

1、软件定义

- (1) IEEE 定义：软件是计算机程序、规程以及运行计算机系统所需要的文档和数据。
 - ① 计算机程序+规程+文档数据 = 软件
- (2) 另一种公认解释：软件是包括程序、数据及其相关文档的完整集合。
 - ① 程序+数据+相关文档 = 软件
 - ② 程序和数据是构造软件的基础
 - ③ 文档是软件质量的保证，也是保证软件更新和生命周期的必需品

2、软件特点

- (1) 逻辑实体，抽象
- (2) 开发过程中没有明显的制造过程
- (3) 存在软件退化问题
- (4) 软件开发和运行受到计算机系统的约束和限制
- (5) 开发尚未完全摆脱手工化的开发方式
- (6) 软件复杂的原因：实际需求的复杂性（业务背景）、程序逻辑复杂性
- (7) 软件研制成本高
- (8) 投入运行后涉及很多社会因素

3、软件分类

- (1) 服务对象范围：通用软件、定制软件
- (2) 软件完成功能所处层次：应用软件、中间件软件、系统软件【OS、设备驱动程序、数据库管理系统】

4、软件发展

(1) 软件危机：

① 背景：

1960 年后至 1970 之间的软件快速发展阶段（程序系统阶段），随着计算机软件应用领域增多，软件规模扩大，软件系统功能多，逻辑复杂，不断扩充，从而导致许多系统开发出现了不良的后果。

软件可用性、可靠性差

无法增加新的功能、难以维护

无法按计划时间完成

② 原因：

1) 软件系统本身的复杂性

2) 软件开发、维护方法不合理；

3) 落后的软件开发方法无法满足快速增长的软件需求

③ 定义：

计算机软件在开发和维护过程中所遇到的一系列严重问题，导致了软件行业的信任危机

④ 表现：

- 1) 软件开发成本难以估算，无法制定合理的开发计划；
- 2) 用户的需求无法确切表达；
- 3) 软件质量存在问题；
- 4) 软件的可维护性差；
- 5) 缺乏文档资料；

- ⑤ 解决途径：1968 年 软件工程大会第一次召开，提出“软件工程”的概念，运用工程化原则和方法，组织软件开发解决软件危机，

(2) 综述

在软件快速发展的程序系统阶段，软件的应用领域扩宽，规模逐渐增大，功能逐渐增多，逻辑更加复杂，软件系统复杂性进一步提升，落后的软件开发技术无法满足日益增长的软件开发需求，导致计算机软件在开发和维护阶段遇到了一系列严重问题，导致了软件行业的危机，也就是软件危机。具体体现在：软件开发成本无法估算、用户需求无法准确表达、软件质量差，可用性不高、软件难以维护和增加新功能、缺乏相关的文档资料。为了解决软件危机，人们提出了“软件工程”的概念，把工程化原则用于软件的开发和维护过程中。

(3) 软件发展阶段

- ① 程序设计、程序系统阶段、软件工程阶段、现代软件工程阶段

阶段 特点	程序设计	程序系统	(现代) 软件工程
软件所指	程序	程序及说明书	程序、文档和数据
程序设计语言	汇编及机器语言	高级语言	软件语言
软件工作范围	程序编写	包括设计和测试	软件生存期
需求者	程序设计本人	少数用户	市场用户
开发软件的组织	个人	开发小组	开发小组及大中型软件开发机构
软件规模	小型	中小型	大中小型
决定质量的因素	个人程序技术	小组技术水平	管理水平
开发技术和手段	子程序/程序库	结构化程序设计	数据库、开发工具、开发环境、工程化开发方法、标准和规范、网络及分布式开发、面向对象技术、软件复用
维护责任者	程序设计者	开发小组	专职维护人员
硬件特征	价格高/存储容量小 工作可靠性差	降价、速度、容量及工作可靠性明显提高	向超高速、大容量、微型化及网络化发展
软件特征	完全不受重视	软件技术的发展不能满足需求，出现软件危机	开发技术有进步，但未获突破性进展，价高，未摆脱软件危机

5、软件工程

(1) 定义：

- ① 得到经济可靠软件而建立使用的一系列工程化原则
- ② 用现在科技知识设计构造计算机程序及开发、运行、维护程序所需要的相关文件资料
- ③ 成本限额内按时完成开发和修改所需要的技术和管理

(2) IEEE 定义：

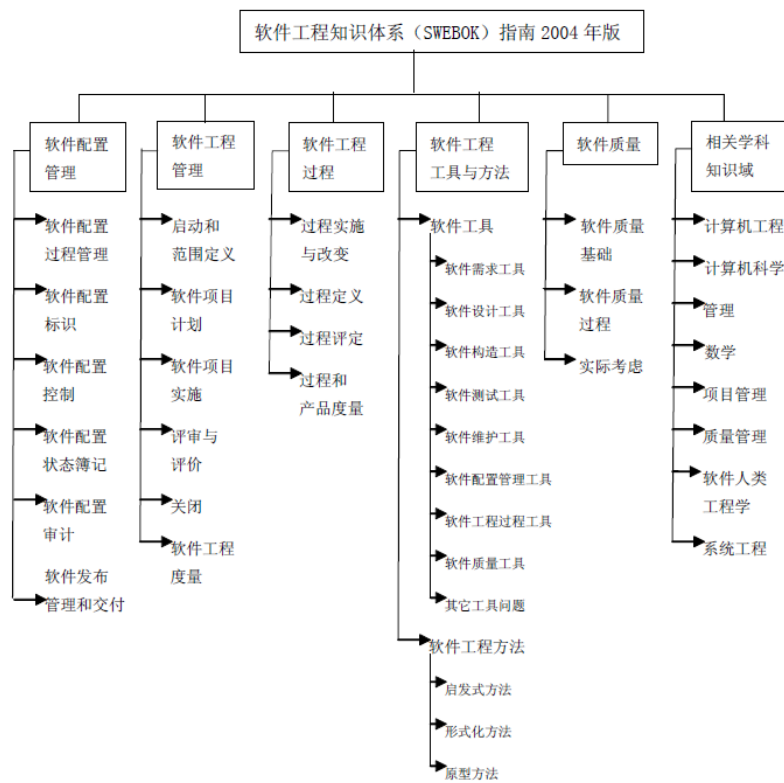
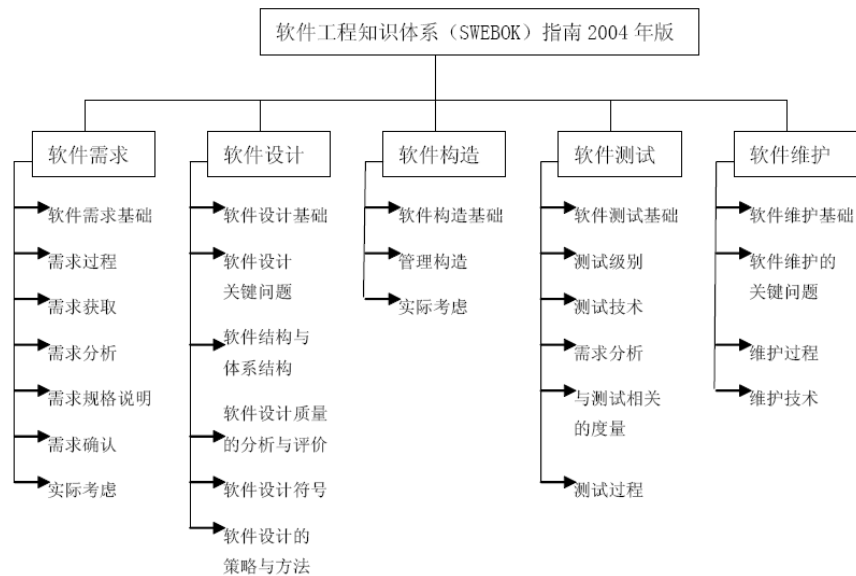
- ① 应用系统化的、规范化的、定量的方法来开发、运行和维护软件，即：将工程应用到软件；
- ② 及其各种方法的研究。

(3) 软件工程三要素：方法、工具、过程

(4) 目标和原则:

- ① 概括目标: 生产具有正确性、可用性以及开销适宜的软件产品。
- ② 最终目的: 摆脱手工生产软件的状况, 逐步实现软件研制和维护的自动化。

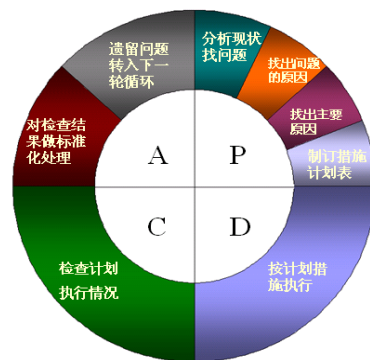
(5) 知识体系知识域



软件生命周期模型

一、工程过程

- (1) 工程项目目标：合理进度、有限经费、一定质量
 - ① **PDCA 循环**，针对**质量目标**提出的模型，称为**戴明环**（Plan, Do, Check, Action）
- (2) **软件工程过程**
 - ① 为了获得**软件产品**，在**软件工具**的支持下，由软件工程师完成的一系列软件工
程活动。
 - ② 具体活动包括：
 - 1) 软件规格说明
 - 2) 软件开发
 - 3) 软件确认
 - 4) 软件演进
- (3) **软件生命周期**：
 - ① 指软件产品从考虑其概念开始，到该软件产品不再使用为止的整个时期。
 - ② **软件生命周期的六个基本步骤**：
 - 1) 制定计划（P）
 - 2) 需求分析（D）
 - 3) 设计（D）
 - 4) 程序编码（D）
 - 5) 测试（C）
 - 6) 运行维护（A）
 - ③ 软件生命周期和戴明环的对应关系：



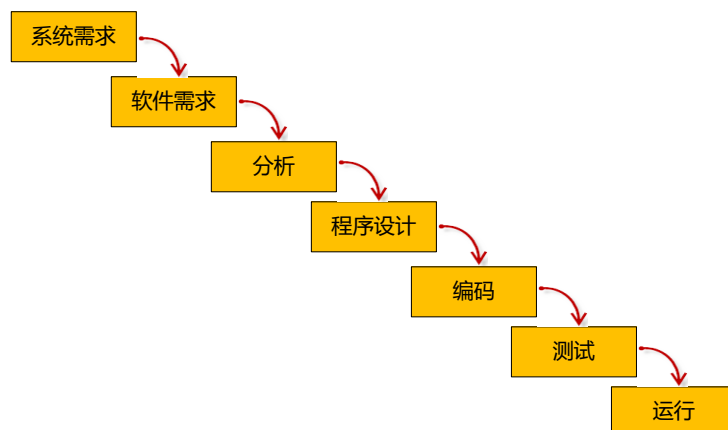
- (4) **软件过程模型**：
 - ① 即软件生命周期模型，是对软件过程的概括描述和抽象。
 - ② **包括**：
 - 1) 各种活动（Activities）
 - 2) 软件工件（artifacts）
 - 3) 参与角色（Actors/Roles）
- (5) **软件生命周期模型**
 - ① 需求定义
 - ② 跨生存期的软件开发、运行、维护全过程，活动任务描述生命周期不同阶段的软件工件，明确活动和执行角色

- ③ 指导软件开发人员按照确定的框架结构和活动进行软件开发的标准。

二、传统软件生命周期模型

1、瀑布模型

- (1) 定义：规定软件生命周期模型提出的一些基本工程活动，并规定他们自上而下相互衔接的固定次序。
- (2) 阶段划分：系统需求--软件需求---分析---程序设计---编码---测试---运行
- (3) 每个阶段都需要进行审核和文档输出
- (4) 意义：为软件开发和维护提供了有效的管理模式。在软件开发早期，在消除非结构化软件、降低软件复杂度、促进软件开发工程化方面起着显著的作用。
- (5) 每个开发活动的特征：
 - ① 本活动依赖于上一项活动的输出作为工作对象，这些输出一般是代表某活动结束的里程碑式文档。
 - ② 根据本阶段的活动规程执行相应的任务。
 - ③ 本阶段活动的最终产出——软件工件，将会作为下一活动的输入。
 - ④ 对本阶段活动执行情况进行评审。
- (6) 优点
 - ① 降低复杂度、提高透明性和可管理性、强调需求分析和设计、阶段审核和文档输出保证了阶段之间的正确衔接，能够及时发现问题
- (7) 缺点
 - ① 缺乏灵活性、不适用于需求不明的开发情况、风险控制能力较低、文档驱动增加了系统的工作量、只依赖于文档来评估进度，可能会得出错误结论、成功周期较长（do twice）
- (8) 适用场景
 - ① 需求明确、开发周期不紧张的较大型系统



2、演化模型

在瀑布模型的基础上，一次性开发难以成功，因此，演化模型提倡进行“两次开发”，分别是试验开发和产品开发。每个开发阶段按照瀑布模型进行具体开发活动。

优点：

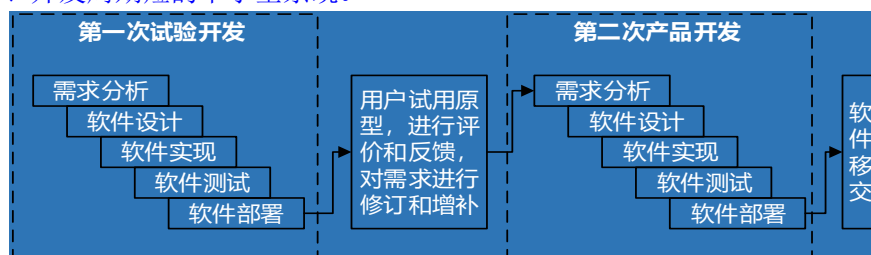
明确用户需求、提高系统质量、降低开发风险

缺点：

管理难度提高、开发结构化较差、可能抛弃文档驱动、可能导致产品结构化较差

适用场景：

需求不清、开发周期短的中小型系统。



3、增量模型

结合瀑布模型和演化模型的优点，在需求不清时，对最核心或最清晰的需求，利用瀑布模型实现。再对后续需求进行重复开发（可能按照需求的优先级逐个进行），从而逐步建成一个完整的软件系统。

优点：

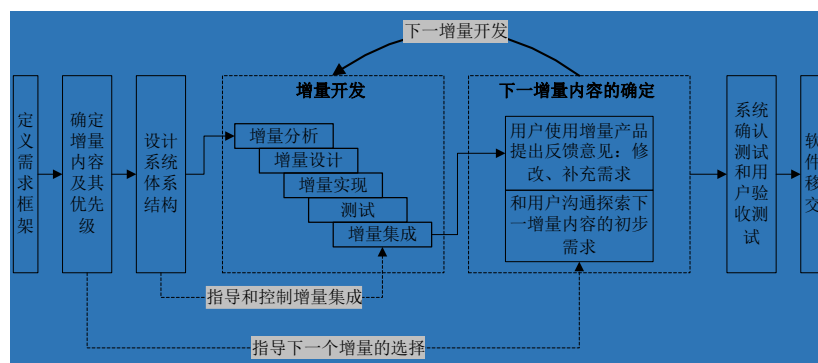
保障核心功能实现、开发风险较低、保障最高优先级的功能的可靠实现、提高系统稳定性和可维护性。

缺点：

增量粒度难以合理选择、确定所有的需求服务较困难

适用场景：

需求不清，了解核心需求的软件系统



4、喷泉模型/迭代模型

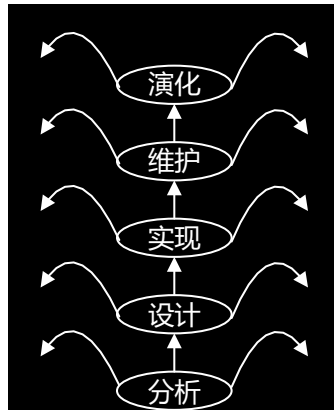
每个阶段是相互重叠、多次反复的。每个开发阶段没有次序要求，可以并发进行，在某个阶段随时补充其他阶段中遗漏的需求。

优点：

提高效率、缩短周期

缺点：

管理结构性较差



5、V 模型和 W 模型

V 模型

是瀑布模型的变种。它将测试模块细化分解，把测试看作与开发同等重要的过程，每一测试阶段的前提和要求是对应开发阶段的文档。

测试模块：

·**单元测试**：根据详细设计说明书，检测每个单元模块是否符合预期要求。主要检查编码过程中可能存在的错误。

·**集成测试**：根据概要设计说明书，检测模块是否正确聚集。主要检查模块与接口之间可能存在的错误。

·**系统测试**：根据需求分析，检测系统作为一个整体在预定环境中能否正常的有效工作。

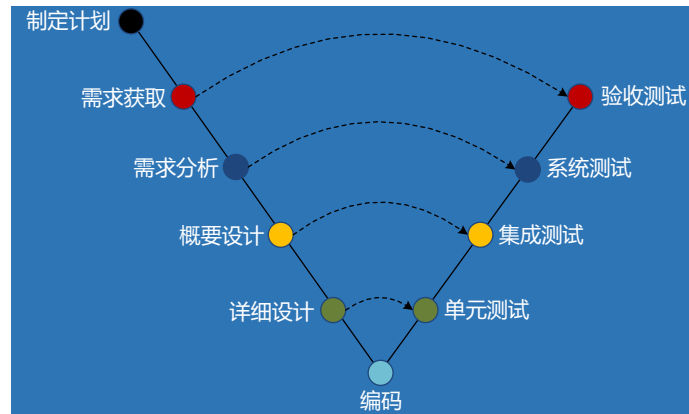
·**验收测试**：是否满足客户的需求。

优点：

改进开发效率和开发效果

缺点：

前期错误，后期难以挽救和弥补或者花费的代价巨大。



W 模型

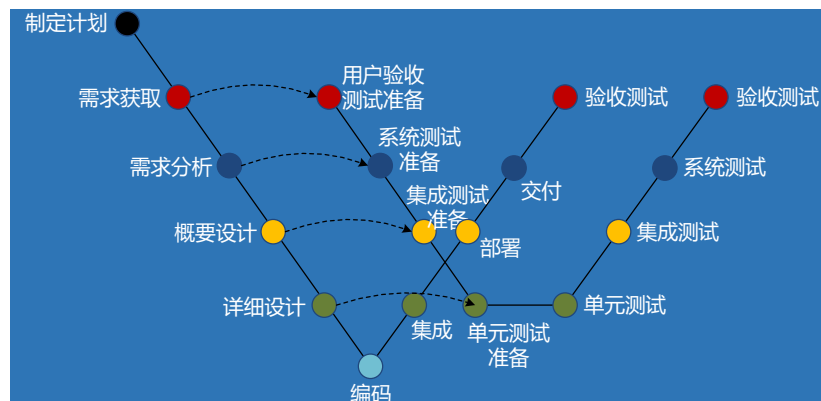
由两个 V 模型构成，分别代表测试与开发过程。每个开发过程对应一个测试，体现了开发和测试的并行关系，测试的不局限于程序，对于阶段文档也需要进行测试。

优点：

增加了软件各开发阶段中应同步进行的验证和确认活动。

缺点：

开发、测试活动保持着一种前后关系，无法支持迭代软件开发模型。



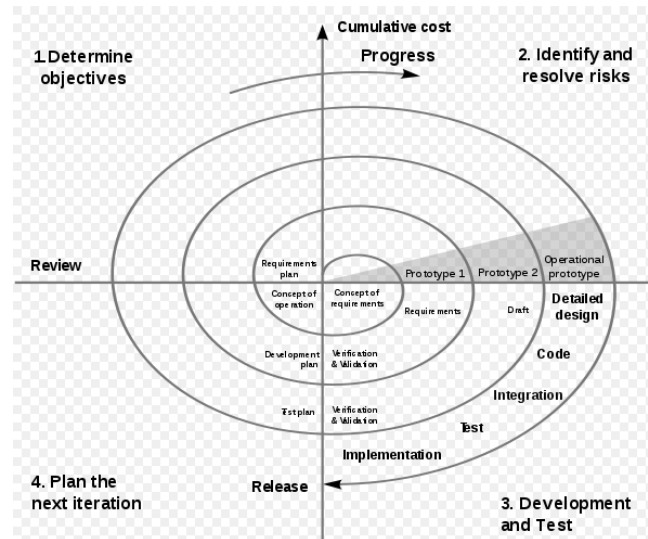
6、螺旋模型

将开发过程分为四个类型：风险分析、制定计划、实施工程、客户评估。每次评估之后确定是否进行螺旋线的下一个回路。

适用对象：

风险较高、开发周期较长的大型软件项目

优点和缺点：降低风险，但是开发周期长。



7、构件组装模型

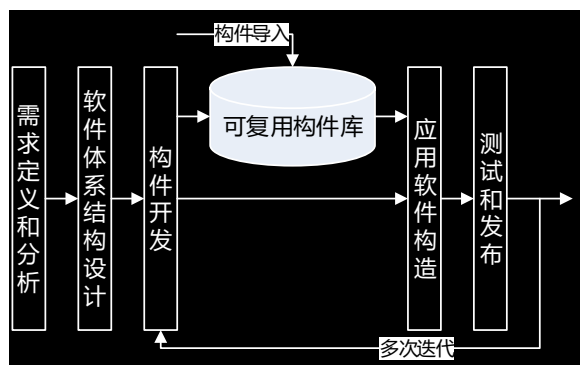
将整个系统模块化，在一定构件模型的支持下，复用构件库中软件构件，通过组装构件构造软件系统。开发过程就是构件组装的过程，是迭代的过程，维护过程就是构件升级、替换和扩充的过程。

优点：

软件复用充分，提高效率，允许多项目同时开发，降低开发费用，提高可维护性，可实现分步提交软件产品。

缺点：

构件组装结构没有通用标准，组装过程存在风险、构件可重用性和系统高效性不易协调；构件质量直接影响产品质量。



8、快速应用开发模型（RAD）

增量型开发模型

优点：

开发周期短

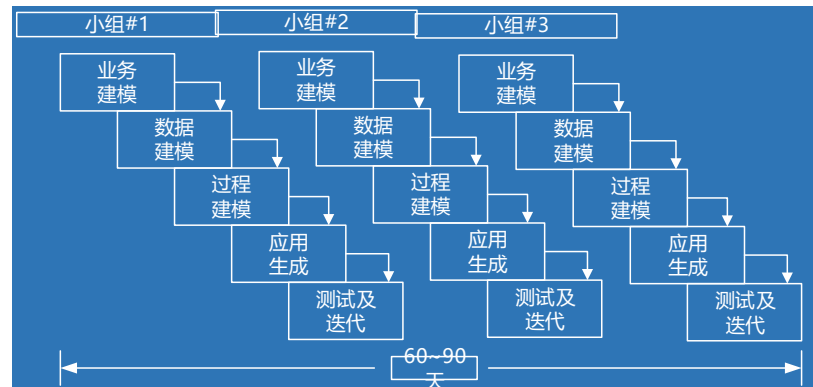
缺点：

需求分析的阶段时间较短、适用性一般、

适用范围：

适合于管理信息系统开发

不适合于技术风险较高、与外围系统的互动操作性较高的系统开发



原型方法

瀑布模型以及基于瀑布模型的软件生命周期模型，都需要精确的需求才能很好地执行后续的开发活动，但是准确的需求分析很难获取。

原型方法指在**获得基本需求后，快速分析，实现一个小型的软件系统原型**，满足用户的基本要求。用户可以提出修改意见，**校正需求**。

原型方法**主要用于明确需求，但也可以用于软件开发的其他阶段**。

种类：

1、废弃策略

·探索型：弄清用户要求，确定期望特性；探讨多个方案的可行性。主要针对需求模糊、用户和开发者对项目都缺乏经验的情况。

·实验型：用于大规模开发和实现之前，考核技术方案是否合适、分析和设计的规格说明是否可靠。

2、追加策略

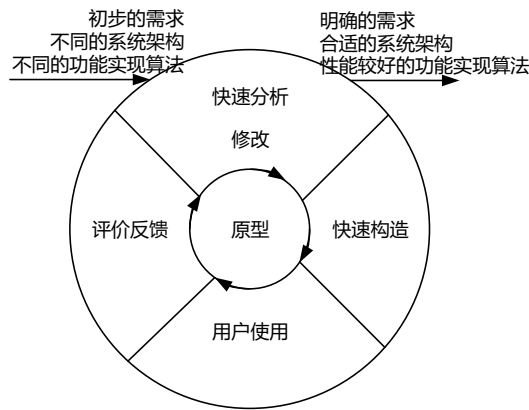
进化型：适应需求变化，不断改进原型，逐步将原型进化成最终系统。它将原型方法的思想扩展到软件开发的全过程，适用于需求经常变动的软件项目。

优点：

有助于快速理解用户需求、容易确定系统性能和设计可行性、有利于建成最终系统。

原型方法的缺点：

文档容易被忽略、建立原型增加工作量、项目难以有效规划和管理。



新型软件生命周期模型

1、RUP

面向对象的程序开发方法论，既是一个生命周期模型，也是一个支持面向对象软件开发的工具。

软件生命周期在时间上被分解为四个顺序的阶段：初始阶段、细化阶段、构造阶段、交付阶段。每个阶段在阶段结尾执行一次评估，确定阶段目标是否满足、是否可以进入下一个阶段。以用例为驱动，软件体系结构为核心，应用迭代及增量的新型软件生命周期模型。

各阶段工作说明

- 初始阶段**：了解业务，确定项目边界。包括验收规范、风险评估、资源估计、阶段计划制定等
- 细化阶段**：分析问题领域，建立软件体系结构基础，编制项目计划，完成技术要求高、风险大的关键需求开发。
- 构造阶段**：将所有剩余的技术构件和业务需求功能开发出来，集成产品，所有功能被详细测试。
- 移交阶段**：重点是确保软件可被最终用户使用。交付阶段可以跨越几次迭代，包括为发布做准备的产品测试，基于用户反馈的少量调整。

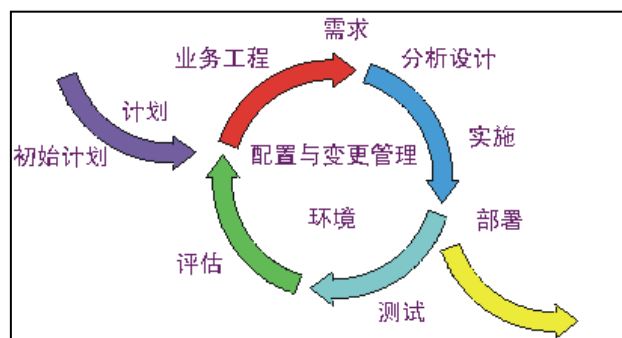
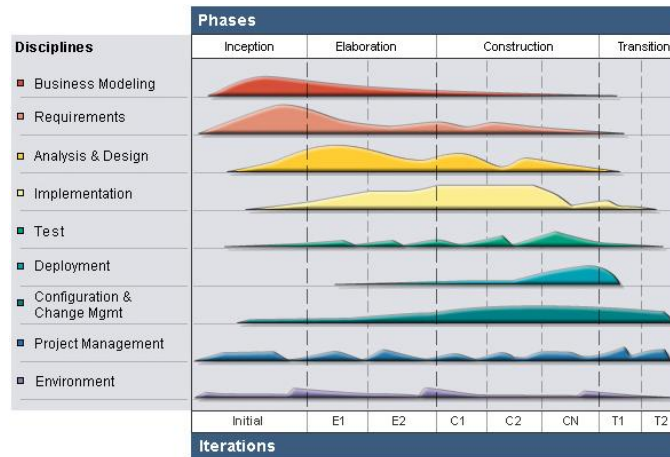
每个迭代的主要活动：

核心过程工作流（6个）：业务建模、需求、分析和设计、实现、测试、部署

核心支持工作流（3个）：配置和变更管理、项目管理、环境

模型特点：

- 适应性开发**：小步骤、快速反馈和调整；
- 使用**用例驱动**软件建模：用例是获取需求、制定计划、进行设计、测试、编写终端用户文档的驱动力量。
- 可视化软件建模**：使用 UML 进行软件建模。
- 用例驱动，架构核心，面向对象

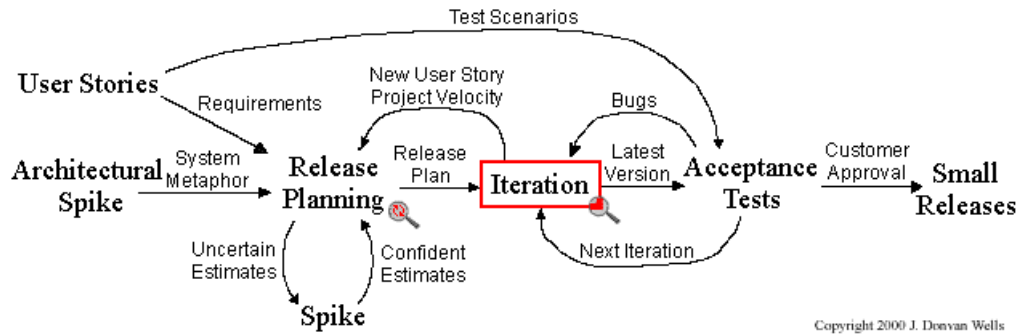


2、敏捷和极限编程（XP）

一种以实践为基础的软件工程过程和思想。使用**快速反馈**、**大量交流**，经过保证的测试来最大限度的满足用户的需求。主要强调用户满意，开发人员可以对需求的变化作出快速的反应。每个参加项目开发的人都将担任一个角色（项目经理、项目监督人等等）并履行相应的权利和义务。

特点：

- 测试驱动**：XP 内层的过程是一个基于 Test Driven Development 周期，每个开发周期都有很多相应的单元测试。
- 大力提倡设计复核（Review）、代码复核以及重整和优化（Refactory）**，这些过程也是优化设计的过程；
- 提倡**配对编程（Pair Programming）**，而且代码所有权是归于整个开发队伍。
- 程序员在写程序和重整优化程序时要**严格遵守编程规范**。
- 任何人可以修改其他人写的程序，但修改后要确定新程序能通过单元测试。
- 在开始写程序之前先写单元测试**



模型	特点	优点	缺点	适用场景
瀑布模型		降低了软件开发过程的复杂性、提高开发过程的透明度和可管理性，强调需求分析和设计，阶段审核和文档输出保证了阶段之间的正确衔接，有利于及时发现问题	缺乏灵活性，风险控制能力低，文档驱动增加了系统的工作量，开发周期较长	需求明确、开发周期不紧张的较大系统
演化模型	进行两次开发：试验开发+产品开发	明确用户需求，提高系统质量，降低风险	管理性差，开发结构性差，	需求不清，开发周期短的中小型系统
增量模型 (体现了原型方法)	优先开发核心需求	保障核心功能，开发风险低，提高系统稳定性和可维护性	增量粒度合理选择的难度大，确定所有需求难度大	需求不清，但是了解核心需求的系统
喷泉模型/迭代模型	阶段重复，可并行开发	提高效率，缩短开发周期	管理结构性差	
V 模型和 W 模型	重视测试，将测试提升到和开发一样重要的地位，测试模块细分	提高开发效率，改善开发效果，W 模型更能及时发现前期错误	不支持迭代开发	
螺旋模型	开发过程分为四个阶段：风险评估、制定计划、实施工程、客户评估，每次评估之后确定是否进入下一回路	降低风险	开发周期长	风险高、开发周期较长的大型软件项目

构建组件模型	系统模块化，并进行组件的复用和组装	复用充分，提高效率，降低开发费用，提高可维护性	没有通用标准，可重用性和高效性存在不易协调的问题，构件质量直接影响产品质量	
快速应用开发模型（RAD）	增量开发。多组并行	开发周期短	需求分析时间段	管理信息的系统开发，风险低，和外围系统互动操作性低
原型方法	获得基本需求，快速分析，不断矫正	有利于快速理解用户需求，确定系统性能和设计的可行性	忽略文档，原型增加工作量，管理性降低	
RUP	用例驱动，架构为核心，面向对象，可视化建模			
敏捷和极限编程	快速反馈，大量交流，测试驱动，明确编程规范并且严格遵守			

软件需求分析

1、需求分析之前的活动

- (1) 系统分析
 - ① 预研：探索软件项目的目标、市场预期、主要的技术指标等，帮助决策者判断是否进行软件项目立项。
- (2) 可行性分析
 - ① 针对项目的目标和范围进行概要的分析和研究，探索问题域中的核心问题及其相应的解决方案，进一步为决策者提供经济、技术甚至是法律上可行性的分析报告。
- (3) 需求分析前进行软件的系统分析和可行性分析

2、需求分析中的基本概念

(1) 需求的定义：

研究一种**无二义性的表达工具**，它能为用户和软件人员双方都接受，并能够把“需求”**严格地、形式地表达出来**。

需求是软件产品的**起源**，也是软件是否合格的**检验标准**。

(2) 需求分析必要性：

- ① 软件开发人员深入描述软件的功能和性能、指明软件和其他系统元素的接口，建立软件必须满足的约束条件。
- ② 允许软件开发人员对关键问题进行细化，并构建相应的分析模型：**数据、功能和行为模型**。
- ③ **是分析模型成为设计模型的基础**，需求规格说明书也为软件测试人员和用户提供了软件质量评估的依据。
- ④ 它能准确表达用户对系统的各项要求。

系统需求和软件设计之间的桥梁；

深入描述系统的功能和性能，指明接口；

细化问题，建立分析模型（数据/行为/功能）

(3) 需求分析的对象、任务和目标、原则

- ① **对象：用户需求**
- ② **任务：准确定义新系统的目标，编制《需求规格说明书》**
- ③ **目标：借助当前系统的逻辑模型导出目标系统的逻辑模型，解决目标系统的“做什么”的问题。**
- ④ 原则：
 - 1) **操作性原则**：信息与必须被表示和被理解；功能必须被定义；行为必须被表示
 - 2) **工程化原则**：

- a. 正确理解问题，之后再**建立分析模型**；
- b. 记录每个需求的起源及原因，保证需求的**可回溯性**；
- c. 开发人机交互过程的原型；
- d. 给需求赋予优先级
- e. 删除歧义性

(4) 需求分析的分析模型

① 数据模型：

- 1) 信息内容和关系；信息流；信息结构。

② 功能模型：

- 1) 对进入软件的信息和数据进行变换和处理的模块，它必须至少完成**三个常见功能**：输入、处理和输出。

③ 行为模型：

- 1) 大多数软件对来自外界的事件做出反应，这种刺激 / 反应特征形成了行为模型的基础。行为模型创建了软件状态的表示，以及导致软件状态变化的事件的表示。

(5) 需求工程

① 把所有与需求直接相关的活动通称为需求工程。

3、需求分析过程

(1) 需求获取--需求分类--需求分析和综合--需求建模--编制文档--确认和评审

(2) 需求获取

- 清楚理解所要解决的问题，完整获得用户需求，提出需求实现条件以及应达到的标准，经过分析之后**生成《用户需求说明书》**
- 难点：用户无法清楚表达需求、需求的理解问题、用户经常变更需求
- 需求获取的准备工作：
 - 准备需求调查问题表，将重点锁定在问题表中；
 - 确定调查方式；
 - 随时记录需求信息；
 - 对需求信息进行分析，消除错误，归纳和总结共性的用户需求；

(3) 需求分类

功能需求、性能需求、环境需求

可靠性需求、安全保密需求、用户界面需求、资源使用需求、软件成本消耗和开发进度需求、估计可能目标

(4) 需求分析和综合：

在需求分析之后，需要对需求进行建模分析，进而逐步细化所有的软件功能，找到系统各元素之间的联系、接口特性、设计限制，分析各元素是否满足功能要求，是否合理

(5) 需求建模：

描述系统需要做什么，而不是如何去做系统；给出系统的逻辑视图以及物理视图；

① 常用的建模分析方法

- 1) 面向数据流的结构化分析方法
- 2) 面向数据结构的 Jackson 方法
- 3) 面向对象的分析方法
- 4) 建立动态模型的状态迁移图

(6) 编制需求分析文档

软件需求规格说明书和用户需求说明书的编制

注意：《软件需求规格说明书》和《用户需求说明书》的区别

《用户需求说明书》是用户或者开发人员从用户角度，使用自然语言表达用户需求，相对简略

《软件需求规格说明书》使用计算机语言、图形符号，详细刻画需求分析，尽量消除了二义性，是软件设计的直接依据

(7) 需求确认和评审

需求规格说明书需要进行内部评审，和用户进行确认

面向对象需求分析方法

一、UML 统一建模语言

1、主要特点

软件界第一个统一的建模语言，标准的图形化建模语言，是面向对象分析与设计的一种标准表示。

- 不是可视化的程序设计语言
- 不是工具或知识库的规格说明
- 不是过程
- 不是方法
- 是一种建模语言，一种标准表示

2、基本结构

基本构造块：事物/关系/图

语义规则：name、scope、visibility、integrity、execution

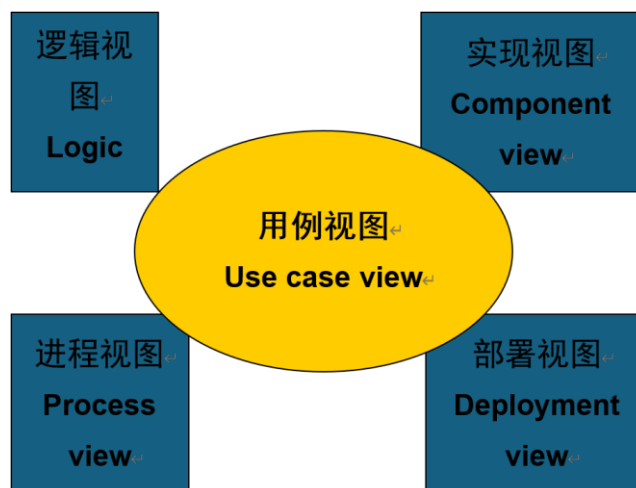
通用机制：specification、adornment、common division、extensibility mechanism

事物及关系：Structural thing、Behavior thing、Group thing、Annotation thing

3、UML 的视图

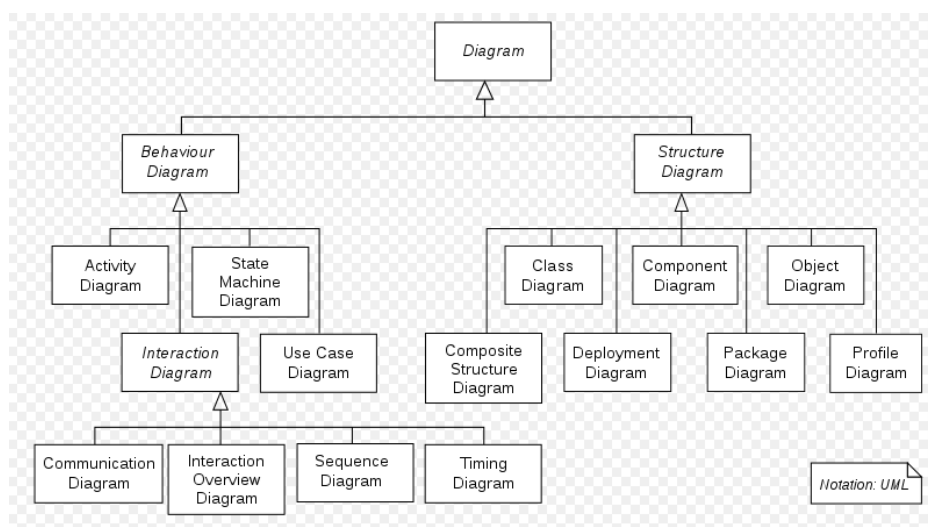
用模型来描述系统结构（静态特征）和行为（动态特征），从不同的视角为系统架构进行建模，从而形成不同的视图，主要有：

逻辑视图/结构模型视图/静态视图	展现系统的静态结构组成及特征
构件视图/实现模型视图/开发视图	关注软件代码的静态组织与管理
进程视图/行为模型视图/过程视图/协作视图/动态视图	描述设计的并发和同步等特性，关注系统非功能性需求
部署视图/环境模型视图/物理视图	描述硬件的拓扑结构以及软件和硬件的映射问题，关注系统非功能性需求（性能、可靠性等）
用例视图/用户模型视图/场景试图（核心）	强调从用户的角度看到的或需要的系统功能



4、9 个基本图

用例图 (Use case diagram)	(从用户的角度) 描述系统的功能
类图 (Class diagram)	描述系统的静态结构 (类及其相互关系)
对象图 (Object diagram)	描述系统在某个时刻的静态结构 (对象及其相互关系)
顺序图 (Sequence diagram)	按时间顺序描述系统元素间的交互
协作图 (Collaboration diagram)	按照时间和空间的顺序描述系统元素间的交互和它们之间的关系
状态图 (State diagram)	描述了系统元素 (对象) 的状态条件和响应
活动图 (Activity diagram)	描述了系统元素之间的活动
构件图 (Component diagram)	描述了实现系统的元素 (类或包) 组织
部署图 (Deployment diagram)	描述了环境元素的配置并把实现系统的元素映射到配置上



5、视图和基本图的关系

用例视图：用例图和活动图

逻辑视图：类图、对象图、顺序图/协作图

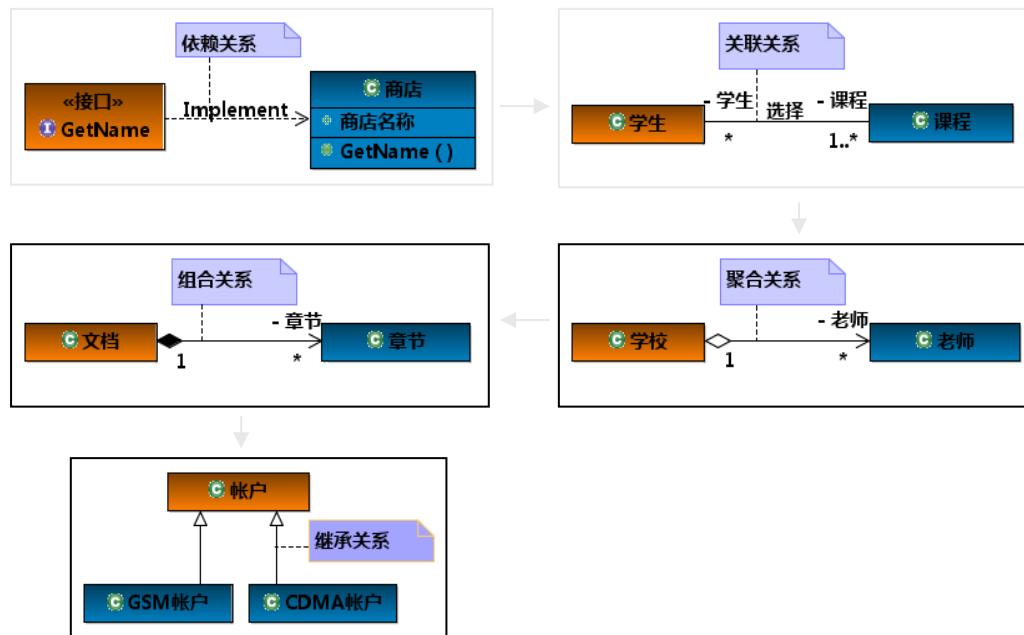
进程视图：状态图、活动图

构件视图：构件图

部署视图：部署图

6、UML 类图

- (1) UML 类图用于描述系统的静态结构——类以及类之间的关系。
- (2) 类包含三部分
 - ① 类名
 - ② 属性（可见性 属性名：类型名= 初始值 {性质串}）
 - ③ 操作（可见性 操作名（参数表）：返回值类型 {性质串}）
- (3) 类的关系
 - ① 关联【普通关联、导航关联、递归关联】
 - ② 组合和聚合
 - ③ 依赖
 - ④ 继承
- (4) 类的另一种分类：
 - ① 依赖关系：
 - 1) 类 A 把 类 B 的实例 作为 方法里的参数使用；
 - 2) 类 A 的某个方法里 使用了 类 B 的实例作为局部变量；
 - 3) 类 A 调用了 类 B 的静态方法
 - ② 关联关系
 - 1) 一个类的属性声明另一个类的对象，或者定义另一个类的引用
 - ③ 聚合关系
 - ④ 组合关系
 - 1) 注意：聚合和组合的关系
 - a. 作图时，作为整体的概念里带有菱形，作为部分的概念类带有箭头——分清整体和部分
 - b. 组合：黑菱形；聚合：白菱形
 - c. 组合：整体消失，部分也不存在
 - d. 聚合：整体消失，部分依然存在
 - ⑤ 继承关系



二、面向对象的需求分析建模

1、模型组成

包含两类模型：

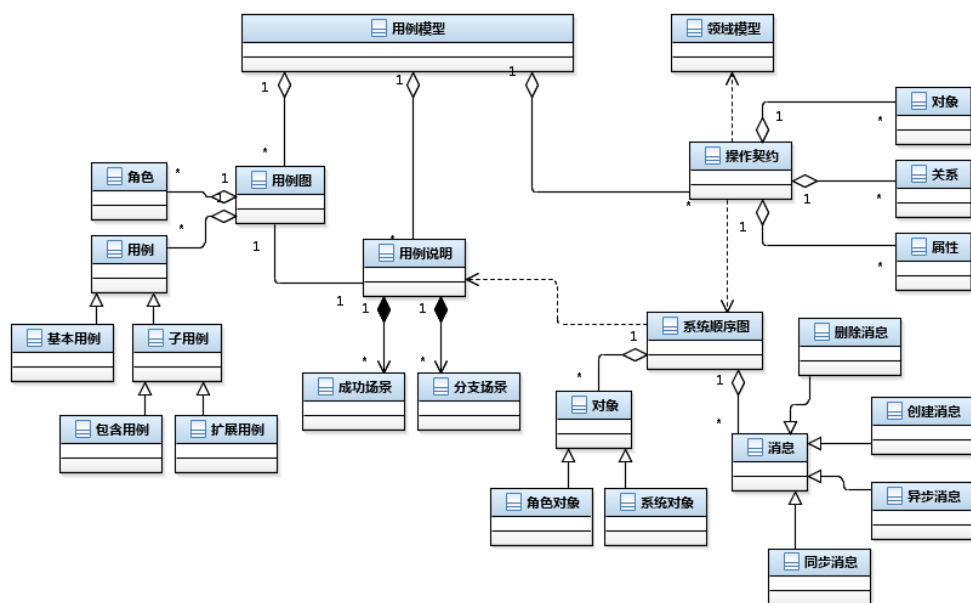
(1) 领域模型

表示了需求分析阶段“当前系统”**逻辑模型的静态结构及业务流程**，针对某一特定领域内概念类或者对象的抽象可视化表示。概括性的**描述业务背景和业务流程**。

不适用于领域模型：软件制品，例如窗口、界面、数据库、软件模型中具有职责或方法的对象。

(2) 用例模型

- ① 定义了“**目标系统**”做什么的需求。
- ② 由以下四个部分组成
 - 1) **用例图**（基础）：角色、用例、联系
 - 2) **用例说明**（基础）：成功场景和分支场景
 - 3) **系统顺序图**（附加说明）
 - 4) **操作契约**（附加说明）



2、领域模型的构建步骤

(1) 识别**候选概念类**：找出当前需求中的候选概念类
通过对**用例描述**中的**名词**或**名词短语**寻找和识别概念类；

(2) 排除某些概念类：

属性一般是可以赋值的，比如数字或者文本。如果该名词不能被赋值，那么就“有可能”是一个概念类。如果对一个名词是概念类还是属性举棋不定的时候，最好将其作为概念类处理。

(3) 添加**关联**：

- ③ 在概念类之间添加必要的关联来记录那些需要保存记忆的关系，概念之间的关系用关联、继承、组合/聚合来表示。
- ④ 关联类型：
 - 1) 须要知道型关联：将概念之间的关系信息保持一段时间的关联，需着重考虑。
 - 2) 只需理解型关联：有助于增强对领域中关键概念的理解的关联。
- ⑤ 找寻关联遵循的原则：
 - 1) 集中查找须要知道型关联；
 - 2) 识别概念类比识别关联更重要，领域模型创建应该更注重概念类的识别；
 - 3) 太多的关联容易使领域模型变得混乱；
 - 4) 避免显示冗余或导出关联；

(4) 添加**属性**：

在领域模型中描述这些概念类。用问题域中的词汇对概念类进行命名，将与当前需求无关的概念类排除在外。在概念类中添加用来实现需求的必要属性。

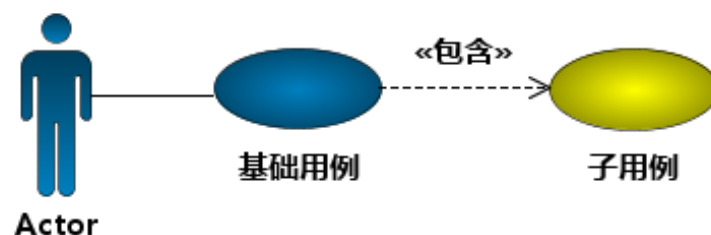
3、用例模型的创建步骤

以用例为核心从使用者的角度描述和解释待构建系统的功能需求

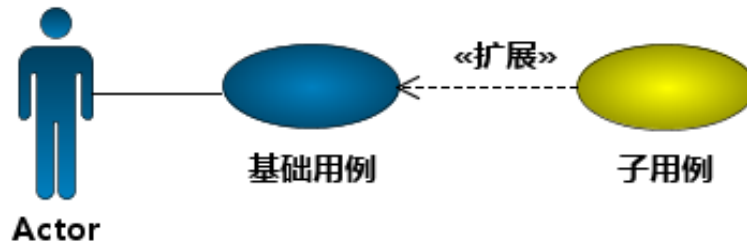
- (5) 确定**问题域的分析范围**；
 - ① 问题域：在一次用例建模过程中，需要思考的问题边界，和场景相关
 - ② 场景包括环节，环节中体现问题
- (6) 确定**该范围内可能出现的角色**；
 - ① 使用系统的使用者、和用户使用场景关联的人、
- (7) 根据业务背景或者领域模型，**确定每个角色需要的用例，并形成用例图**；
 - ① 每个角色用例不同
- (8) 基于确定的用例，整理成规范的**用例描述文本**；
 - ① 也就是用例说明
- (9) **在可能的情况下，将多个角色的用例图整合成一个相对完整的用例图**；
- (10) 针对每个用例，结合相应的用例描述，确定**系统顺序图中角色与系统之间的交互**，绘制基于用例的系统顺序图；
 - ① 系统顺序图描述角色和系统的交互过程
- (11) 基于每个系统顺序图，确定每个事件交互经过系统处理后应该返回给角色的声明性结果，即**操作契约**；

4、用例

- (12) 用例一定是系统功能，系统功能不一定是用例
- (13) **寻找用例的过程：找动词**，确定动作的目的性，概括成一个用例
- (14) 用例分类
 - ① **基本用例**：和角色直接相关的用例，表示系统的功能需求。是用户和系统直接交互形成的事件
 - ② **子用例**：通过场景描述分析归纳出的用例，也表示了系统的功能，但这些用例与角色无直接关系，间接交互，而与基本用例存在关联关系；
 - 1) **包含子用例**：多个基本用例中的某个与角色交互的场景具有相同的操作（条件 1），且这些场景都是基本用例中必须执行的步骤（条件 2）



- a. 基本用例的步骤集合中的一个子集，不会和角色产生直接关联
- b. **包含子用例的确定，必须从用户角度出发**，“用例一定是系统功能，但是系统功能不一定是用例”
- 2) **扩展子用例**：（多个）基本用例中的某些场景存在相同的条件判断的情况，可以将其抽取出来作为基本用例的子用例；



- 3) 例如：取款和查询余额：是否需要打印凭证：同样的条件判断下实现的分支场景集合。那么打印凭证可以作为取款的扩展子用例

5、用例图

1、Actor/User_case/Association

2、找动词

3、角色：使用系统的对象，代表角色可以是人、系统、设备、组织、时钟等，表示一类用户，

- (1) 通过“自问自答”(主要用户？谁会使用这个系统？谁会维护这个系统？有无与其他系统交互的情况？是否存在时钟信号？)
- (2) 用例：场景集合，场景包括成功场景和失败场景，描述用户如何成功使用系统功能(来)实现需求目标。
 - ① 如何判断：分析系统承载的日常任务、为了承载业务所需要提供的功能，系统对数据的处理？哪些事件会影响系统状态？

6、用例说明

1、组成：

用例编号:	每一个用例一个唯一的编号，方便在文档中索引。
用例名称:	(状语 +) 动词 + (定语 +) 宾语，体现参与者的目标。
范围:	应用的软件系统范围
级别:	用例/子用例
参与者:	参与者的名称
项目相关人员及其兴趣:	用户应包含满足所有相关人员兴趣的内容
前置条件:	规定了在用例中的一个场景开始之前必须为“真”的条件。
后置条件:	规定了用例成功结束后必须为“真”的条件。

主要成功场景:描述能够满足项目相关人员兴趣的一个典型的成功路径。不包括条件和分支	
1.	
.....	
.....	包含子用例的名称 或者 扩展子用例的名称
n.	
扩展（或替代流程）：（备选路径）说明了基本路径以外的所有其他场景或分支	
a.	描述任何一个步骤都有可能发生的条件，前边加
5a.	对基本路径中某个步骤的扩展描述，前边加基本路径编号
特殊需求：	与用例相关的非功能性需求
技术与数据的变化列表：	输入输出方式上的变化以及数据格式的变化。
发生频率：	用例执行的频率。
待解决的问题：	不清楚的、尚待解决的问题可集中在此进行罗列

1~n 表示路径中的环节编号

7、系统顺序图

1、对象和消息

- (1) 对象：角色对象、系统对象
- (2) 消息：创建/删除/同步/异步

2、用例描述的基础上需要进一步确定角色与系统之间的交互信息，并以可编程的方式将其命名

3、一般需要三个 UML 的符号元素：

- (1) 角色
- (2) 代表软件系统的对象
- (3) 角色与 system 之间的交互信息，简称消息或操作

4、并不是所有场景都需要，只有比较复杂或者主要的场景才需要绘制系统顺序图

8、操作契约

1、处理系统事件的操作，也称为系统事件；

2、系统顺序图上代表待构建软件系统的对象，在接收到角色所发起的系统事件请求之后，系统对象根据需求的内容所返回的一个明确的结果以及相关对象的状态，以便软件设计时进行参考

3、为系统操作而定义的，参考领域模型中业务对象接收到相同的系统事件后，执行必须的业务处理时，各业务对象的状态以及系统操作执行的结果。

4、模板

操作：	操作以及参数的名称
交叉引用：	（可选择）可能发生此操作的用例
前置条件：	执行该操作之前系统或领域模型对象的状态
后置条件：	操作完成后领域模型中对象的状态： 1、对象的创建和删除； 2、对象之间“关联”的建立或消除； 3、对象属性值的修改；

5、创建操作契约的原则：

(1) 根据系统顺序图识别进入到系统内的所有系统事件，即操作；

·针对每一个系统操作结合对应的用例领域模型，找到与此操作相关的概念类对象；

·对那些相对复杂以及用例描述中不清楚的那些系统操作按照后置条件内容确定对象的状态变化，

6、操作契约：对象、关系、属性

7、操作契约和领域模型产生关联：原因在于创建操作契约的过程中，涉及领域模型的概念类知识。

8、后置条件的陈述应该是声明性的，以强调系统状态所发生的变化，而非强调这种变化是如何设计实现的。【只说结果，不重视过程】

软件设计

1、历史

- (1) 着重点转移到软件体系结构和可用于实现软件体系结构的设计模式。
- (2) 共同特征
 - ① 一种用于将分析模型变换到设计模型的表示机制；
 - ② 用于表示功能件构件及其接口的符号体系；
 - ③ 用于求精（优化）和划分的启发机制；

2、目标

- (1) 根据“目标系统”的逻辑模型确定“目标系统”的物理模型，概括地描述系统如何实现用户所提出来的功能和性能等方面的需求
需要设计：软件结构、处理方式、数据结构、数据存储、界面和可靠性
- (2) 软件设计是软件开发的基础和依据，是软件工程过程中的技术核心
- (3) 取得主观上的最佳方案

3、过程

- (1) 将软件需求转换为功能模型、数据模型、行为模型
- (2) 分为
 - ① 概要设计：描绘总体框架
 - ② 详细设计：对结构进行细化，得到各功能模块的详细数据结构和算法，使得功能模块在细节上接近源程序的软件设计模型

4、软件的概要设计

- (1) 最终结果：生成概要设计规格说明书
- (2) 需要完成的工作：
 - ① 制定设计规范：
 - ② 软件系统结构的总体设计：
 - ③ 处理方式设计
 - ④ 数据结构设计：输入、输出、算法的数据结构
 - ⑤ 可靠性设计：容错性能、
 - ⑥ 界面设计：需求的直接表达
 - ⑦ 编写软件概要设计说明书：概要设计的最终结果
 - ⑧ 概要设计评审

5、软件详细设计

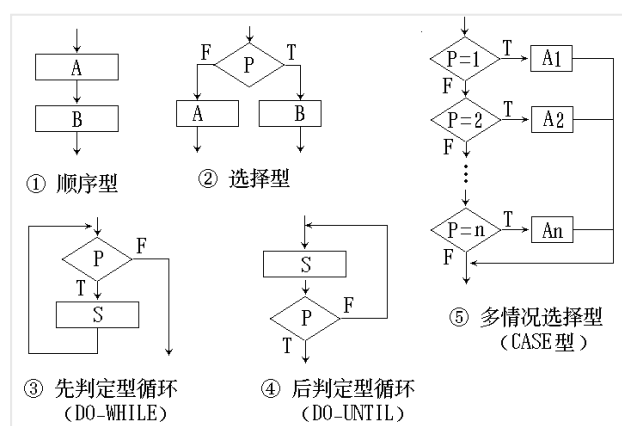
(1) 需要完成的工作：

- ① 确定软件各个功能模块内的算法以及各功能模块的内部数据组织
- ② 选定某种表达形式来描述各种算法
- ③ 编写软件详细设计说明书
- ④ 进行详细设计的评审

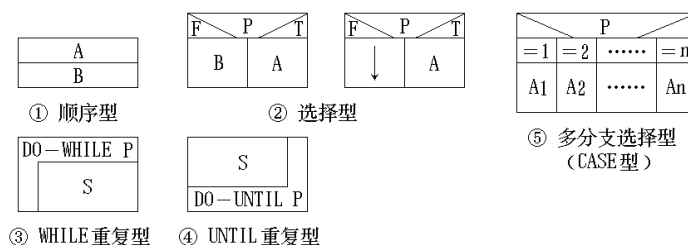
(2) 详细设计工具

① 图形

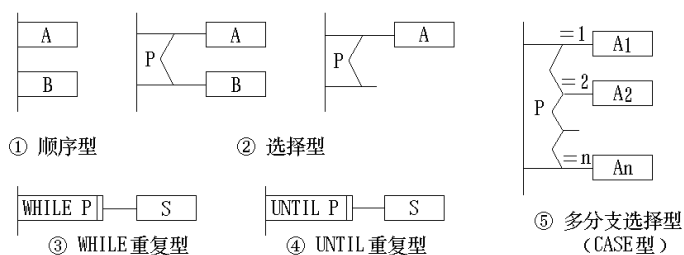
1) 程序流程图



2) N-S 图



3) PAD 图



② 表格

1) 判定表

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
X1	T	T	T	T	T	F	F	F	F	F	F	F	F	F
X2	-	-	-	-	-	T	T	T	F	F	F	F	F	F
X3	-	-	-	-	-	-	-	-	F	F	T	T	T	T
X4	-	-	-	-	-	-	-	-	F	F	-	-	-	-
X5	-	-	-	-	-	T	F	F	-	-	-	-	-	-
X6	T	T	T	F	F	-	-	-	-	-	-	-	-	-
X7	T	T	F	-	-	-	-	-	-	-	-	-	-	-
X8	T	F	-	T	F	-	T	F	F	T	T	F	T	F
a	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
b	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
c	-	-	-	-	-	Y	-	-	-	-	-	-	-	-
d	-	-	-	-	-	-	-	-	-	-	Y	Y	-	-
e	-	-	-	-	-	-	-	-	-	-	-	-	Y	Y
f	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-
g	-	-	-	Y	Y	-	-	-	-	-	-	-	-	-
h	-	-	-	Y	Y	-	-	-	-	-	-	-	-	-
i	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
j	Y	-	-	Y	-	-	Y	-	-	Y	Y	-	Y	-

③ 语言

- 1) PDL: 用于描述功能模块的算法设计和加工细节的语言

```

PROCEDURE spell_check IS
BEGIN
    split document into single words
    look up words in dictionary
    display words which are not in dictionary
    create a new dictionary
END spell_check
  
```

6、软件设计模型

- (1) 软件设计既是过程又是模型

(2) 模型构成:

- ① **静态结构设计**: 由**功能结构**和**数据结构**组成, 展现软件系统能够满足所有需求的框架结构
- ② **动态结构设计**: 表示功能响应用户请求时处理数据的**过程或条件**, 用于进一步解释软件结构中**各功能之间如何协调工作**

- (3) 软件的设计活动

- ① 系统结构设计及数据结构设计;
- ② 接口设计和过程设计;
- ③ 界面设计、组件设计及优化设计等;

- (4) **评估设计过程的技术原则:**

- ① 设计过程应该是**可追踪和可回溯**的
- ② 实现分析模型中描述的所有**显式需求**
- ③ 满足用户希望的所有**隐式需求**【安全性、功能性错误、数据安全、完整性要求】
- ④ **设计说明文档**: 可读的、可理解的、易于编程、测试、维护

- (5) **评估设计模型的技术原则**

- ① **设计模型应该展现软件的全貌**, 包括从实现角度可看到的数据、功能、行为。
- ② 设计模型应该是一个**分层结构**。该结构:
 - 1) 使用可识别的设计模式搭建系统结构;
 - 2) 由具备良好设计特征的构件构成; ——模块设计

3) 可以用演化的方式实现；——容纳变化

- ③ 设计应当**模块化**，即应当建立具有独立功能特征的构件。——软件设计的模块化和问题的分解相互呼应。
- ④ 设计应当建立能够**降低**模块与外部环境之间**复杂连接的接口**。——接口过多不利于系统稳定性、不利于开发和维护
- ⑤ 设计应当根据将要实现的对象和数据**导出合适的数据结构**。

软件模块化

- ⑥ 模块的定义：整个软件可被划分成若干个**可单独命名且可编址组成部分**，这些部分称之为模块。
- ⑦ 模块的基本属性：**功能、逻辑、状态**
- ⑧ 模块的表示：
 - 1) 外部特性：模块名、参数表、给程序甚至整个系统造成的影响
 - 2) 内部特性：完成功能的程序代码和仅供该模块内部使用的数据
- ⑨ 模块划分（**自顶向下**）
难点在于合理地划分模块
 - 1) 如果模块是相互独立的（模块独立性），当模块变得越小，每个模块花费的工作量越低
 - 2) 但当模块数增加时，模块间的联系也随之增加（模块的耦合度），把这些模块联接起来的工作量也随之增加。模块之间的关系能够达到**信息隐藏**，就可以认为是合理的模块划分。

(6) 模块设计标准

- ① 模块可分解性：可将系统按“问题 / 子问题”的分解原则分解成系统的模块层次结构；
- ② 模块可组装性：可利用已有的设计构件组装成新系统，不必一切从头开始。
- ③ 模块可理解性：一个模块可不参考其他模块而被理解；
- ④ 模块连续性：对软件需求的一些微小变更只导致对某个模块的修改而整个系统不用大动；
- ⑤ 模块保护：将模块内出现异常情况的影响范围限制在模块

(7) 抽象化

- ① **过程抽象**：
 - 1) 软件计划阶段：软件被看作是一个相对宏观的系统元素
 - 2) 软件需求分析阶段：用“问题所处环境的为大家所熟悉的术语”来描述软件的解决方法。
 - 3) 概要设计阶段：使用**规定的符号**表示软件的轮廓和结构
 - 4) 详细设计阶段：使用**面向代码的符号**表示软件的内部结构
- ② **数据抽象**：数据抽象与过程抽象一样，允许设计人员在不同层次上描述数据对象的细节。
- ③ **控制抽象**：可以包含一个程序控制机制而无须规定其内部细节。

(8) 模块独立性

- ① 功能独立性是抽象、模块化和信息隐藏的直接产物。
- ② 如果一个模块能够**独立于其他模块被编程、测试和修改**，而和软件系统中其它

的模块的接口是简单的，则该模块具有功能独立性。

③ **两个度量准则：**

- 1) 模块间的耦合
- 2) 模块的内聚

模块内聚

- ① 定义：**模块功能强度的度量**。一个模块内部各元素之间的联系越紧密，则它的内聚性就越高。
- ② **联系紧密程度、内聚性、独立性成正比，内聚性和耦合性成反比**
- ③ **判断方法**：如果一个模块或者一个类，它完成的任务相同或者相似，则该模块或者类的内聚性比较高
- ④ 内聚分类：
 - 1) **巧合内聚**
 - a. 当几个模块内**凑巧**有一些**程序段代码相同**，又没有明确表现出独立的功能，为了**减少存储把这些代码独立出来**建立一个新的模块，这个模块就是巧合内聚模块。
 - b. 它是**内聚程度最低**的模块。——绝对避免
 - 2) **逻辑内聚**
 - a. 把几种相关的**功能组合**在一起，每次被调用时，由传送给模块的**判定参数**来确定该模块应执行哪一种功能。
 - b. **该内聚的缺陷**：
 - a) 调用模块需要的功能很简单，但是调用另外一个功能模块时，编译未用到的功能
 - b) 违背了“消息隐藏”的原则
 - c. **尽力避免**
 - 3) **时间内聚（经典内聚）**
 - a. 含义：这种模块一般为**多功能模块**，但模块的各个功能的执行与时间有关，**通常要求所有功能必须在同一时间段内执行**。
 - b. 内聚性相比于前两个较好，但是并不理想
 - 4) **过程内聚**
 - a. 使用流程图做为工具设计程序时，把**流程图中的某一部分**划出组成模块，就得到过程内聚模块。
 - b. 要求：程序执行有严格的时间顺序
 - 5) **通信内聚**
 - a. 含义：一个模块内各功能部分都**使用了相同的输入数据，或产生了相同的输出数据**，则称之为通信内聚模块。
 - b. 判定：通信内聚模块可通过**数据流图**来判定。
 - 6) **信息内聚**
 - a. 含义：模块**完成多个功能**，各功能都在**同一数据结构**上操作，每一项功能有一个唯一的入口点。
 - b. 聚合度高
 - 7) **功能内聚**

- a. 含义: 一个模块中各个部分都是完成某一具体功能必不可少的组成部分, 或者说该模块中所有部分都是为了完成一项具体功能而协同工作, 紧密联系, 不可分割的。则称该模块为功能内聚模块。
- b. 聚合度很高, 很难达到

模块耦合性

- ① 定义: 模块之间互相连接的紧密程度的度量。
- ② 接口复杂程度、调用模块的方式、模块数量——都影响了模块的耦合程度
- ③ 耦合度过高的影响: ??、整合需要花费更多的时间、很难复用和测试
- ④ 耦合分类
 - 1) 内容耦合 (耦合度最高)
 - a. 一个模块直接访问另一个模块的内部数据, 不通过正常入口转到另一模块内部、两个模块有一部分程序代码重叠(汇编语言中)、一个模块有多个入口。
 - 2) 公共耦合
 - a. 定义: 一组模块都访问同一个公共数据环境
 - b. 公共的数据环境可以是全局数据结构、共享的通信区、内存的公共覆盖区等。
 - a) 公共耦合的复杂程度随耦合模块的个数增加而显著增加。
 - b) 只是两模块间有公共数据环境, 则公共耦合有两种情况。松散公共耦合和紧密公共耦合。
 - c) 公共耦合度高引发的问题:
 - i. 所有公共耦合模块都与某一个公共数据环境内部各项的物理安排有关, 若修改某个数据的大小, 将会影响到所有的模块。
 - ii. 无法控制各个模块对公共数据的存取, 严重影响软件模块的可靠性和适应性。
 - iii. 公共数据名的使用, 明显降低了程序的可读性。
 - 3) 外部耦合
 - a) 一组模块都访问同一全局简单变量而不是同一全局数据结构, 而且不是通过参数表传递该全局变量的信息
 - b) 和公共耦合的区别: 单一全局变量, 而不是公共数据环境
 - 4) 控制耦合
 - a) 一个模块通过传送开关、标志、名字等控制信息, 明显地控制选择另一模块的功能
 - b) 控制模块 + 被控制模块
 - c) 在单一接口上选择多功能模块的功能
 - d) 控制模块必须知道所控制模块内部的一些逻辑关系, 降低了模块的独立性。——违背了“信息隐藏”的准则
 - 5) 标记耦合
 - a) 一组模块通过参数表 (复杂的数据结构) 传递记录信息

- b) 把数据结构上的操作全部集中在一个模块中，来消除这种耦合。

6) 数据耦合

- a) 一个模块访问另一个模块时，彼此之间是通过**简单数据参数（不是控制参数、公共数据结构或外部变量）**来交换输入、输出信息的
- b) 松散耦合，模块间的独立性强

7) 非直接耦合

- a. 耦合度最低，独立性最强，很难达到
- b. 两个模块之间没有直接关系，它们之间的联系完全是通过主（上级）模块的控制和调用来实现的

⑤ 耦合度计算公式

$$\bullet \text{ Coupling}(C) = 1 - \frac{1}{d_i + 2 \times c_i + d_o + 2 \times c_o + g_d + 2 \times g_c + w + r}$$

- a.
- b. 值越大，模块之间的耦合度越大
- c. 经验范围：0.67（低耦合）~ 1.0（高耦合）
- d. 参数说明：
 - a) 针对数据参数和控制参数的耦合项：
 - i. d_i ：数据参数的输入个数；
 - ii. c_i ：控制参数的输入个数；
 - iii. d_o ：数据参数的输出个数
 - iv. c_o ：控制参数的输出个数；
 - b) 针对全局耦合项：
 - i. g_d ：作为数据的全局变量个数；
 - ii. g_c ：作为控制的全局变量个数；
 - c) 针对环境耦合项：
 - i. w ：调用的模块数，也称为扇出数；
 - ii. r ：调用的模块数，也称为扇入数。

⑥ 降低耦合度

- a. 根据问题的特点**选择适当的耦合类型**
 - a) 模块间的信息传递：数据信息和控制信息的选择；
 - b) 模块间的调用方式：传送地址和传送判定参数的选择；
 - c) 系统的错误处理模块：集中处理与分散处理的选择；
- b. **降低模块接口的复杂度**
 - a) 传送信息的数量：参数多和参数少的区别；
 - b) 模块的调用方式：简单调用和直接引用的区别；
- c. 将**模块的通信信息放在缓冲区**中

面向对象的设计原则

(1) 单一职责 (Single Responsibility)

- ① 定义：针对**类**，应该只有一个引起它变化的原因——“职责”，

- ② 如果有其他原因去改变一个类，那么这个类就具有其他的职责。**类具有多个职责，等于这些职责具有耦合关系。**
- ③ 为了提高类的内聚度，**应将对象的不同职责分离至两个或多个类中**，确保引起该类变化的原因只有一个。
- ④ **一个类应该只完成一个职责**，如果完成多个职责，会影响功能执行

(2) 开闭原则 (Open Closed)

- ① 软件实体（类、模块、函数）**可以扩展、不能修改**
 - 1) **对扩展开放，对更改封闭**——需求改变时进行代码拓展而不是代码修改
 - 2) OCP 实现的关键是使用抽象，强调对变化的类进行抽象

(3) 里氏替换原则 (Liskov Substitution) (LSP)

- ① **子类应当可以替换父类并出现在父类能够出现的任何地方。**
- ② 原始定义：若对于每一个类型 S 的对象 o1，都存在一个类型 T 的对象 o2，使得在所有针对 T 编写的程序 P 中，用 o1 替换 o2 后，程序 P 的行为功能不变，则 S 是 T 的子类型。
- ③ **所有子类的行为功能必须和客户类对其父类所期望的行为功能保持一致。**
- ④ 强调了实现抽象化的具体规范。

(4) 依赖倒置原则 (Dependency Inversion)

- ① **高层模块不应依赖于低层模块，二者都应依赖于抽象；**
- ② **细节依赖于抽象**
- ③ **传统的模块间的调用关系与该原则正好相反**，低层模块的改动使得高层模块不得不随之改动。
- ④ 程序中所有的依赖关系应该终止于**抽象类或者接口**：
 - 1) 任何变量都不应该持有一个指向具体类的指针或者引用；
 - 2) 任何类都不应该从具体类派生，或者说继承自一个具体类；
 - 3) 任何方法都不应该覆盖它的任何基类中的已经实现的方法。
- ⑤ 每个较高层次都为它所需要的服务声明一个抽象接口，较低的层次实现了这些抽象接口
- ⑥ **每个层次通过一个定义良好的、受控的接口向外提供了一组内聚的服务**

(5) 接口隔离原则 (Interface Segregation)

- ① 采用**多个与特定客户类的接口** 比 采用一个通用的涵盖多个业务方法的接口要好
- ② 服务器类应该为每一个客户类创建特定的业务接口，而不要为所有客户类提供统一的业务接口
- ③ **接口隔离——多个特定接口，而不是统一接口**

(6) 组合/聚合复用 (Composite/Aggregation Reuse)

- ① 在一个新对象里面使用一些已有对象，使之成为新对象的一部分；新对象通过向已有对象委托 (delegate) 一部分责任而达到复用已有对象的目的。
- ② 继承复用

继承复用

优点：简单易用

缺点：

- 1、破坏封装性；
- 2、LSP原则得不到保障；
- 3、父类更改导致子类必须更改；
- 4、继承的实现为静态，缺乏灵活性
- 5、依赖于相同的上下文环境；

③ 组合/聚合复用

组合/聚合复用

将已有对象组合/聚合为一个新对象的方式实现对已有对象行为功能的复用

优点:

- 1、支持封装性, 实现黑盒复用;
- 2、新对象符合SRP原则;
- 3、通过接口实现动态引用;
- 4、不依赖于上下文;

缺点:

- 1、将已有对象扩充到新对象中比较困难;
- 2、会产生大量新对象, 管理困难;

(7) 迪米特法则 (Law of Demeter)

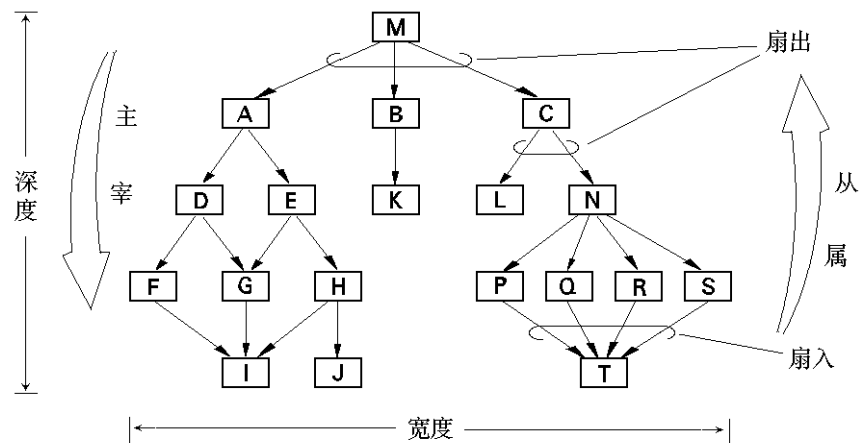
- ① 最少知识原则: 一个对象应当可能少的了解其它对象。
- ② 只和某些对象进行沟通:
 - 1) 当前对象本身 (this);
 - 2) 以参量形式传入到当前对象方法中的对象;
 - 3) 当前对象的实例变量直接引用的对象;
 - 4) 当前对象的实例变量如果是一个聚集, 那么聚集中的元素也都是朋友;
 - 5) 当前对象所创建的对象。

软件设计基础

(1) 自顶向下, 逐步细化

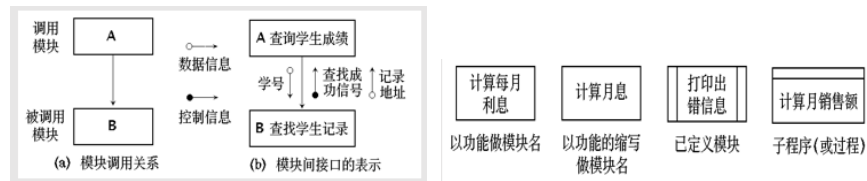
(2) 系统控制结构

表明了模块的组织情况



(3) 结构划分和结构图

- ① 水平划分
按主要的程序功能来定义模块结构的各个分支
- ② 垂直划分 (因子)
 - 1) 用于程序的体系结构中
 - 2) 一般来说效果更好
- ③ 功能结构图
 - 1) 模块用矩形框表示, 用模块名来标记



(4) 数据结构

应当确定数据的组织、存取方式、相关程度、以及信息的不同处理方法

(5) 软件过程

- ① 对模块内部细节的实现机制进行详细的描述
- ② 包括事件的顺序、正确的判定点、重复的操作直至数据的组织和结构提供精确的处理说明
- ③ 程序结构和软件过程有关
- ④ 软件过程是层次化的

软件体系结构

(1) 根本目的

要解决软件重用、软件质量、软件维护问题

(2) 定义

- ① Booch: 软件体系结构 = {组织, 元素, 子系统, 风格}
- ② Bass: 系统的一个或多个结构, 包括软件构件 (Components)、构件的外部可视属性 (Properties)、构件之间的关系 (Relationships)
- ③ Shaw: 结构模型、框架模型、动态模型、过程模型
- ④ Garlan & Shaw 模型: 软件体系结构 = {构件, 连接件, 约束}

(3) 三要素

- ① 程序构件 (模块) 的层次结构
- ② 构件之间交互的方式
- ③ 数据的结构

(4) 软件构件的分类和调用方式

构件	特点和示例	连接	特点与示例
纯计算构件	具有简单的输入/输出关系, 没有运行状态的变化。例如, 数值计算、过滤器 (Filters)、转换器 (Transformers) 等。	过程调用	在某一个执行路径中传递执行指针。例如, 普通过程调用 (同一个命名空间)、远程过程调用 (不同的命名空间)。
存储构件	存放共享的、永久性的、结构化的数据。例如, 数据库、文件、符号表、超文本等。	数据流	相互独立的处理通过数据流进行交互, 在得到数据的同时被赋予控制权限。例如, UNIX系统中的管道 (pipes)。
管理构件	执行的操作与运行状态紧密耦合。例如, 抽象数据类型 (ADT)、面向对象系统中的对象、许多服务器 (Servers) 等。	间接激活	处理是因事件的发生而激活的, 在处理之间没有直接的交互。例如, 事件驱动系统、自动垃圾回收等。
控制构件	管理其它构件运行的时间、时机及次序。例如, 调度器、同步器等。	消息传递	相互独立的处理之间有明确的交互, 通过显式的离散方式的数据传递。这种传递可以是同步的, 也可以是异步的。例如, TCP/IP。
链接构件	在实体之间传递信息。例如, 通信机制、用户界面等。	共享数据	构件们通过同一个数据空间进行开发的操作。例如, 多用户数据库、数据黑板系统。

(5) 软件体系结构风格

- ① 描述某一特定应用领域中系统组织方式的惯用模式

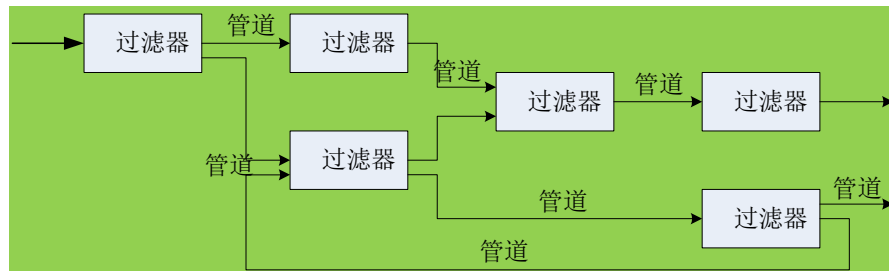
② **核心问题**：是能否使用重复的体系结构模式。

③ 四要素：

- 1) 提供一个词汇表
- 2) 定义一套配置规则
- 3) 定义一套语义解释原则
- 4) 定义对基于这种风格的系统所进行的分析

④ **管道和过滤器风格**

- 1) 最早出现在 Unix 系统中，它适用于对有序数据进行一系列已经定义的独立计算的应用程序
- 2) 构件：过滤器（filter）。【过滤器之间彼此独立】
- 3) 对输入流进行处理、转换，处理后的结果在输出端流出。而且这种计算处理方式是递进的，所以可能在全部的输入接受完之前就开始输出。系统中可以并行地使用过滤器。
- 4) 连接件：连接件位于过滤器之间，起到信息流的导管作用，被称为管道（pipe）
- 5) 不适合处理交互性数据应用



⑤ **调用和返回风格**

- 1) 调用/返回风格的体系结构在过去的 30 年之间占有重要的地位，是大型软件开发中的主流风格的体系结构。
 - a. 主/子程序风格的体系结构——面向过程
 - b. 对象风格的体系结构——面向对象
 - c. 分层风格的体系结构

⑥ **基于事件的风格**

- a. 构件不直接调用一个过程，而是声明或广播一个或多个事件。
- b. 适用于设计低耦合构件集合的应用程序，每个构件完成一定的操作，并可能触发其他构件的操作。
- c. 构件的接口不仅提供一个过程的集合，也提供一个事件的集合。这些过程既可以用一般的方式调用，也可能被注册为与某些事件相关。
- d. 构件可以声明或广播一个或多个事件，或者向系统注册用以表明它希望响应一个或多个事件。

⑦ **客户端/服务器风格**

- a. 目标是达到可测量性的需求，并适用于应用程序的数据和处理分布在一定范围内的多个构件上，且构件之间通过网络连接。

⑧ **解释器/虚拟机风格**

- 1) 建立一种虚拟机去弥合程序的语义与作为计算引擎的硬件的差异。
- 2) 适用于应用程序不能直接运行在最合适的机器上或不能直接以最适合的语言执行

⑨ 仓库风格

- 1) 体系结构由两种构件组成：
 - a. 中央数据结构，表示当前状态；
 - b. 独立构件的集合，它对中央数据结构进行操作。
- 2) 传统的数据和状态控制方法：输入事务选择进行何种处理
- 3) 黑板体系结构：中央数据结构的当前状态决定进行何种处理

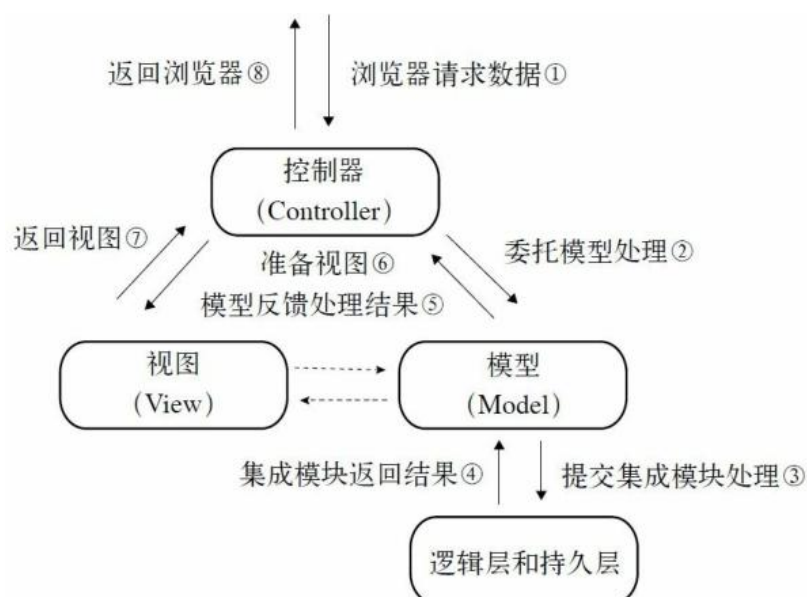
⑩ 黑板风格

- 1) 用于在信号处理方面进行复杂解释的应用程序、松散的构件访问共享数据的应用程序
- 2) 知识源：特定应用程序知识的独立散片
 - a. 知识源之间的交互旨在黑板内部发生
- 3) 黑板数据结构
- 4) 控制器

11 MVC 风格——使用较多

a. 模型-视图-控制器

- b. **视图**：为用户显示模型信息。视图从模型获取数据，一个模型可以对应多个视图。
- c. **模型**：模型是应用程序的核心，它封装内核数据与状态。对模型的修改将扩散到所有视图中。所有需要从模型中获取数据的对象都必须注册为模型的视图。
- d. **控制器**：是提供给用户进行操作的接口。每个视图与一个控制器构件相关联。控制器接受用户输入，输入事件转换成服务请求，传送到模型或视图。用户只通过控制其与系统进行交互
- e. 目的是将 Model 和 View 的实现代码分离，从而使同一个程序可以使用不同的表现形式。Controller 存在的目的则是确保 M 和 V 的同步，一旦 M 改变，V 应该同步更新。



面向对象设计方法

1、面向对象设计综述

- (1) **面向对象的设计**：以需求分析阶段的用例模型和领域模型为基础，运用 UML 构建软件系统结构，通过一系列设计模型说明用例的实现过程。
- (2) 设计活动
 - ① 选择合适的软件架构
 - ② 根据架构通过 UML 交互图描述每个用例的实现过程
 - ③ 给出以 UML 类图表示的能够满足所有用例的系统静态结构
 - ④ 根据系统的设计原则进行优化
- (3) 软件概要设计步骤
 - ① 选择合适的软件架构
 - ② 系统的动态结构设计
 - ③ 系统的静态结构设计
- (4) 软件详细设计
- (5) 面向对象设计的关键步骤
 - ① 发现对象（发现软件类）
 - ② 确定对象属性
 - ③ 确定对象行为
 - ④ 确定对象之间的关系

2、模型层次化（基于 B/S 结构）

- (1) **用户界面层**
 - ① 系统功能的各种界面表现形式
 - ② 尽量将用户界面层与系统的业务逻辑分离，专门处理系统与用户的交互
- (2) **控制器层**
 - ① 角色发给对象的请求，由控制器进行处理
 - ② 处理不同用例的系统事件的对象，称为控制器
 - ③ 协调控制其他类共同完成用例规定的功能或者行为
- (3) **业务层/应用层**
 - ① 尽量参考领域模型中的**概念类**对软件对象进行定义和命名
 - ② 概念类和软件对象存在对应关系，实现用例对应的核心功能
 - ③ 业务层主要用来实现用例要求的各种系统级功能
- (4) **持久化层**
 - ① 对于数据进行存储管理
 - ② 降低耦合性
 - ③ 对象持久化：将对象状态永久保存到物理存储介质中
- (5) **系统层**
 - ① 将操作系统提供的系统调用封装起来，生成系统访问类
 - ② 使软件与操作系统分离

3、设计用例实现方案

- (1) 面向对象设计--主要考虑用例如何实现--用例实现设计
- (2) 用例实现: 在设计模型中描述分层结构中相互协作的软件对象如何实现用例的各个特定场景 (用例), 包括所有的成功和失败场景
- (3) 用例实现的设计方案: UML 的 sequence/collaboration diagram 进行绘制
- (4) 面向对象的设计模式

① 对象的职责通过调用对象的方法来实现

② 最关键的活动是正确地给对象分配职责

③ 模式

- 1) 是面向对象软件的设计经验, 是可重用的设计思想
- 2) 定义了一组相互协作的类, 包括类的职责和类之间的交互方式
- 3) 组成:
 - a. 名称
 - b. 问题: 问题域, 何时解决
 - c. 解决方案: 设计的组成部分、组成部分的相互关系、各自的职责和协作方式
 - d. 效果: 模式应用的效果和使用模式应该权衡的问题

④ 类职责分配模式 (GRASP)

- 1) 设计类的来源:
 - a. 领域模型中的概念类
 - b. 新增类: 负责对象持久化的类、负责通信的类
- 2) 设计类的职责类型
 - a. knowing 了解型: 【自己干自己的事】
 - b. doing 行为型: 【自己干自己能干的事】
- 3) 职责内聚: 职责的内聚【自己只干自己的事】: 目的是提高内聚降低耦合, 减少不必要的关联关系

• GRASP的9个模式

1. (信息)专家
2. 创建者
3. 底耦合
4. 高内聚
5. 控制者
6. 多态
7. 纯虚构
8. 中介者
9. 受保护化

4) 控制器模式 (controller)

- a. 把接收和处理系统事件的职责分配给位于控制器层的对象
- b. 在相同的用例场景中使用同一个控制器类处理所有的系统事件;
- c. 指导原则: 不论是外观控制器还是用例控制器, 它们只是接收系统事件消息, 并没有实现系统操作的职责, 系统操作应该委托给领域对象处理

5) 创建者模式 (creator)

- a. 如果符合下面的一个或者多个条件，则可将创建类 A 实例的职责分配给类 B (B 创建 A)。
 - a) B 聚合 (aggregate) 或包含 (contain) 对象 A;
 - b) B 记录 (record) 对象 A;
 - c) B 密切使用对象 A;
 - d) B 拥有创建对象 A 所需要的初始化数据 (B 是创建对象 A 的信息专家)。
- b. 创建者模式体现了**低耦合**的设计思想，是对迪米特法则的具体运用。

6) 信息专家模式 (Information Expert)

- a. 给对象分配职责的通用原则: 将职责分配给拥有履行职责所必需信息的类，即信息专家。换言之，对象具有处理自己拥有信息的职责或能力。
- b. 根据信息专家模式，应该找到拥有履行职责所必须的信息的类，
- c. 选取类的方法：
 - a) 如果在设计模型中存在相关的类，先到设计模型中查看;
 - b) 如果在设计模型中不存在相关的类，则到领域模型中查看，试着应用或扩展领域模型，得出相应的设计类。
- d. 职责的实现（即功能）需要信息，而信息往往分布在不同的对象中，一个任务可能需要多个对象（信息专家）协作来完成。

(5) 动态结构设计

- ① 输入条件：用例，SSD，操作契约及领域模型
- ② 输出结果：用例的一系列交互图，展示并证明系统如何执行用例的过程

(6) 静态结构设计（用例级别）

- ① 输入条件：用例的动态结构
- ② 输出结果：用例级别的静态类图
- ③ 系统级静态结构：对每个用例的静态类图进行扫描，去除重复出现的软件类，修改并确定软件类层次之间以及同层软件类之间的关系

软件实现

- 软件实现是软件详细设计的后续阶段及任务，即**程序编码**；
- 程序编码需要根据具体情况条件确定具体的程序设计语言；
 - 参考应用领域；
 - 根据用户的要求；
 - 参考现有的工具及环境；
 - 程序员的能力水平；
 - 可移植性的要求；
- 按照详细设计及概要设计的要求转换成选定的编程语言；
- 源程序文档化；
- 注释：
 - 序言性注释：位于程序代码之前，说明该模块（类及方法）具体作用；
 - 功能性注释：对于程序体中复杂难于理解的程序结构进行局部说明；
- 进一步按照要求进行必要的软件单元测试，使可执行程序达到软件的质量要求。

软件测试

1、目的和原则

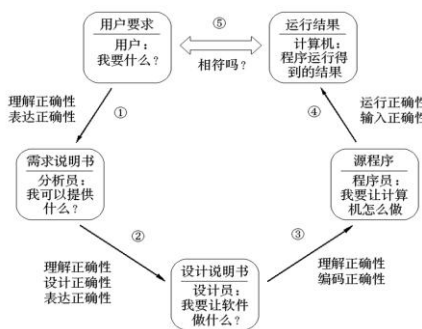
1. Glenford J.Myers：软件测试目的

- 测试是程序的执行过程，目的在于发现错误；
- 一个好的测试用例在于能发现至今未发现的错误；
- 一个成功的测试是发现了至今未发现的错误的测试；
- 测试不能表明软件中不存在错误，只能说明软件中存在错误。

2. 测试用例 = 测试输入数据 + 对应的预期输出结果

3. 软件测试对象

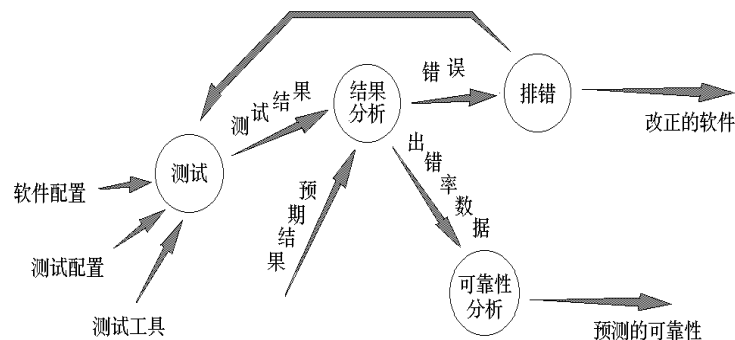
- (1) 软件测试并不等于程序测试，软件测试应贯穿于软件定义与开发的整个周期
- (2) 软件开发各阶段所得到的文档都应成为软件测试的对象



用例编号	XT-YH-0001	用例名称	增加用户-合法输入
测试功能	增加用户	测试目的	在各项用户数据均合法的情况下，系统能够正确添加一个用户。
业务说明	此处给出该系统在增加用户时，各项输入字段的限制要求，以及系统处理的业务规则。		
前置条件	此处说明进行该功能操作时，所需要的数据环境、登录用户所需具有权限等说明		
操作过程	1. 执行“系统管理”下的“用户管理”，出现用户管理页面； 2. 单击“用户管理”页面中的“新增用户”按钮，弹出增加用户窗口； 3. 在“增加用户”窗口中输入用户各项信息，每个字段均符合业务说明中的限制要求； 4. 单击“确定”按钮。		
预期结果	系统提示“增加用户成功”，在返回的“用户管理”页面的用户列表中，显示出新增用户信息。		
其他说明	无。		
用例设计人	张三	用例审核人	李四

4. 软件测试与软件各阶段的关系

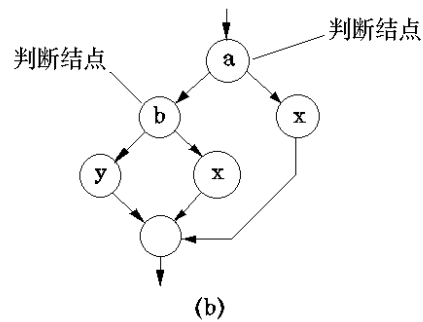
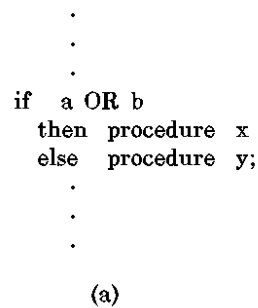
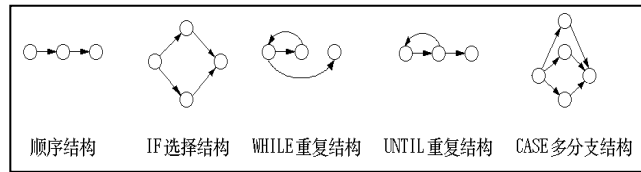
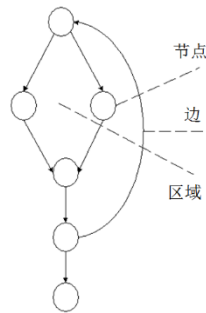
软件测试是软件实现之后开始的一系列测试活动



2、测试方法

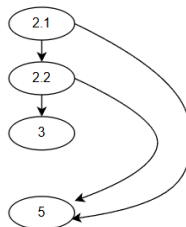
1. 白盒测试

- (1) 定义：将测试对象看作一个透明的盒子，允许利用程序内部的逻辑结构及有关信息，设计或选择测试用例，**对程序所有逻辑路径进行测试**。通过在不同点检查程序的状态，确定实际的状态是否与预期的状态一致，又称为结构测试或逻辑驱动测试。
- (2) 主要应用于单元测试，是检查程序逻辑错误的主要方法
- (3) 检查程序
 - ① 程序模块的所有独立的执行路径至少测试一次；
 - ② 对所有的逻辑判定，取“真”与取“假”的两种情况都至少测试一次；
 - ③ 在循环的边界和运行界限内执行循环体；
 - ④ 测试内部数据结构的有效性。
- (4) 逻辑覆盖
 - ① 语句覆盖
每一个可执行语句至少执行一次
 - ② 判定覆盖（分支）
每个判断的取真分支和取假分支至少经历一次
 - ③ 条件覆盖
每个判断的每个条件的可能取值至少执行一次
 - ④ 判定-条件覆盖
 - ⑤ 条件组合覆盖
使得每个判断的所有可能的条件取值组合至少执行一次
 - ⑥ 路径覆盖
设计足够的测试用例，覆盖程序中所有可能的路径
- (5) 基本路径测试
 - ① 控制流图

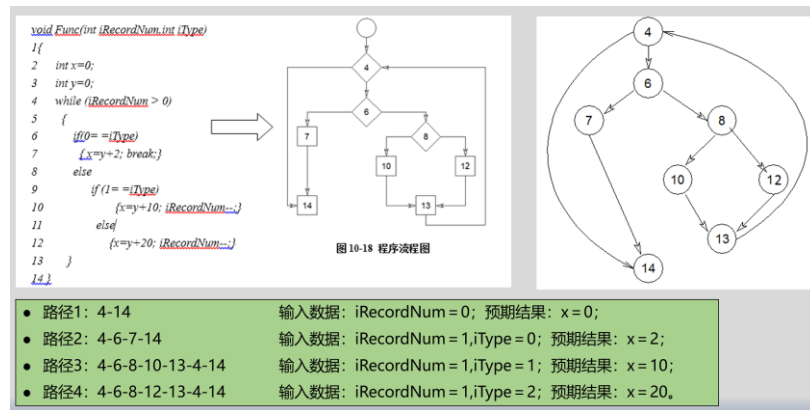


```

1  int test(int a, int b) {
2      if (a < 10 && b > 50) {
3          return a + b;
4      }
5      return a - b;
6  }
  
```



- ② 环路复杂度 (McCabe 复杂度)
- 1) 确定了程序中独立路径的上界
 - 2) 计算方法:
 - a. 等于控制流图中的 **区域数**, 包括封闭区域和开放区域;
 - b. 设 E 为控制流图的边数, N 为图的结点数, 则定义环路复杂性为 $V(G) = E - N + 2$;
 - c. 若设 P 为控制流图中的判定结点数, 则有 $V(G) = P + 1$
 - 3) 基本路径集: 程序的控制流图中从入口到出口的路径, 该路径至少经历一个从未走过的边 (不唯一, 最大的基本路径条数就是环路复杂度)
- ③ 控制流图转换



2. 黑盒测试

- (1) 定义：完全不考虑程序内部的逻辑结构和内部特性，只依据程序的**需求规格说明书**和**概要设计说明**，检查程序的功能是否符合它的功能说明，又称为功能测试或数据驱动测试
- (2) 方法
 - ① 等价类划分；
 - ② 边界值分析；
 - ③ 因果图；
- (3) 一般用于集成测试、系统测试和验收测试，某些特殊情况也会用到单元测试
- (4) 用于测试程序接口
- (5) 等价类
 - ① 划分：根据输入域的要求和数据类型定义寻找等价类，确定等价类表结构
 - ② 分类
 - 1) 有效（合理）
 - 2) 无效
 - ③ 划分原则
 - 1) 区间
 - 2) 数据集合
 - 3) 布尔 -> 真假
 - 4) 数值
 - 5) 限制条件/规则
- (6) 测试用例
 - ① 为每一个等价类规定一个唯一编号；
 - ② 设计一个新的测试用例，使其尽可能多地覆盖尚未被覆盖的有效等价类，重复这一步，直到所有的有效等价类都被覆盖为止；
 - ③ 设计一个新的测试用例，使其仅覆盖一个尚未被覆盖的无效等价类，重复这一步，直到所有的无效等价类都被覆盖为止。
- (7) 等价类划分表
 - ① 输入条件：表示等价类的种类
 - ② 有效等价类
 - ③ 无效等价类
 - ④ 编号

输入条件	有效等价类	编号	无效等价类	编号
工作证号	整数	1	非整数	11
	[1,5000]	2	<1	12
			>5000	13
姓名	中文字符	3	包含非中文字符	14
	不超过 20 个汉字	4	超过 20	15
	不能为空	5	空	16
密码	长度大于等于 6	6	长度为 6 位以下	17
	必须包括数字和字母	7	只包含数字	18
			只包含字母	19
			不包含数字和字母	20
参加工作时间	8 位	8	不是 8 位	21
	数字	9	出现非数字	22
	YYYYMMDD	10	YYYY<1	23
			MM>12	24
			MM<1	25
			DD>31	26
			DD<1	27

(8) 边界值分析

- ① 对等价类划分方法的补充
- ② 选取正好等于/刚刚大于/刚刚小于边界的值测试

(9) 因果图

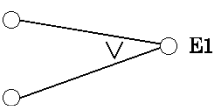
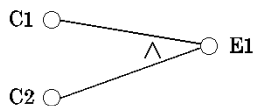
① 步骤

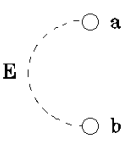
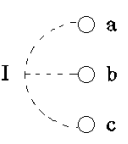
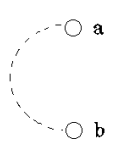
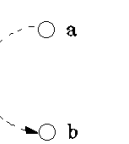
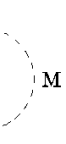
- 1) 分析（需求）软件规格说明描述中，哪些是原因（输入或状态），哪些是结果（输出或动作），并给每个原因和结果赋予一个唯一的标识符。
- 2) 分析（需求）软件规格说明中的语义，找出原因与原因之间，原因与结果之间的关系，根据这些关系，画出因果图。
- 3) 由于语法或环境限制，有些原因与原因之间，原因与结果之间的组合情况不可能出现。为表明这些特殊情况，在因果图上用一些记号标明约束或限制条件。
- 4) 把因果图转换成判定表，并根据因果图中的制约关系对判定表进行化简，去掉不可能存在的组合情况。
- 5) 简化后的判定表中的每一列就是一种有效的条件组合，对应一个测试用例。

② 符号

- 1) 通常在因果图中用 C_i 表示原因，用 E_i 表示结果，各结点表示状态
- 2) 0 不出现；1 出现

(a) 恒等 ○ ———— ○ E₁ (b) 非 C₁ ○ ———— / \ ———— ○ E₁

(c) 或  E₁ (d) 与  E₁

 E
  I
  O
  R
  M
 (1) E (互斥·排他) (2) I (包含·或) (3) O (唯一) (4) R (要求) (5) M (屏蔽)

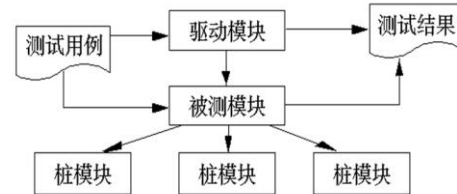
3、基本类型

1. 单元测试

(1) 编码阶段运用**白盒测试**方法，对已实现的最小单位代码进行正确性检查

(2) 包括

- ① 路径测试
- ② 接口测试
- ③ 边界条件测试
- ④ 局部数据结构测试
- ⑤ 错误处理测试



2. 集成测试

(1) 编码阶段在单元测试的基础上，运用**黑盒测试**方法检查被测单元的接口问题，并检查代码集成后各功能的完整性

(2) 在单元测试的基础上，需要将所有模块按照设计要求集成成为系统（在单元测试的同时可进行集成测试）

(3) 方法

① 一次性集成

首先对所有模块进行单元测试，然后再把所有模块集成在一起进行测试，最终得到要求的软件系统

② 增殖式集成

首先对每个模块进行模块测试，然后将这些模块逐步集成为较大的系统；在集成的过程中边连接边测试；最后逐步集成成为要求的软件系统

1) 自顶向下

将模块按系统程序结构，沿**控制层次**自顶向下进行集成

DFS 回归测试

2) 自底向上

从程序模块结构的**最底层的模块**开始集成和测试

3) 混合

3. 确认测试（有效性测试）

(1) 开发后期，针对系统级的软件验证所实现的功能和性能是否与用户的要求一致

(2) **黑盒方法**验证

4. 系统测试

(1) 在开发环境或实际运行环境中，以**系统需求分析规格说明书**作为验收标准，对软硬件系统进行的一系列集成和确认测试

5. 验收测试

(1) 在实际运行环境中，试运行一段时间后所进行的测试活动，确认系统功能和性能符合生产要求。验收通过后交付给用户使用

(2) 以用户为主

软件维护

1. 定义：在软件运行 / 维护阶段对软件产品所进行的一切改动
2. 分类
 - (1) 改正性维护
 - (2) 适应性维护
 - (3) 完善性维护
 - (4) 预防性维护
3. 影响工作量的因素
 - (1) 系统大小
 - (2) 程序设计语言
 - (3) 系统年龄
 - (4) 其他：应用的类型、数学模型、任务的难度、开关与标记、IF 嵌套深度、索引或下标数等