计结果画出相应流图
2. 计算流图的环形复杂度

流图

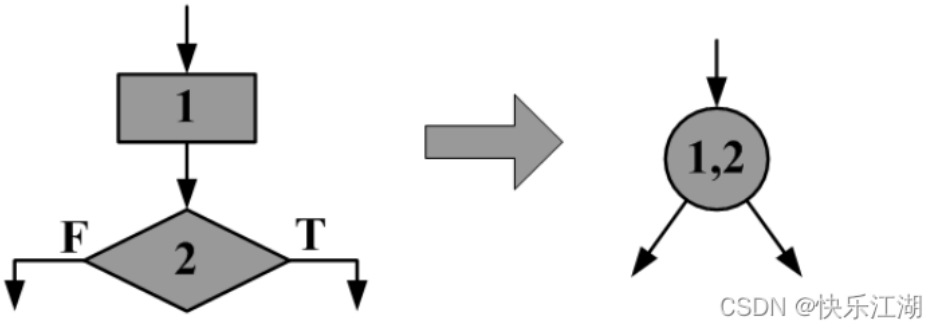
流图实质上是“退化了的”程序流程图，描绘程序的控制流程，不表现对数据的具体操作以及分支或循环的具体条件

流图描述程序控制流，基本图形符号如下

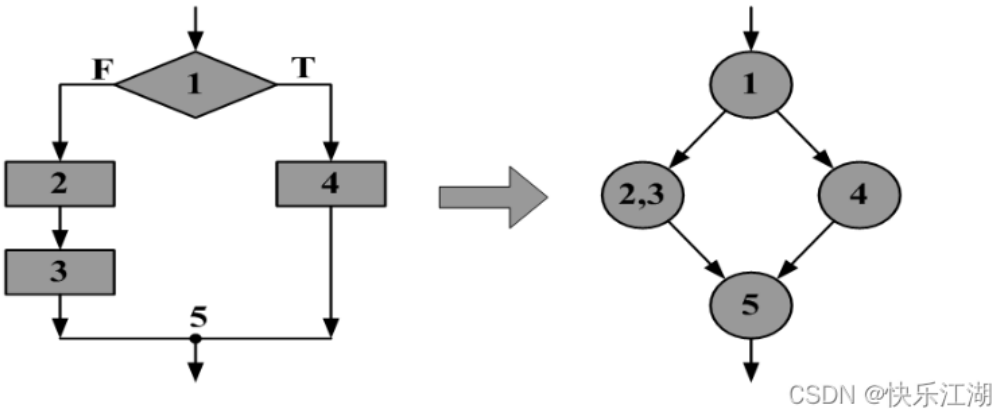
绘制方法

- 结点：一个圆代表一条或多条语句，一个顺序结构可以合并一个结点
- 边：流图中的箭头线称为边，代表控制流，流图中一条边必须终止于一个结点
- 区域：由边和结点围成的面积称为区域，包括图外部未被围起来的区域

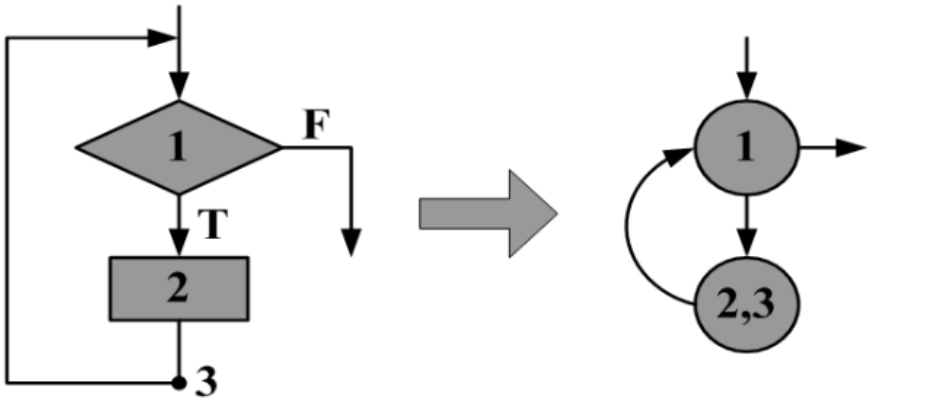
1. **对于顺序结构，一个顺序处理和下一个选择可以映射为一个结点**



2. **对于选择语句，开始/结束语句映射为一个结点，两条分支至少各映射成一个结点**

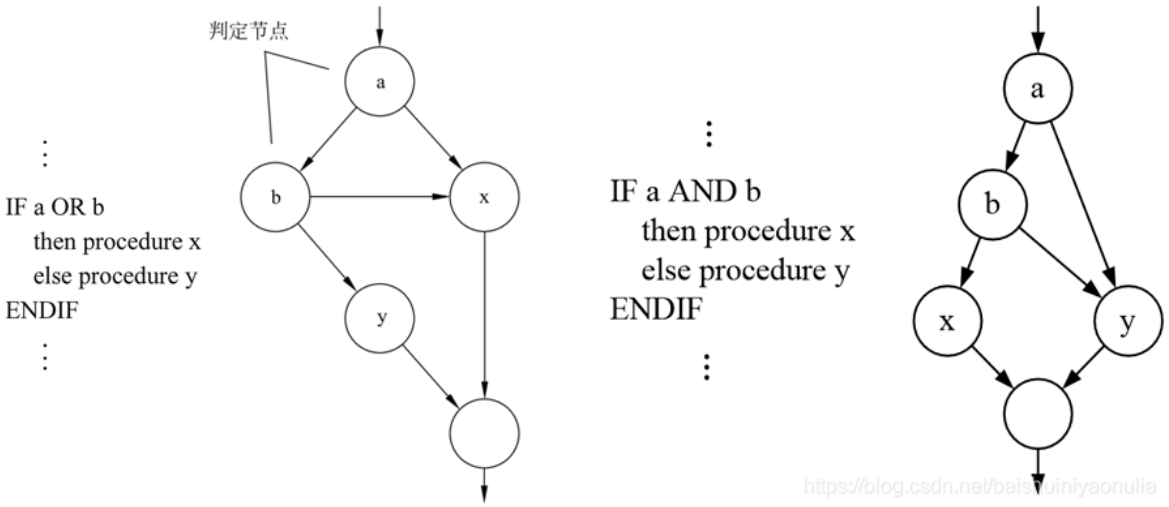


3. **开始语句和结束语句各映射成一个结点**

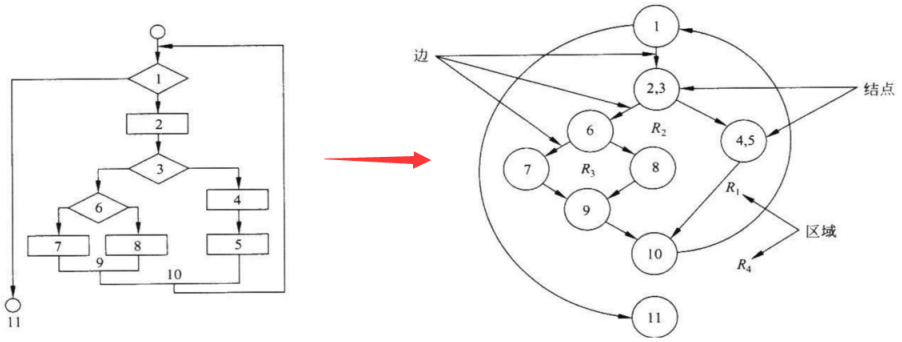


包含复合条件时，应该把复合条件分解为若干个简单条件，每个简单条件对应流图中一个结点。所谓复合条件，就是在条件中包含了一个或多个布尔运算符(逻辑OR，AND，NAND，NOR)

> 这里就是 a 和 b 条件都要独立判定一次了，之后要保留指向对应结果不同情况对象



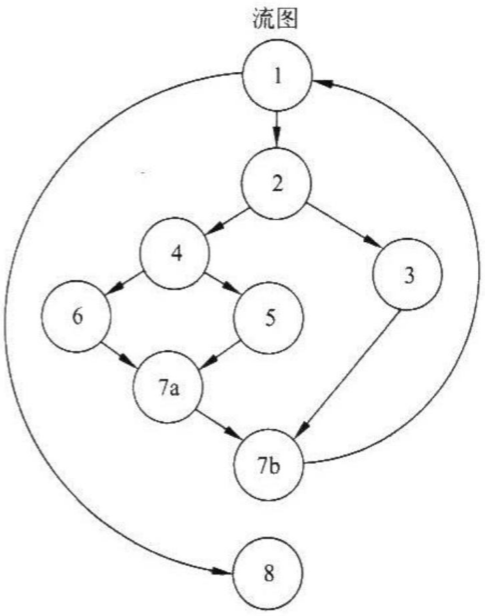
翻译



- 流图中用**圆表示结点，一个圆代表一条或多条语句（比如4、5）。**程序流程图中的**一个顺序的处理框序列和一个菱形判定框，可以映射成流图中的一个结点（比如2,3）**
- 流图中的箭头线称为**边**，代表**控制流。流图中一条边必须终止于一个结点（比如9必须为一个结点）**，即使这个结点并不代表任何语句
- 由边和结点围成的面积称为**区域**，计算区域数时应**包括图外部未被围起来的区域**

示例：由PDL翻译成的流图

```
PDL
procedure:sort
1:   do while records remain
2:      read record;
        if record field 1=0
3:        then process record;
            store in buffer;
            increment counter;
4:      elseif record field 2=0
5:        then reset counter;
6:        else process record;
            store in file;
7a:     endif
        endif
7b:  enddo
8:   end
```



环形复杂度

环形复杂度定量度量程序的逻辑复杂度，是对测试难度的一种定量度量，也能对软件最终的可靠性给出某种预测

V(G)小于等于10比较科学

- 流图中的区域数等于环形复杂度

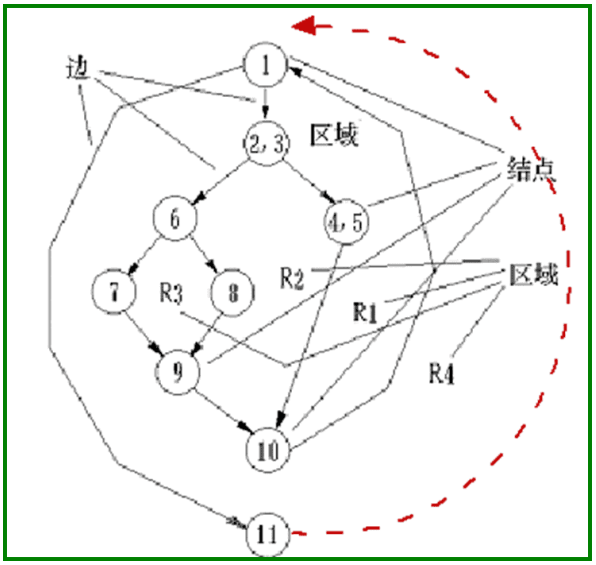
  > 区域就想象是PS的填充桶工具，必须要是封闭的区域

- E-N+2，E是流图中边的条数，N是结点数
- P+1，P是流图中判定结点(分支选择的节点)的数目

三种方法：

- V（G）＝区域数
- V（G）＝E-N＋2
  E为流图中边数，N为流图中节点数
- V（G）＝P＋1　　P为判定点数

示例



V(G)＝4 （区域数）

V(G)＝11(边数)－9(结点数)＋2＝4

V(G)＝3(判定结点数)＋1＝4

6实现

2 软件测试基础

1. 测试是为了**发现程序中的错误**而执行程序的过程；
2. 好的测试方案是极有可能发现迄今尚未**发现的尽可能多的错误**的测试；
3. 成功的测试是发现了迄今**尚未发现的错误**的测试。

测试准则

1. 所有测试应能追溯到用户需求
2. 应尽早地和不断地进行软件测试
3. 充分注意测试中群集现象(二八法则) **80%的错误是由20%的模块造成的**
4. 测试应从小规模开始，逐步进行大规模测试
5. 不能做到穷举测试，**测试只能证明程序有错误，而不能证明程序没有错误**
6. 第三方测试原则

测试方法

静态测试方法和动态测试方法

- 静态测试
  - 不在机器上进行测试，而是采用人工检测和计算机辅助静态分析手段对程序进行检测
    - 人工测试 人工审查和评审软件
    - 计算机辅助静态分析

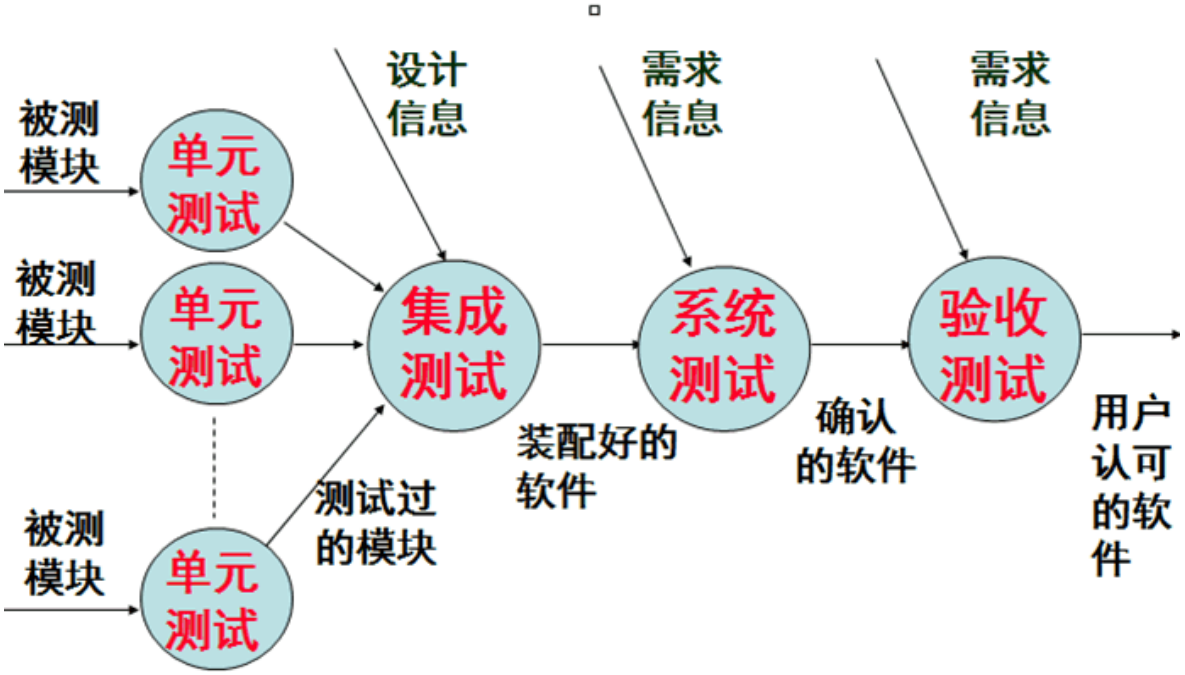
- 动态测试
  - 跑起来发现错误
  - 黑盒测试和白盒测试
    - 黑盒测试：如果知道产品应具有**功能**，可通过测试来检验是否每个功能都能正常使
    - 白盒测试：如果知道产品**内部工作过程**可通过测试来检验产品内部动作是否按照规格说明书的规定正常进行

测试步骤

测试过程也必须分步骤进行，后一个步骤在逻辑上是前一个步骤的继续

大型软件系统通常由若干个子系统组成，每个子系统又由许多模块组成，因此，大型软件系统的测试过程基本上由 模块测试、子系统测试、系统测试、验收测试和平行运行等 五个步骤组成

测试步骤示意



测试阶段	主要依据	测试人员	测试方法	测试内容
单元测试	系统设计文档	开发小组	白盒测试	接口测试 路径测试
子系统测试	系统设计文档 需求文档	独立测试小组	白盒测试 黑盒测试	接口测试 路径测试 功能测试 性能测试
系统测试	需求文档	独立测试小组	黑盒测试	功能测试、健壮性测试 性能测试、用户界面测试 安全性测试、压力测试 可靠性测试、安装/卸载测试
验收测试	需求文档	用户	黑盒测试	功能测试、健壮性测试 性能测试、用户界面测试

测试阶段	主要依据	测试人员	测试方法	测试内容
				安全性测试、压力测试 可靠性测试、安装/卸载测试

3 单元测试

单元测试集中检测软件设计的最小单元一模块，它和编码属于软件过程的同一个阶段。在编写出源程序代码并通过了编译程序的语法检查之后，就可以用详细设计描述作指南，对重要的执行通路进行测试，以便发现模块内部的错误。单元测试主要使用白盒测试技术，而且对多个模块的测试可以并行地进行，包括人工测试和计算机测试两种

- **测试依据**：详细设计文档
- **测试技术**：白盒测试技术
- **测试方法**：人工测试和计算机测试

4 集成测试

测试和组装软件的系统化技术

概念

不同集成测试策略的比较与回归测试

集成测试策略	优点	缺点
非渐增式	无	没有错误隔离手段 主要设计错误发现迟 潜在可重用代码测试不充分 需要驱动程序和存根程序
渐增测试自顶向下	具有错误隔离手段 主要设计错误发现早 不需要驱动程序	潜在可重用代码测试不充分 需要存根程序
渐增测试自底向上	具有错误隔离手段 潜在可重用代码能充分测试 不需要存根程序	主要设计错误发现迟 需要驱动程序
渐增测试混合	具有错误隔离手段 主要设计错误发现早 潜在可重用代码能充分测试	较少

由模块组装成程序时有两种方法：

非渐增式测试策略

先分别测试每个模块，再把所有模块按设计要求放在一起结合成所要的程序

> 像啊，很像啊（指平时自己写代码，每个模块都work well，啪一下合并起来就down了）

- 把所有模块放在一起，测试者面对的情况十分复杂
- 在庞大的程序中诊断定位一个错误非常困难

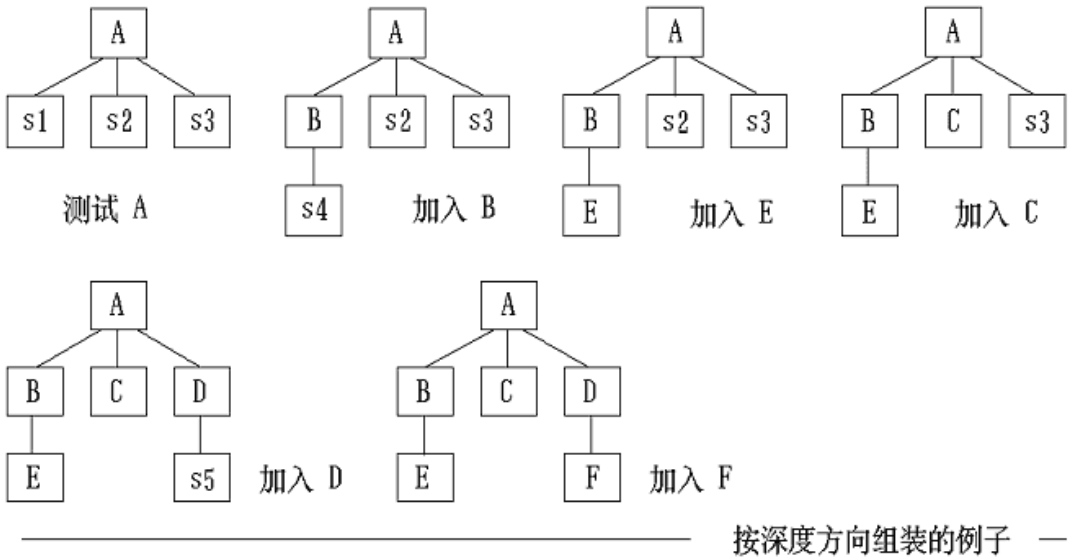
- 一旦改正一个错误之后，又会遇到新的错误，没有穷尽

渐增式测试策略

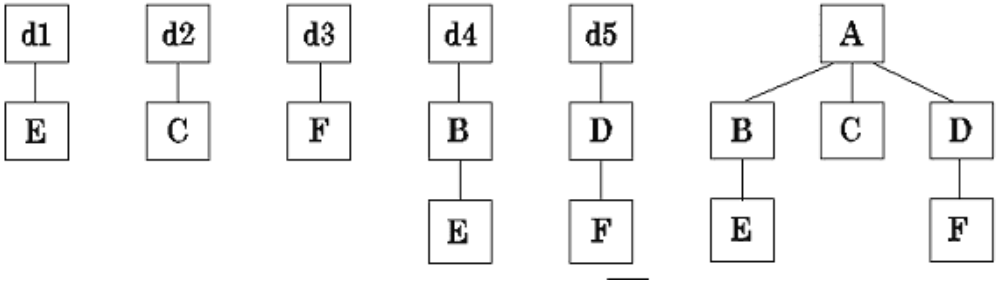
使用渐增方式把模块结合到程序中去时的集成策略

将模块逐步组装成较大系统

1. 自顶向下集成



2. 自底向上集成



3. 混合策略

   - 改进的自顶向下测试方法

     基本用自顶向下方法，早期用自底向上测试关键模块

   - 混合法

     软件结构上层模块用自顶向下，下层用自底向上。

自顶向下集成

从主控制模块开始，沿着程序的控制层次向下移动，逐渐把各个模块结合起来。在把附属于主控制模块的模块组装到程序结构中时，使用深度优先的策略或宽度优先的策略

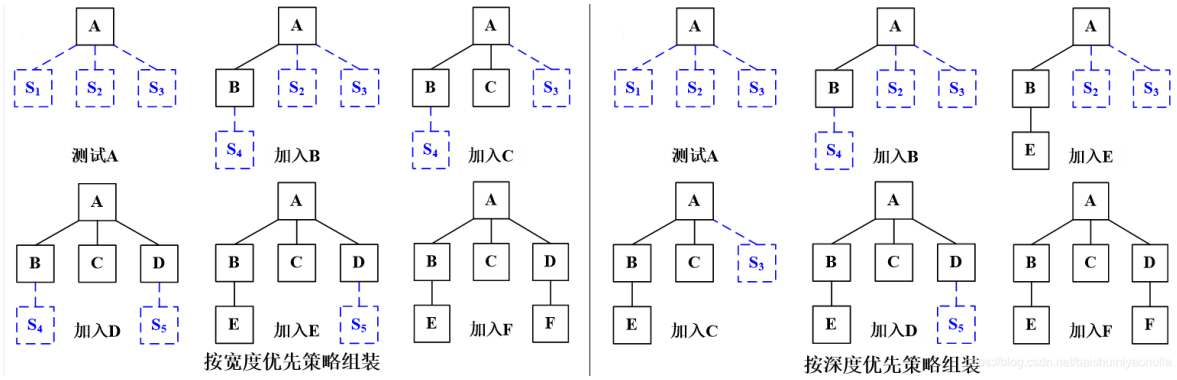
步骤

- 对主控制模块进行测试，测试时用**存根程序**代替所有直接附属于主控制模块的模块
- 根据选的结合策略（深度优先或宽度优先），每次用一个**实际模块**代换一个存根程序
- 在结合进一个模块的同时进行测试
- 为了保证加入模块没有引进新的错误，可能需要进行**回归测试**

结合策略

深度优先先组装在**软件结构的一条主控制通路上的所有模块**

宽度优先**沿软件结构水平地移动，把处于同一个控制层次上的所有模块组装起来**

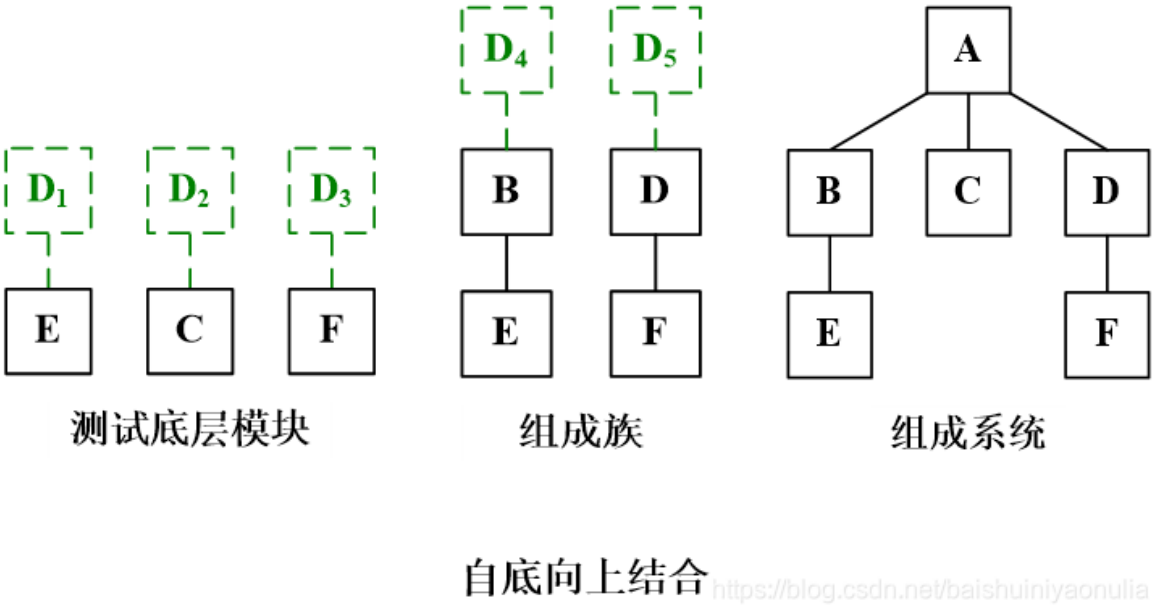


自底向上集成

自底向上测试从软件结构最低层的模块开始组装和测试。因为是从底部向上结合模块，总能得到所需的下层模块处理功能，所以不需要存根程序

步骤

- 把低层模块组合成实现某个特定的软件子功能的**族**
- 写一个用于测试的控制程序，协调测试数据的输入和输出
- 对由模块组成的子功能族进行测试
- 去掉驱动程序，沿软件结构自下向上移动，把子功能族组合起来形成大的子功能族



混合

混合集成测试策略，主要有两种

1. **改进的自顶向下测试方法**
   基本上使用自顶向下的测试方法，但是在早期使用自底向上的方法测试软件中的少数关键模块。该策略能在测试的早期发现关键模块中的错误；测试关键模块时需要驱动程序。
2. **混合法**
   对软件结构中较上层使用的自顶向下方法与对软件结构中较下层使用的自底向上方法相结合，该策略兼有两种方法的优缺点，当被测试的软件中关键模块比较多时，这种混合法可能是最好的折衷方法。

回归测试

回归测试是指**重新执行已经做过的测试的某个子集，以保证上述这些变化没有带来非预期的副作用。**它可以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

5 确认测试

也称为验收测试，它的目标是验证软件的有效性

确认测试必须有用户积极参与，或者以用户为主进行，使用用户界面输入测试数据并且分析评价测试的输出结果，在验收之前通常要由开发单位对用户进行培训，一般来说确认测试分为Alpha和Beta测试

Alpha测试 和 Beta测试

Alpha测试

Alpha测试由用户在开发者的场所进行，并且在开发者对用户的“指导”下进行测试，且开发者负责记录发现的错误和遇到的问题。即Alpha测试是在受控的环境中进行的

（内部实机演示）

Beta测试

Beta测试由软件的最终用户们在一个或多个客户场所进行。开发者通常不在Beta测试的现场，即Beta测试是软件在开发者不能控制的环境中的“真实”应用

（上线内测）

6 白盒测试技术

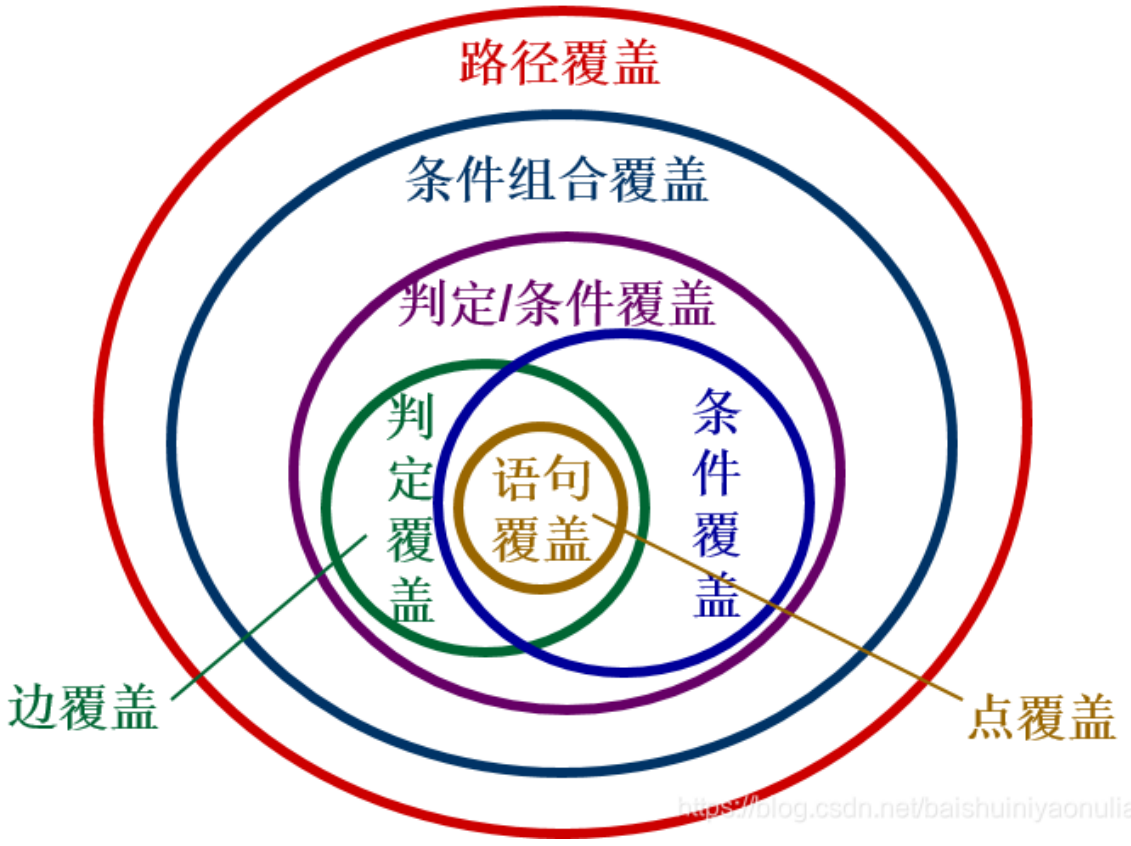
通常把测试数据和预期的输出结果称为测试用例

逻辑覆盖

逻辑覆盖是对一系列测试过程的总称，这组测试过程逐渐进行越来越完整的通路测试

下面是分类

包含关系：



语句覆盖

选择足够多的测试数据，被测试程序中的每条语句至少执行一次

很弱

点覆盖

连通图G的子图G'是连通的，而且包含G的所有结点，则称G'是G的点覆盖。

满足点覆盖标准要求选取足够多的测试数据，使得程序执行路径至少经过流图的每个结点一次，也即**点覆盖标准和语句覆盖标准是相同的**

判定覆盖

又叫分支覆盖

不仅每个语句至少执行一次，而且**每个判定**的每种可能的结果都应该至少执行一次

> 强调整个表达式的结果，分支往哪里走

比语句覆盖强，但是对程序逻辑的覆盖程度仍然不高

边覆盖

连通图G的子图G''是连通的，而且包含G的所有边，则称G''是G的边覆盖。为满足边覆盖的测试标准，要求选取足够多的测试数据，使程序执行路径至少经过流图每条边一次，也即**边覆盖与判定覆盖是相同的**

条件覆盖

不仅每个语句至少执行一次，而且使判定表达式中的**每个条件**都取到各种可能的结果

> 强调每个表达式的细节组成条件，不一定比判定覆盖强；这里的细节要联动数学中的取反的情况，全真的反应该是部分假和全假!

条件覆盖通常比判定覆盖强，因为它使每个条件都取到了两个不同的结果，判定覆盖却只关心整个判定表达式的值

判定覆盖不一定包含条件覆盖，条件覆盖也不一定包含判定覆盖

判定/条件覆盖

选择足够多的测试数据，使 判定表达式中的**每个条件**都取到各种可能的结果，而且**每个判定表达式**也都取到各种可能的结果。

> 表达式的走向和每个表达式内部的条件都被考虑到了，但是没有组合起来

同时满足判断覆盖和条件覆盖

条件组合覆盖

选取足够多的测试数据，使得每个**判定表达式**中**条件**的各种**可能组合**都至少出现一次。

> 次强的，不一定使每条路径都执行一次

满足条件组合覆盖，也一定满足判断覆盖、条件覆盖和判断/条件覆盖

路径覆盖

选取足够多的测试数据，使程序的每条可能路径都至少执行一次，如果程序图中有环，则要求每个环至少经过一次

控制结构测试

基本路径测试

Tom McCabe提出的一种白盒测试技术

使用这种技术设计测试用例时，首先计算程序的**环形复杂度**，**并用该复杂度为指南，定义执行路径的基本集合，从该基本集合导出的测试用例可以保证程序中的每条语句至少执行一次，而且每个条件在执行时都将分别取真、假两种值**

步骤

- 根据过程设计结果画出相应的**流图**
- 计算流图的**环形复杂度**
- 确定**线性独立路径**(至少包含一条在定义该路径之前不曾用过的边)的基本集合
- 设计可**强制执行基本集合中每条路径**的测试用例

1. 根据过程设计结果画出相应流图
2. 计算流图的环形复杂度

   > 见之前

3. 确定线性独立路径的基本集合
   - 独立路径：至少包含一条在定义改路径之前不曾用过的边。
   - 环形复杂度为独立路径基本集的上界

     > 程序的环形复杂度为6，因此共有6条独立路径
4. 设计测试用例覆盖基本集合的路径的测试用例

条件测试

关系表达式

一个简单条件是一个布尔变量或一个关系表达式，在布尔变量或关系表达式之前还可能有一个NOT(¬)运算符，关系表达式的形式如下：

```
E1<关系算符>E2
```

E1和E2是算术表达式，而<关系算符>是下列算符之一：<，≤，=，≠，>或≥

布尔算符有OR(|)，AND(&)和NOT(┐)

不包含关系表达式的条件称为布尔表达式

条件错误的类型

- 布尔算符错;
- 布尔变量错;
- 布尔括弧错;
- 关系算符错;
- 算术表达式错

评价

- 容易度量条件的测试覆盖率
- 程序内条件的测试覆盖率可指导附加测试的设计

循环测试

循环测试是一种白盒测试技术，它专注于测试循环结构的有效性。在结构化的程序中通常只有3种循环，即简单循环、串接循环和嵌套循环

7 黑盒测试技术

概念

黑盒着重**软件功能**

黑盒测试并不能取代白盒测试，它是与白盒测试互补的测试方法，它很可能发现白盒测试不易发现的其他类型的错误

通常设计测试方案时总是联合使用等价划分和边界值分析两种技术

公认标准主要有两个：（1）测试用例尽可能少；（2）一个测试用例能指出一类错误

目的

发现错误:

1. 功能不正确或遗漏
2. 界面错误
3. 数据结构或外部数据库访问错误
4. 性能错误
5. 初始化或终止错误

适用性

白盒测试在测试过程的早期阶段进行，黑盒测试主要用于测试过程的后期

用例标准

- 能够减少为达到合理测试所需要设计的测试用例的**总数**
- 能够告诉人们，是否**存在某些类型的错误**，而不是仅仅指出与特定测试相关的错误是否存在

等价划分

把程序的输入域划分成若干数据类，从每一数据类选取少数有代表性数据做为测试用例

等价划分法力图设计出能发现**若干类**程序错误的测试用例，从而减少必须设计的测试用例的数目

1. 划分等价类
    1. 有效等价类：合理，有意义输入数据构成集合。
    2. 无效等价类：不合理，无意义输入数据构成的集合。
2. 确定测试用例

    建立等价类表，列出所有划分出等价类

    1. 为每一等价类规定一唯一编号；
    2. 设计一新测试用例，尽可能多覆盖尚未被覆盖有效等价类，重复，直到所有有效等价类被覆盖。
    3. 设计一新测试用例，仅覆盖一尚未被覆盖无效等价类，重复，直到所有无效等价类被覆盖

划分数据的等价类

- 第一，需要研究**程序的功能说明**，从而确定输入数据的**有效等价类**和**无效等价类**
- 第二，在确定输入数据的等价类时常常还需要**分析输出数据的等价类**
- 第三，在划分等价类时还应考虑**编译程序的检错功能**

根据等价类设计测试方案

- 第一，设计一个新的测试方案以尽可能多地覆盖**尚未被覆盖的有效等价类**，重复这一步骤直到**所有有效等价类都被覆盖为止**
- 第二，设计一个新的测试方案，使它**覆盖一个而且只覆盖一个尚未被覆盖的无效等价类**，重复这一步骤直到**所有无效等价类都被覆盖为止**

边界值分析

使用边界值分析方法设计测试方案首先应该确定边界情况，选取的数据应该**刚好等于**、**稍小于**和**稍大于**等价类边界值，即应该选取刚好等于、稍小于和稍大于等价类边界值的数据作为测试数据，而不是选取每个等价类内的典型值或任意值作为测试数据

错误推测

靠经验和直觉推测程序可能存在错误，有针对性编写检查这些错误的测试用例

9 软件可靠性

软件可靠性

程序在给定的时间间隔内，成功运行的概率

软件可用性

程序在给定的时间点，成功运行的概率

7维护

1 软件维护的概念

软件维护是在软件已经交付使用后，为了改正错误或满足新的需要而修改软件的过程，是软件生命周期的最后一个阶段，其基本任务是保证软件在一个相当长的时期能够正常运行

> 软件维护绝不仅限于纠正使用中发现的错误，事实上在全部维护活动中一半以上是完善性维护

改正性维护：诊断和改正错误的过程

适应性维护：为了和变化了的环境适当地配合而进行的修改软件的活动

完善性维护：为了满足用户提出的增加新功能或修改已有功能的要求和一般性改进要求

预防性维护：当为了改进未来的可维护性或可靠性，或为了给未来的改进奠定更好的基础而修改软件

2 软件维护的特点

结构化维护和非结构化维护差别巨大

- **非结构化维护**：唯一成分是程序代码，维护活动从艰苦地**评价程序代码**开始，需要付出很大代价
- **结构化维护**：有完整的软件配置存在，维护工作从**评价设计文档**开始

4 软件的可维护性

可维护性指的是维护人员**理解、改正、改动**或**改进**这个软件的难易程度。提高可维护性是支配软件工程方法学所有步骤的**关键目标**

决定软件可维护性的因素

修改之前必须理解待修改的对象，修改之后应该进行必要的测试

如下

可理解性

可测试性

可修改性

可移植性

可重用性

8-12特供

800引论

1 面向对象方法学概述

面向对象方法学的出发点和基本原则，**是尽可能模拟人类习惯的思维方式**，使开发软件的方法与过程尽可能**接近人类认识世界解决问题的方法与过程**，使描述问题的问题空间(问题域)与实现解法的解空间(求解域)在结构上尽可能一致

定义

面向对象方法学方程式

$$OO = 对象 + 类 + 继承 + 传递消息实现通信$$

要点

对象

面向对象的软件系统是由对象组成的，软件中的**任何元素都是对象**，复杂的软件对象由比较简单的对象组合而成。**用对象分解取代了传统方法的功能分解**，对象是从客观世界中的实体抽象而来的，是不固定的

类

把所有对象都划分成**各种对象类**,每个对象类都定义了**一组数据和一组方法。**数据用于表示对象的**静态属性**,是对象的状态信息。类中定义的方法，是允许施加于该类对象上的操作，是该类所有对象共享的，**并不需要为每个对象都复制操作的代码**

继承性

按子类与父类的关系，**把若干个对象类组成一个层次结构的系统。子类自动具有和上层的父类相同的数据和方法**，而且低层的特性将**屏蔽**高层的同名特性

封装性

对象彼此之间仅能通过传递消息互相联系。对象是进行处理的主体，必须发消息请求它执行它的某个操作，处理它的私有数据，而不能从外界直接对它的私有数据进行操作。一切局部于该对象的私有信息，都被封装在该对象类的定义中，就好像装在一个不透明的黑盒子中一样，在外界是看不见的，更不能直接使用

2 面向对象的概念

1. 对象：具有相同状态的一组操作的集合，对状态和操作的封装。
2. 类

   对具有相同状态和相同操作的一组相似对象的定义。

   类是一个抽象数据类型。
3. 实例

   实例是由某个特定类所描述的一个具体对象。
4. 消息

   要求某对象执行某个操作的规格说明。

   三部分：

   - 接受消息的对象
   - 消息名
   - 0或多个变元
5. 方法和属性

   - 方法：对象执行的操作，即类中定义的服务。
   - 属性：类中所定义数据，对客观世界实体具体性质的抽象。
6. 继承

   子类自动共享基类中定义的属性和方法的机制。
7. 多态性

在类等级不同层次可共享一个方法名，不同层次每个类按各自需要实现这个方法。

- 优点：
  - 提高程序可复用性（接口设计的复用，不是代码实现的复用）
  - 派生类的功能可被基类指针引用，提高程序可扩充性和可维护性。

8. 重载
   1. 函数重载

      在同一作用域内，参数特征不同的函数可使用相同的名字。

      - 优点

        调用者不需记住功能雷同函数名，方便用户；

        程序易于阅读和理解。

   2. 运算符重载

      同一运算符可施加于不同类型操作数上面。

设计原则

- 单一职责原则
- 开闭原则
- 里氏替换原则
- 依赖倒置原则
- 接口隔离原则
- 控制反转
- 依赖注入
- 面向切面编程

4 对象模型 UML

四种主要关系

1. 关联关系(Association)
2. 依赖关系(Dependency)
3. 泛化(一般化)关系(Generalization)
   泛化指的是类之间的继承关系
4. 聚集关系(Aggregation)
   聚集指的是整体与部分之间的关系，在实体域对象之间很常见

UML中的两类九种图

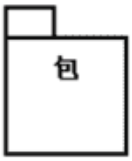
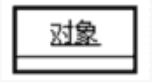
静态模型图：描述系统的结构

1. 类图
2. 对象图
3. 组件图
4. 部署图

动态模型图：描述系统的行为

1. 用例图
2. 活动图
3. 时序图
4. 协作图
5. 状态图

类图的基本符号

	类
	接口
	包
	对象
	关联
	组成关联
	聚集关联
	链接

定义类

UML中类的图形符号为**长方形**，用两条横线把长方形分**上、中、下**3个区域，3个区域分别放类**的名字、属性和服务**

命名规则

类名应该是**富于描述的、简洁的而且无二义性的**

- 使用**标准术语**，不要随意创造名字
- 使用具有**确切含义**的名词，不要使用空洞或含义模糊的词作名字
- 必要时可用**名词短语**作名字，有时也可以加入形容词

定义属性

可见性属性名： 类型名 = 初值 {性质串}

- **可见性**：有公有的（+）、私有的（-）和保护的（#）

  > 注意，没有默认的可见性

- **类型名**：表示该属性的数据类型

  > 属性名和类型名之间用冒号(:)分隔

- **赋值**：在创建类的实例时应给其他属性赋值，如果给某个属性定义了初值，则该初值可作为创建实例时这个属性的默认值

> 类型名和初值之间用等号（=）隔开

- **性质串**：明确地列出该属性所有可能取值，用逗号隔开

符号	表示
+	public
-	private
#	protected
~	package(可以理解为静态类型)

定义服务

可见性操作名（参数表）： 返回值类型{性质串}

- **可见性**：有公有的（+）、私有的（-）和保护的（#）
- **参数表**：用逗号隔开不同参数，每个参数语法为“`参数名:类型名=默认值`”

表示关系的符号

类与类之间通常具有以下四种关系

关联

关联表示两个类的对象之间存在某种**语义上的联系**

关联使用实线 + 简单箭头

关联的角色

在任何关联中都会涉及**参与此关联的对象所扮演的角色**，在某些情况下显式标明角色名有助于别人理解类

如果没有显式标出角色名，则意味着用类名作为角色名

（1）普通关联

普通关联是最常见的关联关系，**只要在类与类之间存在连接关系就可以用普通关联表示**

表示

- 普通关联的图示符号是**连接两个类之间的直线**
- 关联是**双向**的，可为关联起一个名字。在名字前面(或后面)加一个**表示关联方向的黑三角**
- 在表示关联的直线两端可以写上**重数**，它表示该类有多少个对象与对方的一个对象连接。**未明确标出关联的重数，则默认重数是1**

重数	表示
0..1	0到1个对象
0.. *或 *	0到多个对象
1 +或1.. *	1到多个对象
1..15	1到15个对象
3	3个对象

（2）限定关联

限定关联通常用在一对多或多对多的关联关系中，可以把模型中的重数从一对多变成一对一， 或从多对多简化成多对一

（3）关联类

为了说明关联的性质，可能需要一些**附加信息。**关联类可以用来**记录相关信息**

关联类通过一条**虚线**与关联连接

- 关联类与一般的类一样，也有属性、操作和关联
- 关联中的每个连接与关联类的一个对象相联系

聚集

也称为聚合，是关联的特例

聚集表示类与类之间的关系是整体与部分的关系。使用的“包含”、“组成”、“分为……部分”等字句，意味着存在聚集关系。有共享聚集和组合聚集两种特殊的聚集关系

（1）一般聚集

部分可以脱离整体存在，一般聚集和共享聚集的图示符号，都是在表示关联关系的直线末端紧挨着整体类的地方画一个**空心菱形**

（2）共享聚集

如果在聚集关系中处于部分方的对象可同时参与多个处于整体方对象的构成，则该聚集称为**共享聚集**

（3）组合聚集

如果部分类完全隶属于整体类，部分与整体共存，整体不存在了部分也会随之消失，则该聚集称为组合聚集(组成)

组成关系用**实心菱形**示例

泛化(继承)

泛化关系就是继承关系，**它是通用元素和具体元素之间的一种分类关系。**具体元素完全拥有通用元素的信息，并且还可以附加一些其他信息。在UML中，用**一端为空心三角形的实线连线表示泛化关系，三角形的顶角紧挨着通用元素**

泛化关系指出在类与类之间存在“一般--特殊”关系。泛化可进一步划分成普通泛化和受限泛化

（1）普通泛化

①：抽象类

没有具体对象的类称为抽象类。抽象类通常都有抽象操作，来指定该类的所有子类应具有哪些行为

表示抽象类是在类名下方附加一个标记值{abstract}，表示抽象操作是在操作标记后面跟随一个性质串{abstract}

②：具体类

具体类有自己的对象，并且该类的操作都有具体的实现方法

（2）受限泛化

给泛化关系附加约束条件，以进一步说明该泛化关系的使用方法或扩充方法，这样的泛化关系称为受限泛化。

预定义的约束有4种（都是语义约束）

- **多重**：一个子类可以同时多次继承同一个上层基类
- **不相交**：一个子类不能多次继承同一个基类。-般的继承都是不相交继承
- **完全**：父类的所有子类都已在类图中穷举出来了
- **不完全**：父类的子类并没有都穷举出来，随着对问题理解的深入，可不断补充和维护。是默认的继承关系

实现

实现关系用来表示**类与接口**之间的实现关系，用一条**虚线空心箭头**由子类指向父类

（实现是虚线，继承是实线）

依赖和细化

依赖关系

依赖关系描述两个模型元素之间的**语义连接关系**：

其中**一个模型元素是独立的，另一个模型元素不是独立的**，它依赖于独立的模型元素，**如果独立的模型元素改变了，将影响依赖于它的模型元素**

- 在UML类图中用**带简单箭头的虚线**连接有依赖关系的两个类，箭头指向独立的类。在虚线上可以带一个**版类标签**，具体说明依赖的种类

细化关系

对同一个事物在不同抽象层次上描述时，这些描述之间具有细化关系

- 细化的图示符号为由元素B指向元素A的一端为**空心三角形的虚线**

900分析

- 面向对象分析中，主要由对象模型、动态模型和功能模型组成
- 面向对象分析的关键工作，是分析、确定问题域中的对象及对象间的关系，并建立起问题域的对象模型

1 面向对象分析的基本过程

抽取和整理用户需求并建立问题域精确模型的过程

概念

主题

指导读者理解大型、复杂模型的一种机制。即通过划分主题把一个大型、复杂的对象模型分解成几个不同的概念范畴

7 +- 2 原则

短期记忆能力一般限于**一次记忆5~9个对象**

面向对象从两个方面体现这条原则

- 控制可见性：控制读者能见到的层次数目来控制可见性
- 指导读者注意力：增加了主题层，可从高层次描述总体模型，并指导读者的注意力

3 建立对象模型

（1） 基本概念

对象模型

面向对象分析的首要工作就是建立问题域的**对象模型。**

对象模型表示**静态的、结构化的系统的**数据性质。它是对模拟客观世界实体的对象以及对象彼此间的关系的映射，描述了系统的**静态结构。**对象模型为建立动态模型和功能模型，提供了实质性的框架

原因

- 静态数据结构对**应用细节**依赖较少，比较容易确定
- 当用户的需求变化时，静态数据结构相对来说**比较稳定**

信息来源

需求陈述、应用领域的专业知识、客观世界的常识，是建立对象模型时的主要信息来源

典型的建模步骤

- 确定对象**类和关联**(对于大型复杂问题还要进一步划分出若干个主题)
- 给类和关联增添**属性**，以进一步描述它们
- 使用适当的**继承**关系进一步合并和组织类
- 对类中操作的最后确定，则需等到建立了动态模型和功能模型之后，因为这两个子模型更准确地描述了对类中提供的服务的需求。

（2）确定类与对象

找出候选的类与对象

客观事物分类法

对象是对问题域中有意义的事物的抽象，它们既可能是物理实体，也可能是抽象概念。

非正式分析法

以用自然语言书写的需求陈述为依据，把陈述中的**名词作为类与对象的候选者**，用**形容词作为确定属性的线索**，把**动词作为服务的候选者**

这种方法确定的候选者是非常不准确的，其中往往包含大量不正确或不必要的事物，需要经过**更进一步的严格筛选**

筛选出正确的类与对象

冗余

如果两个类表达了同样的信息，则应该保留在此问题域中**最富于描述力**的名称

无关

需要把**与本问题密切相关的类与对象**放进目标系统中

笼统

系统无须记忆**笼统的、泛指的名词信息**，把这些笼统的或模糊的类去掉

属性

把描述的是**其他对象属性的词**从候选类与对象中去掉

操作

慎重考虑**既可作为名词，又可作为动词的词**，以便正确地决定把它们作为类还是作为类中定义的操作。本身具有属性，需独立存在的操作，应该作为类与对象

实现

应该去掉**仅和实现**有关的候选的类与对象

（3）确定关联

关联

两个或多个对象之间的**相互依赖、相互作用**的关系就是关联。在需求陈述中使用的**描述性动词或动词词组**,通常表示关联关系

确定关联的重要性

分析确定关联，能促使分析员考**虑问题域的边缘情况**，有助于发现**尚未被发现的类与对象**

初步确定关联

- 直接提取**动词短语**得出的关联
- **需求陈述**中隐含的关联
- 根据**问题域知识**得出的关联

筛选

根据下述标准**删除**候选的关联

- **已删去的类之间的关联**：如果在分析确定类与对象的过程中已经删掉了某个候选类，则与这个类有关的关联也应该删去，或用其他类重新表达这个关联
- **与问题无关的或应在实现阶段考虑的关联**：应该把处在本问题域之外的关联与实现密切相关的关联删去
- **瞬时事件**：关联应该描述问题域的静态结构，而不应该是一个瞬时事件
- **三元关联**：三个或三个以上对象间的关联，可分解为二元关联或用词组描述成限定的关联
- **派生关联**：去掉那些可以用其他关联定义的冗余关联

完善

- **正名**：仔细选择含义更明确的名字作为关联名
- **分解**：为了能够适用于不同的关联，必要时应该分解以前确定的类与对象
- **补充**：发现了遗漏的关联就应该及时补上
- **标明重数**：应该初步判定各个关联的类型，并粗略地确定关联的重数

如此即可获得原始的类图

（4）划分主题

在开发大型、复杂系统的过程中，为了降低复杂程度,把系统再进一步划分成几个不同的主题，即在概念上把系统包含的内容分解成若干个范畴

针对不同类型的方法

- **规模小的系统**：可能无须引入主题层
- **含有较多对象的系统**：首先识别出类与对象和关联，然后划分主题，并用它作为指导开发者和用户观察整个模型的一种机制
- **规模大的系统**：首先由高级分析员粗略地识别对象和关联，然后初步划分主题，经进一步分析，对系统结构有更深入的了解之后，再进一步修改和精炼主题

原则

- 按**问题领域**而不是用功能分解方法来确定主题
- 按照**使不同主题内的对象相互间依赖和交互最少**的原则来确定主题

（5）确定属性

属性是对象的性质，借助于属性人们能对类与对象和结构有更深入更具体的认识

- 注意:在分析阶段不要用属性来表示对象间的关系，使用关联能够表示两个对象间的任何关系，而且把关系表示得更清晰、更醒目

分析

- 在需求陈述中用**名词词组**表示属性，用**形容词**表示可枚举的具体属性
- 借助于**领域知识**和**常识**分析需要的属性
- 仅考虑与**具体应用**直接相关的属性，不要考虑那些超出所要解决的问题范围的属性
- 首先找出**最重要**的属性，以后再逐渐把其余属性增添进去
- 不要考虑那些**纯粹用于实现**的属性

选择

从初步分析确定下来的属性中删掉不正确的或不必要的属性

- **误把对象当作属性**：如果某个实体的独立存在比它的值更重要，则应把它作为一个对象而不是对象的属性
- **误把关联类的属性当作一般对象的属性**：如果某个性质依赖于某个关联链的存在，则该性质是关联类的属性，在分析阶段不应把它作为一般对象的属性
- **把限定误当成属性**：如果把某个属性值固定下来以后能减少关联的重数，则应该考虑把这个属性重新表达成一个限定词。

- **误把内部状态当成了属性**：如果某个性质是对象的非公开的内部状态，则应该从对象模型中删除这个属性。
- **过于细化**：在分析阶段应该忽略那些对大多数操作都没有影响的属性
- **存在不一致的属性**：类应该是简单而且一致的。如果得出一些看起来与其他属性毫不相关的属性，则应该考虑把类分解成两个不同的类

（6）识别继承关系

建立继承关系的方式

确定了类中应该定义的属性之后，就可以利用继承机制共享公共性质，并对系统中众多的类加以组织。可以使用以下两种方式建立继承关系

- **自底向上**：抽象出现有类的共同性质泛化出父类，这个过程实质上模拟了人类**归纳**思维的过程
- **自顶向下**：把现有类细化成更具体的子类，这模拟了人类的**演绎**思维过程。从应用域中常常能明显看出应该做的自顶向下的具体化工作

多重继承

利用多重继承可以**提高共享程度**，但增加了**概念上以及实现时**的复杂程度

要点

- 指定一个**主要父类**，从它继承大部分属性和行为；
- 次要父类只补充一些属性和行为

（7）反复修改

必要性

软件开发过程就是一个**多次反复修改、逐步完善**的过程。仅仅经过一次建模过程很难得到完全正确的对象模型

面向对象在修改时的优点

面向对象的概念和符号在整个开发过程中都是**一致的**，比使用结构分析、设计技术更容易实现反复修改、逐步完善的过程

4 建立动态模型

概念

适用性

- 对于**仅存储静态数据的**系统来说，动态模型并没有什么意义
- 在开发**交互式系统**时，动态模型却起着很重要的作用

- **收集输入信息**是系统的主要工作时，则在开发时建立正确的动态模型是至关重要的

步骤

1. 编写典型**交互行为**的脚本
2. 从脚本中提取出事件，确定**触发每个事件的动作对象以及接受事件的目标对象**
3. 排列事件发生的**次序**，确定每个对象的**状态及状态间的转换关系**，用**状态图**描绘
4. 比较各个对象的状态图，确保事件之间的**匹配**

（1）编写脚本

脚本是指**系统在某一执行期间内出现的一系列事件。**脚本描述用户与目标系统之间的一个或多个典型的交互过程。编写脚本的过程，就是**分析用户对系统交互行为的要求的过程**

目的

保证**不遗漏重要的交互步骤**，有助于确保交互过程的**正确性、清晰性**

内容

脚本描写的范围主要由编写脚本的具体目的决定，既可以包括**系统中发生的全部事件**，也可以只包括由**某些特定对象触发的事件**

方法

- 编写正常情况的脚本
- 考虑特殊情况
- 考虑出错情况

（2）设想用户界面

用户界面的美观程度、方便程度、易学程度以及效率等，是用户使用系统时最先感受到的。用户界面的好坏往往对用户是否喜欢、是否接受一个系统起很重要的作用

目的

这个阶段用户界面的细节并不太重要，重要的是在这种界面下的**信息交换方式。**目的是确保能够完成**全部必要的信息交换**，而不会丢失重要的信息

方法

快速地建立起用户界面的原型，供用户试用与评价

（3）画事件跟踪图

用自然语言书写的脚本往往**不够简明**，而且有时在阅读时会有二义性。为了有助于建立动态模型，需要画出事件跟踪图

1.确定事件

提取出所有外部事件

- 找出**正常事件**、**异常事件**和**出错条件**(传递信息的对象的动作也是事件)
- 把**对控制流产生相同效果的事件**组合为一类事件，并取一个**唯一的名字**

2.画出事件跟踪图

- 一条竖线代表一个**对象**
- 每个事件用一条**水平的箭头线**表示
- 箭头方向从事件的**发送对象指向接受对象**
- 时间从**上向下**递增
- 用箭头线在垂直方向上的相对位置表示**事件发生的先后**，不表示事件间的时间差

（4）画状态图

状态图描绘**事件与对象状态的关系。**当对象接受了一个事件以后，它的下个状态取决于**当前状态及所接受的事件。**由事件引起的改变称为“**转换**”。一张状态图描绘了**一类对象的行为**，它确定了**由事件序列引出的状态序列**

适用性

对于**仅响应与过去历史无关的那些输入事件**,或者**把历史作为不影响控制流的参数类**的对象，状态图是不必要的

方法

- 仅考虑事件跟踪图中**指向某条竖线的那些箭头线。**把这些事件作为状态图中的**有向边**，边上标以**事件名**
- 两个事件之间的间隔就是一个**状态**，每个状态取个有意义的名字。从**事件跟踪图中当前考虑的竖线射出的箭头线**，是这条竖线代表的对象达到某个状态时所做的行为。
- 根据一张事件跟踪图画出状态图后，再把**其他脚本的事件跟踪图**合并到该图中
- 考虑完正常事件后再考虑**边界情况和特殊情况**，包括在不适当时候发生的事件

（5）审查动态模型

- 检查系统级的**完整性**和**一致性**
- 审查每个事件，跟踪**它对系统中各对象所产生的效果**，保证与每个脚本都匹配

5 建立功能模型

功能模型表明了**系统中数据之间的依赖关系**，以及有关的数据处理功能，它由一组**数据流图**组成。在建立了对象模型和动态模型之后再建立功能模型

（1）画出基本系统模型图

基本的系统模型有下述两部分组成：

- **数据源点/终点**：数据源点输入的数据和输出到数据终点的数据，是**系统与外部世界**间交互事件的参数
- **处理框**：代表了**系统加工、变换数据**的整体功能

（2）画出功能级数据流图

把基本系统模型中单一的处理框分解成**若干个处理框**，以描述系统加工、变换数据的**基本功能**，就得到功能级数据流图

（3）描述处理框功能

要点

着重描述**每个处理框所代表的功能**，而不是实现功能的具体算法

分类

- **说明性描述（更为重要）**：规定了**输入值和输出值**之间的关系，以及**输出值**应遵循的规律
- **过程性描述**：通过算法说明 做什么

1000设计

1 面向对象设计的准则

模块化

抽象

信息隐藏

高内聚

低耦合

可重用

2 启发规则

设计结果应该清晰易懂

一般一特殊结构的深度适当

设计简单的类

使用简单的协议

使用简单的服务

把设计变动减至最小

3 软件重用

重用也叫再用或复用，是指同一事物不作修改或稍加改动就多次重复使用

软件重用可分为以下3个层次

- 知识重用
- 方法和标准的重用
- 软件成分的重用

类构件

面向对象技术中的“类”，是比较理想的可重用软构件

类构件的重用方式

实例重用

- 使用适当的构造函数，按照需要创建类的实例
- 用几个简单的对象作为类的成员创建出一个更复杂的类

继承重用

继承性提供了一种对已有的类构件进行裁剪的机制

多态重用

- 使对象的对外接口更加一般化，降低了消息连接的复杂程度
- 提供一种简便可靠的软构件组合机制

4 系统分解

分解思想

在设计比较复杂的应用系统时，先把系统分解成**若干个较小部分**，然后分别设计每个部分。这样做有利于降低设计的难度，有利于分工协作，也有利于维护人员对系统理解和维护

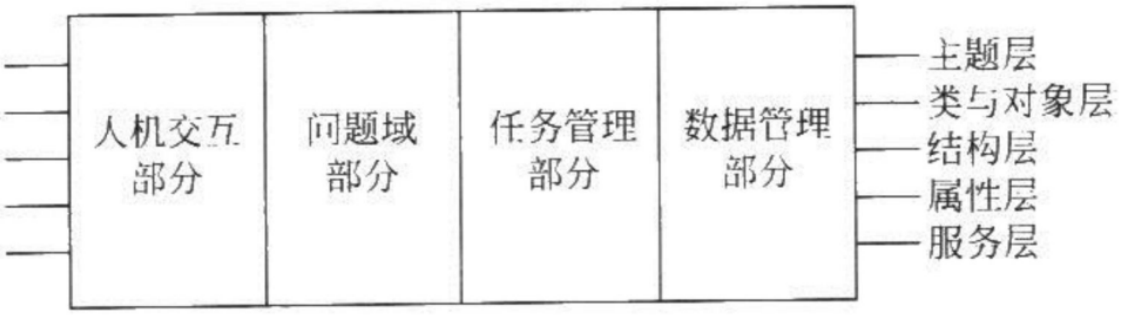
子系统

系统的主要组成部分称为子系统，通常根据所提供的功能来划分子系统

划分原则

- 根据所提供的**功能**来划分子系统，子系统数目应该与系统规模基本匹配
- 各个子系统之间应该具有尽可能简单、明确的接口
- 应该尽量减少子系统彼此间的**依赖性**

面向对象设计模型



- 面向对象设计模型由**主题、类与对象、结构、属性、服务**5个层次组成。这5个层次一层比一层表示的细节更多，可以把这5个层次想象为整个模型的水平切片
- 面向对象设计模型在逻辑上都由4大部分组成，分别对应于组成目标系统的4个子系统，即**问题域子系统**、**人机交互子系统**、**任务管理子系统**和**数据管理子系统**

子系统间交互方式

客户–供应商关系

作为“客户”的子系统调用作为“供应商”的子系统，后者完成某些服务工作并返回结果。作为客户的子系统必须了解作为供应商的子系统的接口，后者却无须了解前者的接口

平等伙伴关系

每个子系统都可能调用其他子系统，每个子系统都必须了解其他子系统的接口。由于各个子系统需要相互了解对方的接口，子系统之间的交互复杂，且还可能存在通信环路

> 总地说来，单向交互比双向交互更容易理解，也更容易设计和修改，因此应该**尽量使用客户–供应商关系**

组织系统的方案

层次组织

软件系统组织成一个**层次系统**，每层是一个**子系统。**上层在下层的基础上建立，下层为实现上层功能而提供必要的**服务。**每一层内所包含的对象，彼此间**相互独立**，而处于不同层次上的对象，彼此间有关联。在上、下层之间存在**客户–供应商**关系。低层子系统提供服务，上层子系统使用下层提供的服务

- **封闭式**：每层子系统仅仅使用其**直接**下层提供的服务。降低了各层次之间的相互依赖性，更容易理解和修改
- **开放式**：子系统可以使用处于其下面的任何一层子系统所提供的服务。优点是减少了需要在每层重新定义的服务数目，使系统更高效更紧凑。但其不符合**信息隐藏**原则

块状组织

把软件系统**垂直**地分解成若干个**相对独立的、弱耦合**的子系统，一个子系统相当于一块，每块提供**一种类型的服务**

层次和块状的结合

当混合使用层次结构和块状结构时，**同一层次可以由若干块组成，而同一块也可以分为若干层**

1100实现

略

12软件项目管理

1 估算软件规模

代码行技术

功能点技术

2 工作量估算

工作量是软件规模的函数，工作量的单位通常是**人月**(pm)

3 进度计划

估算开发时间

成本估算模型也同时提供了估算开发时间T的方程。与工作量方程不同，各种模型估算开发时间的方程很相似

```
Brooks规律：  向一个已经延期的项目增加人力，只会使得它更加延期
```

生产率略

甘特图

甘特图是制定进度计划的工具，优点是能形象描述任务分解情况，直观简洁和容易掌握

缺点

（1）不能显式地描绘各项作业彼此间的依赖关系。

（2）进度计划的关键部分不明确，难于判定哪些部分应当是主攻和主控的对象。

（3）计划中有潜力的部分及潜力的大小不明确，往往造成潜力的浪费

工程网络

> 当把一个工程项目分解成许多子任务，并且它们彼此间的依赖关系又比较复杂时，仅仅用Gantt图作为安排进度的工具是不够的，不仅难于做出既节省资源又保证进度的计划，而且还容易发生差错

工程网络是制定进度计划时另一种常用的图形工具，它同样能描绘任务分解情况以及每项作业的开始时间和结束时间，此外，它还显式地描绘各个作业彼此间的依赖关系

绘制

- 用箭头表示作业(例如，刮旧漆，刷新漆，清理等)
- 用圆圈表示事件(一项作业开始或结束)
- 事件仅仅是可以明确定义的时间点，它并不消耗时间和资源
- 作业通常既消耗资源又需要持续一定时间
- 用开始事件和结束事件的编号标识一个作业
- 虚线箭头表示虚拟作业，也就是事实上并不存在的作业。为了显式地表示作业之间的依赖关系

估算工程进度

画出工程网络之后，系统分析员就可以借助它的帮助估算工程进度了。为此需要在工程网络上增加一些必要的信息

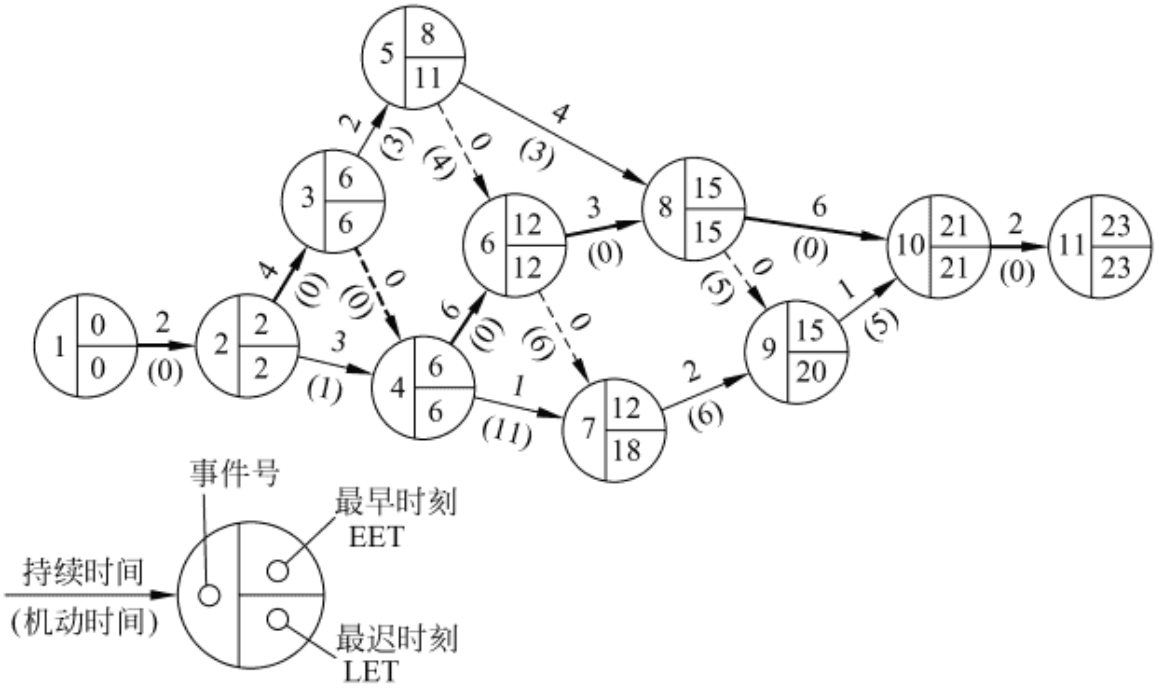
补充图例

- 把每个作业估计需要使用的时间写在表示该项作业的箭头上方。

  > 注意，箭头长度和它代表的作业持续时间没有关系，箭头仅表示依赖关系，它上方的数字才表示作业的持续时间

- 为每个事件计算下述两个统计数字： 最早时刻EET和最迟时刻LET

  > 这两个数字将分别写在表示事件的圆圈的右上角和右下角



EET

事件的最早时刻是该事件可以发生的最早时间

通常工程网络中第一个事件的最早时刻定义为零，其他事件的最早时刻在工程网络上从左至右按事件发生顺序计算。计算最早时刻EET使用下述3条简单规则

（1）考虑进入该事件的所有作业

（2）对于每个作业都计算它的持续时间与起始事件的EET之和

（3）选取上述和数中的最大值作为该事件的最早时刻EET

LET

事件的最迟时刻是在不影响工程竣工时间的前提下，该事件最晚可以发生的时刻。

关键路径

关键路径用粗线箭头表示

关键路径上的事件(关键事件)必须准时发生，组成关键路径的作业(关键作业)的实际持续时间不能超过估计的持续时间，否则工程就不能准时结束

> 工程项目的管理人员应该密切注视关键作业的进展情况，如果关键事件出现的时间比预计的时间晚，则会使最终完成项目的时间拖后；如果希望缩短工期，只有往关键作业中增加资源才会有效果

机动时间

不在关键路径上的作业有一定程度的机动余地—实际开始时间可以比预定时间晚一些，或者实际持续时间可以比预定的持续时间长一些，而并不影响工程的结束时间

一个作业可以有的全部机动时间等于它的结束事件的最迟时刻减去它的开始事件的最早时刻，再减去这个作业的持续时间：

$$机动时间 = (LET)结束-(EET)开始-持续时间$$

> 在工程网络中每个作业的机动时间写在代表该项作业的箭头下面的括号里

4 人员组织

- 民主制
- 主程序员组
- 现代程序员组

5 质量保证

软件质量就是**软件与明确地和隐含地定义的需求相一致的程度**

衡量因素

- 正确性
- 健壮性
- 效率
- 完整性
- 可用性
- 风险

质量保障措施

- 基于非执行测试（复审或评审）
- 基于执行测试（软件测试）
- 程序正确性的证明（数学方法）

6 软件配置管理

软件配置管理是在软件生命周期内管理变化的一组活动，用来标识、控制、报告变化，确保适当的实现了变化

基线

通过了正式复审的软件配置项，可以作为进一步开发的基础，只有通过正式的变化控制过程才能改变它

软件配置管理过程

- 标识对象
- 版本控制
- 变化控制
- 配置审计
- 状态报告

7 能力成熟模型

能力成熟度模型（CMM）是用于评价软件机构的软件过程能力成熟度模型，用于帮助软件开发机构建立一个有规模的，成熟的软件过程。

五个等级从低到高为

- 初始级
- 可重复级
- 已定义级
- 已管理级
- 优化级