BUPT
TSEG

软件工程 总复习

田野： yetian@bupt.edu.cn

第一章 软件工程概述

要点:

- 软件的定义
- 软件的发展和软件危机
- 软件工程

软件的定义

- 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。
- Wirth中指出：
  - 在结构化程序设计：程序 = 算法 + 数据结构
  - 在软件工程中：软件 = 程序 + 文档。
- 另一种对软件的公认解释是：软件是包括程序、数据及其相关文档的完整集合。
- 程序和数据是构造软件的基础，文档是软件质量的保证，也是保证软件更新及生命周期长短的必需品。

软件危机

- 20世纪60年代后，随着计算机软件应用领域增多，软件规模不断扩大，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：
  - 系统存在大量错误，可用性和可靠性差；
  - 系统无法增加新功能，难于维护；
  - 系统无法按照计划时间完成；
  - ...等因素
  - 导致很多软件系统 的彻底失败。

软件危机 Software Crisis

- 所谓软件危机就是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，具体表现在：
  - 软件开发成本难以估算，无法制定合理的开发计划；
  - 用户的需求无法确切表达；
  - 软件质量存在问题；
  - 软件的可维护性差；
  - 缺乏文档资料；

Projects running over-budget
Projects running over-time
Software was very inefficient
Software was of low quality
Software often did not meet requirements
Projects were unmanageable and code difficult to maintain
Software was never delivered

软件危机的解决途径

- 产生软件危机的原因：
  - 软件系统本身的复杂性；
  - 软件开发的方法和技术不合理及不成熟；
- 软件工程方法
  - 1968年 Friedrich Ludwig（Fritz）Bauer 提出运用工程化原则和方法，[illegible]技解决软件危机，并提出“软件工程 Software Engineering”的概念。

软件工程要素

- 软件工程三要素：方法、工具和过程。
  - 方法：提供了“如何做”的技术
  - 工具：提供了自动或半自动的软件支撑环境
  - 过程：将软件工程的方法和工具综合起来以达到合理、及时地进行计算机软件开发的目的

BUPT
TSEG

第二章 软件生命周期模型

要点:

- 软件生命周期
- 传统软件生命周期模型
- 新型软件生命周期模型

软件生命周期

- 软件生命周期：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，一般包括**概念阶段、分析与设计阶段、构造阶段、移交和运行阶段**等不同时期。
- 软件生命周期的六个基本步骤
  - 制定计划　　P
  - 需求分析　　D
  - 设计　　D
  - 程序编码　　D
  - 测试　　C
  - 运行维护　　A

软件生命周期模型

- 软件过程模型：从一个特定角度提出的对软件过程的概括描述，是对软件开发实际过程的抽象，包括构成软件过程的各种活动（Activities）、软件工件（artifacts）以及参与角色（Actors/Roles）等。
- 软件生命周期模型是一个框架，描述从软件需求定义直至软件经使用后废弃为止，跨越整个生存期的软件开发、运行和维护所实施的全部过程、活动和任务，同时描述生命周期不同阶段产生的软件工件，明确活动的执行角色等。

传统模型种类

- 瀑布模型
- 演化模型
- 增量模型
- 喷泉模型
- V模型和W模型
- 螺旋模型
- 构件组装模型
- 快速应用开发模型
- 原型方法

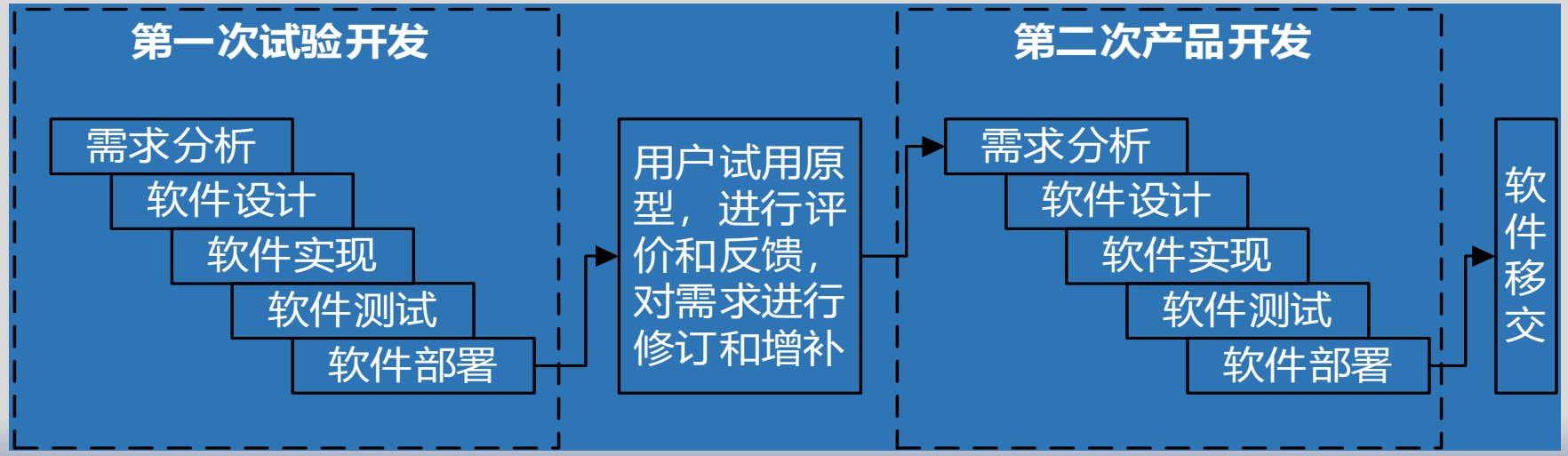
瀑布模型

- 瀑布模型中的每一个开发活动具有下列特征：
  - 本活动的工作对象来自于上一项活动的输出，这些输出一般是代表该阶段活动结束的里程碑式的文档。
  - 根据本阶段的活动规程执行相应的任务。
  - 产生本阶段活动相关产出—软件工件，作为下一活动的输入。

优点	缺点
降低了软件开发的复杂程度，提高了软件开发过程的透明性及软件开发过程的可管理性。	模型缺乏灵活性，特别是无法解决软件需求不明确或不准确的问题。
推迟了软件实现，强调在软件实现前必须进行分析和设计工作。	模型的风险控制能力较弱。
以项目的阶段评审和文档控制为手段有效地对整个开发过程进行指导，保证了阶段之间的正确衔接，能够及时发现并纠正开发过程中存在的缺陷，从而能够使产品达到预期的质量要求。	瀑布模型中的软件活动是文档驱动的，当阶段之间规定过多的文档时，会极大地增加系统的工作量；而且当管理人员以文档的完成情况来评估项目完成进度时，往往会产生错误的结论。

演化模型

- 使用瀑布模型人们认识到，由于需求很难调研充分，所以很难一次性开发成功。
- 演化模型提倡两次开发：
  - 第一次是试验开发，得到试验性的原型产品，其目标只是在于探索可行性，弄清软件需求；
  - 第二次在此基础上获得较为满意的软件产品。



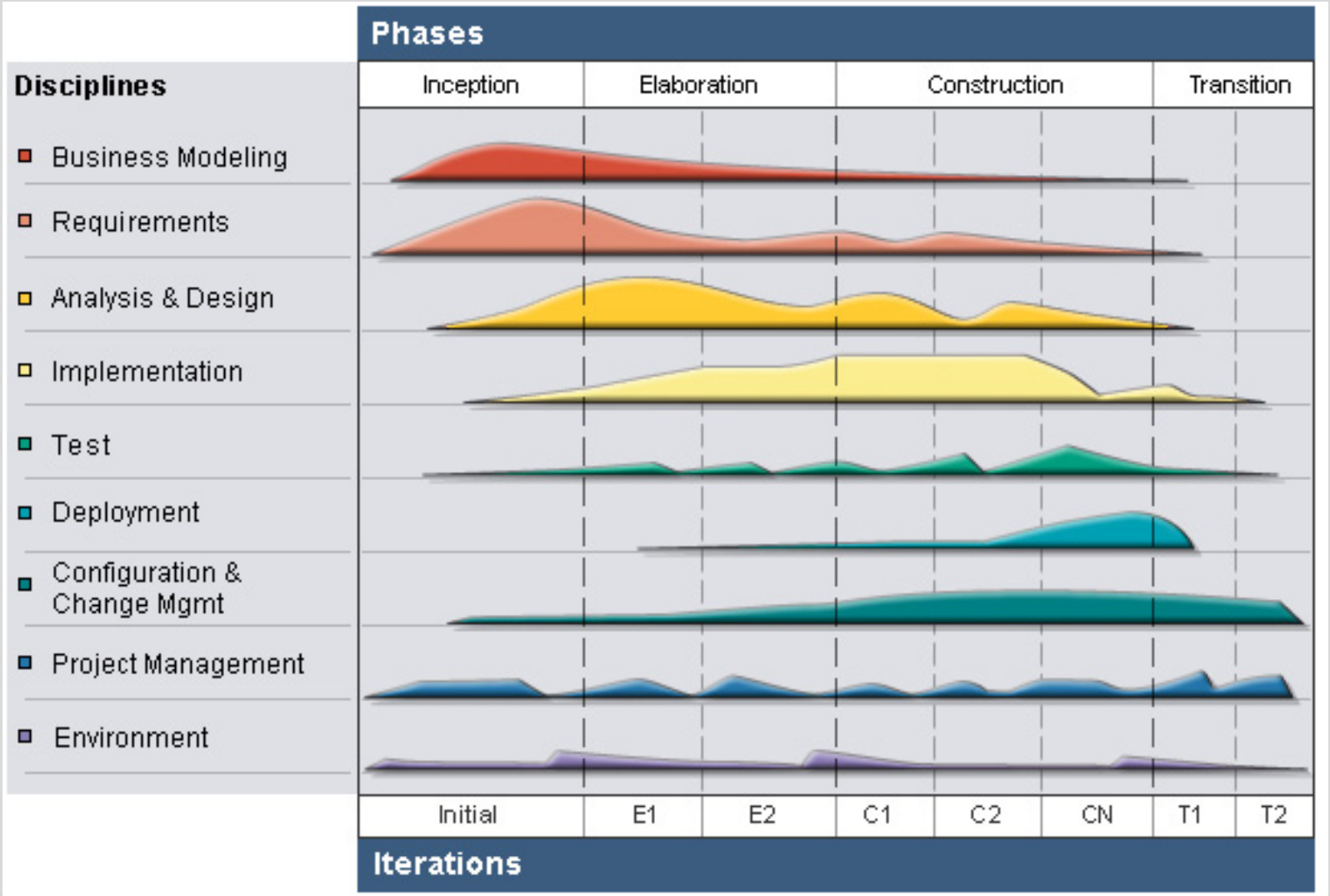
演化模型

- 演化模型的特点：
  - 优点：明确用户需求、提高系统质量、降低开发风险；
  - 缺点：
    - 难于管理、结构较差、技术不成熟；
    - 可能会抛弃瀑布模型的文档控制优点；
    - 可能会导致最后的软件系统的系统结构较差；
- 演化模型适用范围：
  - 需求不清楚；
  - 小型或中小型系统；
  - 开发周期短

新型软件生命周期模型

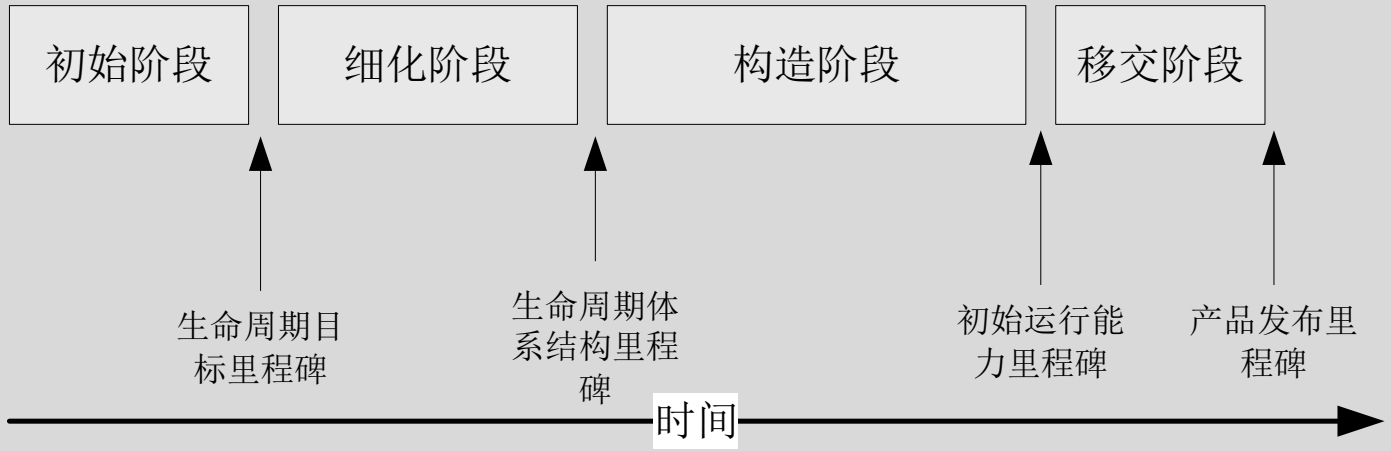
- RUP
  - RUP（Rational Unified Process）是由Rational公司（现被IBM公司收购）开发的一种软件工程过程框架，是一个基于面向对象的程序开发方法论 。
  - RUP既是一种软件生命周期模型，又是一种支持面向对象软件开发的工具，它将软件开发过程要素和软件工件要素整合在统一的框架中 。
- 敏捷及极限编程
  - 敏捷建模（Agile Modeling，AM）是由Scott W. Ambler从许多的软件开发过程实践中归纳总结出来的一些敏捷建模价值观、原则和实践等组成的，它是快速软件开发的一种思想代表，具体的应用有极限编程、SCRUM、水晶、净室开发等。
  - 2001年敏捷联盟成立，其主要特点就是具有快速及灵活的响应变更的能力。

RUP的基本结构



RUP的四个主要阶段

- RUP中的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（Inception）、细化阶段（Elaboration）、构造阶段（Construction）和交付阶段（Transition）。
- 每个阶段结束于一个主要的里程碑(Major Milestones)，并在阶段结尾执行一次评估以确定这个阶段的目标是否已经满足。如果评估结果令人满意的话，可以允许项目进入下一个阶段。



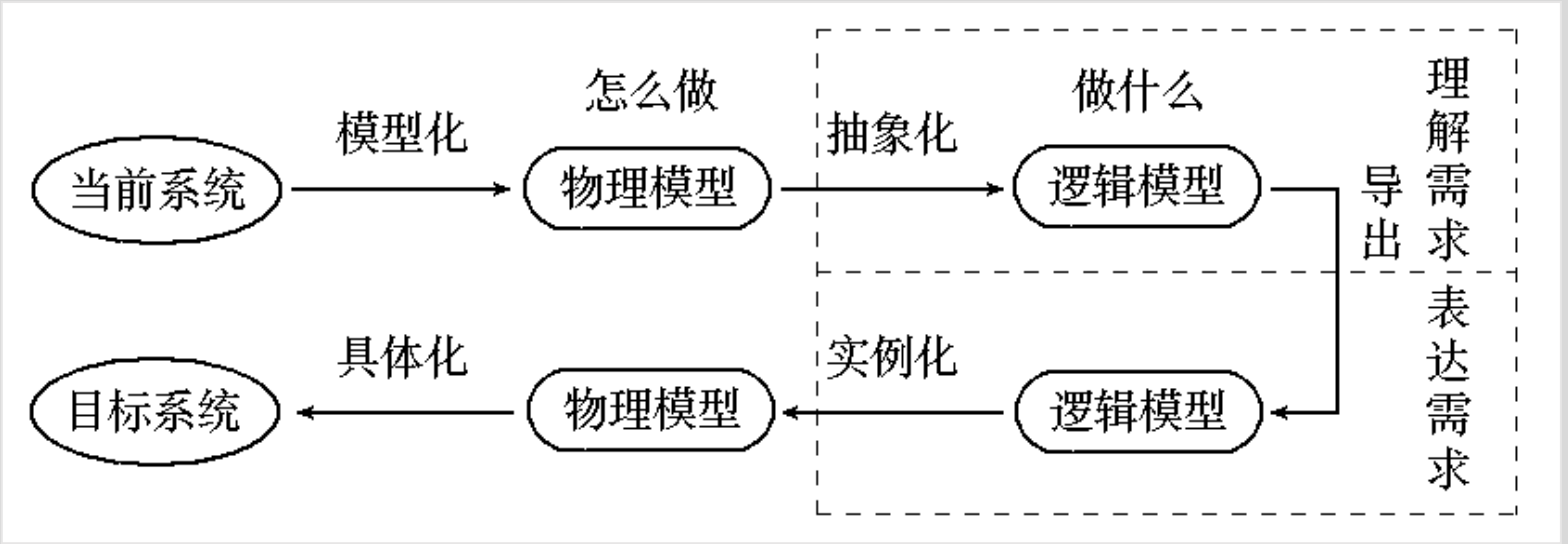
第三章 软件需求分析

要点：

- 需求分析的对象、任务和目标
- 数据、功能、行为建模
- 需求类别

需求分析的对象、任务和目标

- 软件需求分析的对象：用户要求。
- 软件需求分析的任务是：准确地定义新系统的目标，回答系统必须“做什么”的问题并编制需求规格说明书。
- 需求分析的目标：借助于当前（业务）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。



数据、功能及行为建模

- **数据模型：**
  - 信息内容和关系；信息流；信息结构。
- **功能模型：**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。
- **行为模型：**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。

需求类别

- **功能需求**：列举出所开发软件在功能上应做什么，这是最主要的需求。
- **性能需求**：给出所开发软件的技术性能指标，尤其是系统的实时性和其他时间要求，如响应时间、处理时间、消息传送时间等；资源配置要求，精确度，数据处理量等要求。
- **环境需求**：是对软件系统运行时所处环境的要求。
  - 在硬件方面，采用什么机型、有什么外部设备、数据通信接口等等。
  - 在软件方面，采用什么支持系统运行的系统软件（指操作系统、数据库管理系统等）。
  - 在使用方面，需要使用部门在制度上、操作人员的技术水平上应具备什么样的条件等等。

其他需求类别

- 可靠性需求：指软件的有效性和数据完整性。各种软件在运行时失效的影响各不相同。在需求分桥时，应对所开发软件在投入运行后不发生故障的概率，按实际的运行环境提出要求。
- 安全保密要求：工作在不同环境的软件对其安全、保密的要求显然是不同的，应当把这方面的需求恰当地做出规定。
- 用户界面需求：软件与用户界面的友好性是用户能够方便有效愉快地使用该软件的关键之一。
- 资源使用需求：指所开发软件运行时所需的数据、软件、内存空间等各项资源，以及软件开发时所需的人力、支撑软件、开发设备等。
- 软件成本消耗与开发进度需求：在软件项目立项后，要根据合同规定，对软件开发的进度和各步骤的费用提出要求，作为开发管理的依据。
- 预先估计以后系统可能达到的目标：在开发过程中，可对系统将来可能的扩充与修改做准备。一旦需要时，就比较容易进行补充和修改。

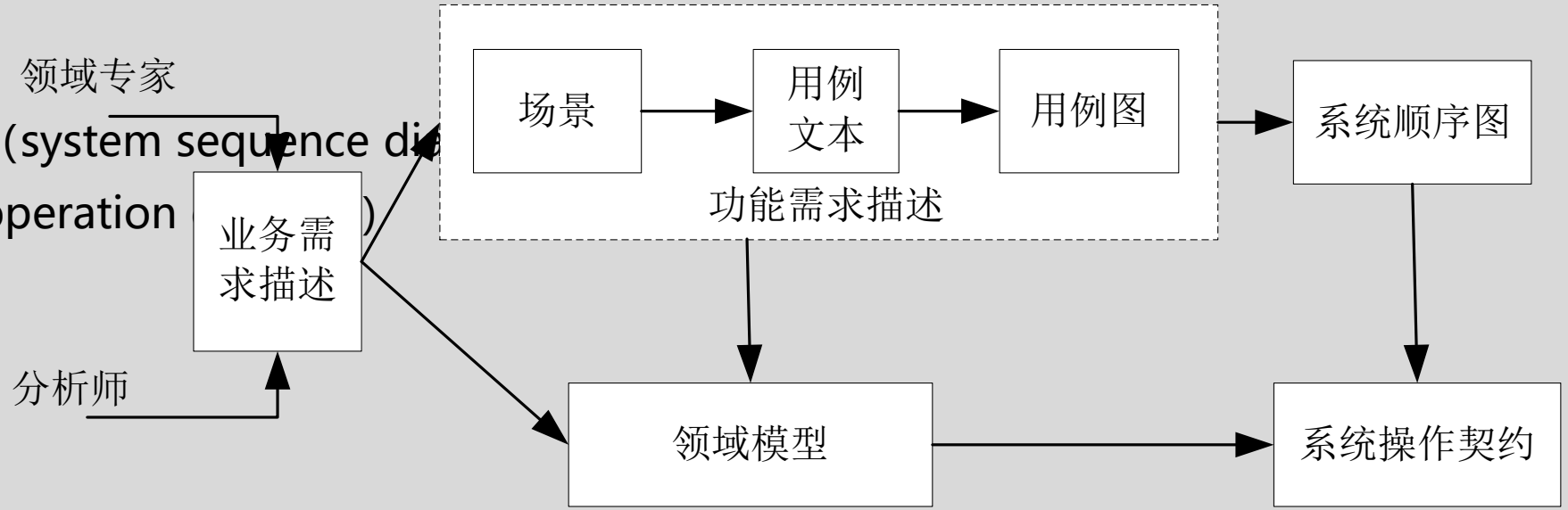
第四章 面向对象需求分析方法

要点:

- 领域建模
- 用例建模

面向对象的需求分析建模

- 面向对象分析方法中的需求分析包含两个模型：领域模型和用例模型。
  - 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；
  - 用例模型是“目标系统”的逻辑模型，定义了“目标系统”做什么的需求。由以下四个部分组成：
    - 用例图
    - 用例说明
    - 系统顺序图（system sequence di
    - 操作契约（operation



领域模型

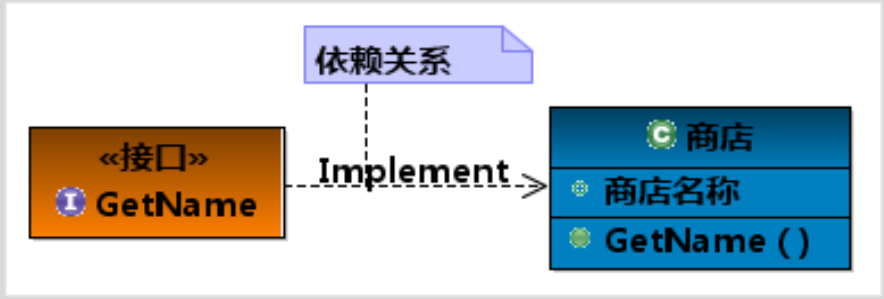
- 领域模型：针对某一特定领域内概念类或者对象的抽象可视化表示。
- 主要用于概括地描述业务背景及重要的业务流程，并通过UML的类图和活动图进行展示，帮助软件开发人员在短时间内了解业务。
  - 业务背景：可由需求定义或者用例说明中具有代表业务概念或者业务对象的词汇获得，这些词汇可统称为“概念类”；并通过能够代表关系的词汇建立概念类之间的关系，表示成能够代表业务知识结构的类图；
  - 业务流程：一般由角色及其执行的活动（活动及任务节点）构成，活动的输出一般有数据对象和传给另一个活动的消息组成，建议使用UML的活动图进行描述。

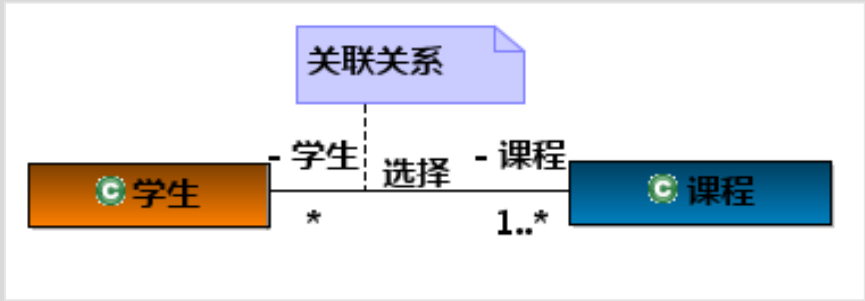
创建领域模型的步骤

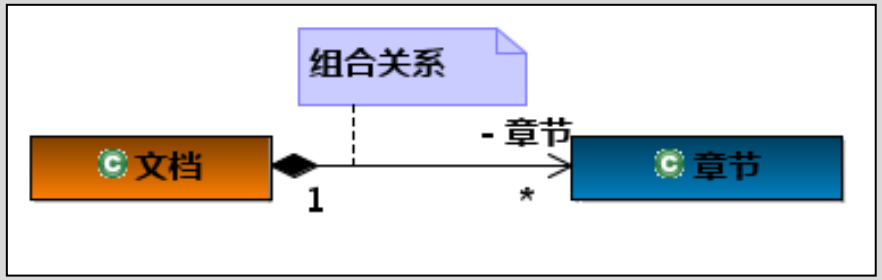
- 理解领域模型对理解系统需求至关重要，领域模型的创建步骤如下：
  - 第1步，找出当前需求中的候选概念类；
  - 第2步，在领域模型中描述这些概念类。用问题域中的词汇对概念类进行命名，将与当前需求无关的概念类排除在外。
  - 第3步，在概念类之间添加必要的关联来记录那些需要保存记忆的关系，概念之间的关系用关联、继承、组合/聚合来表示。
  - 第4步，在概念类中添加用来实现需求的必要属性。

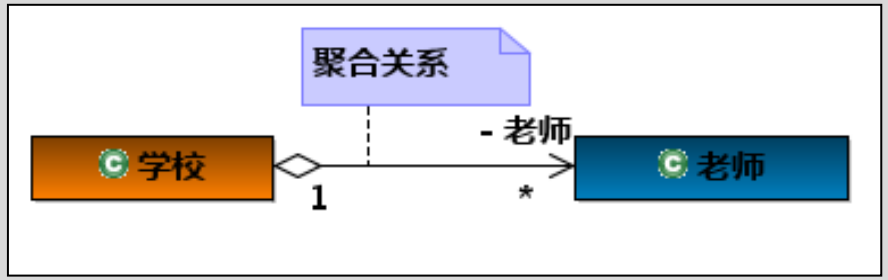
类的关系

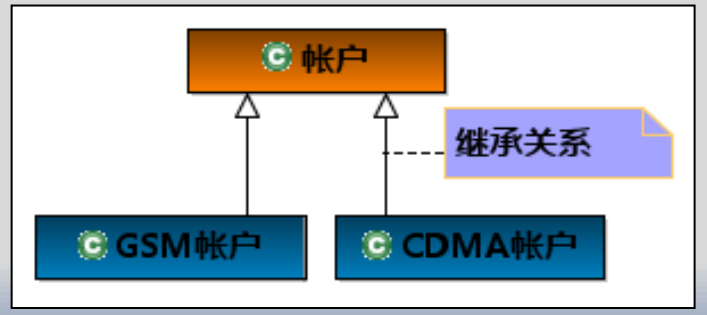
- 以下按照由松散到紧密地的关系进行说明









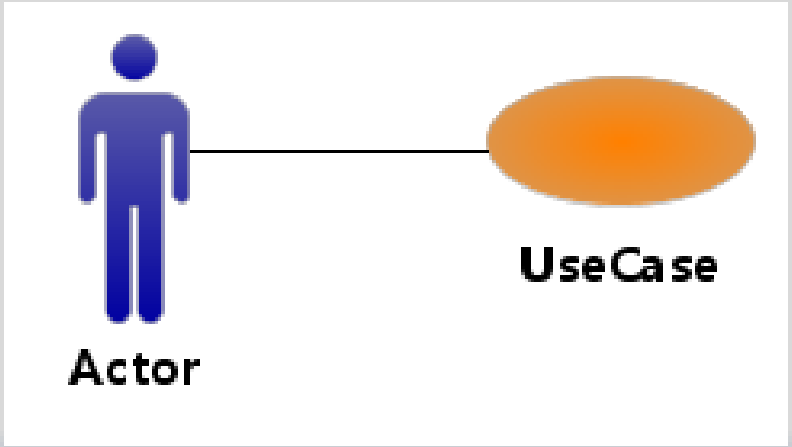


用例模型

- 用例模型由以下四个部分组成：
  - 用例图；
  - 用例说明；
  - 系统顺序图（system sequence diagram，option）；
  - 操作契约（operation contract，option）；
- 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

用例图

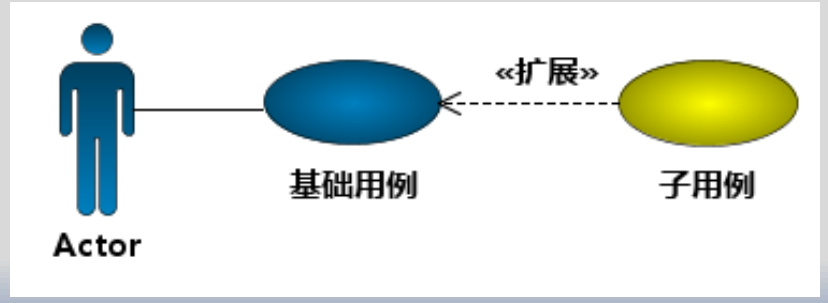
- 用例图由三个基本元素组成
  - Actor：称为角色或者参与者，表示使用系统的对象，代表角色的不一定是人，也可以是组织、系统或设备；
  - Use_case：称为用例，描述角色如何使用系统功能实现需求目标的一组成功场景和一系列失败场景的集合；
  - Association：表示角色与用例之间的关系，以及用例和子用例之间的关系；



基本用例与子用例

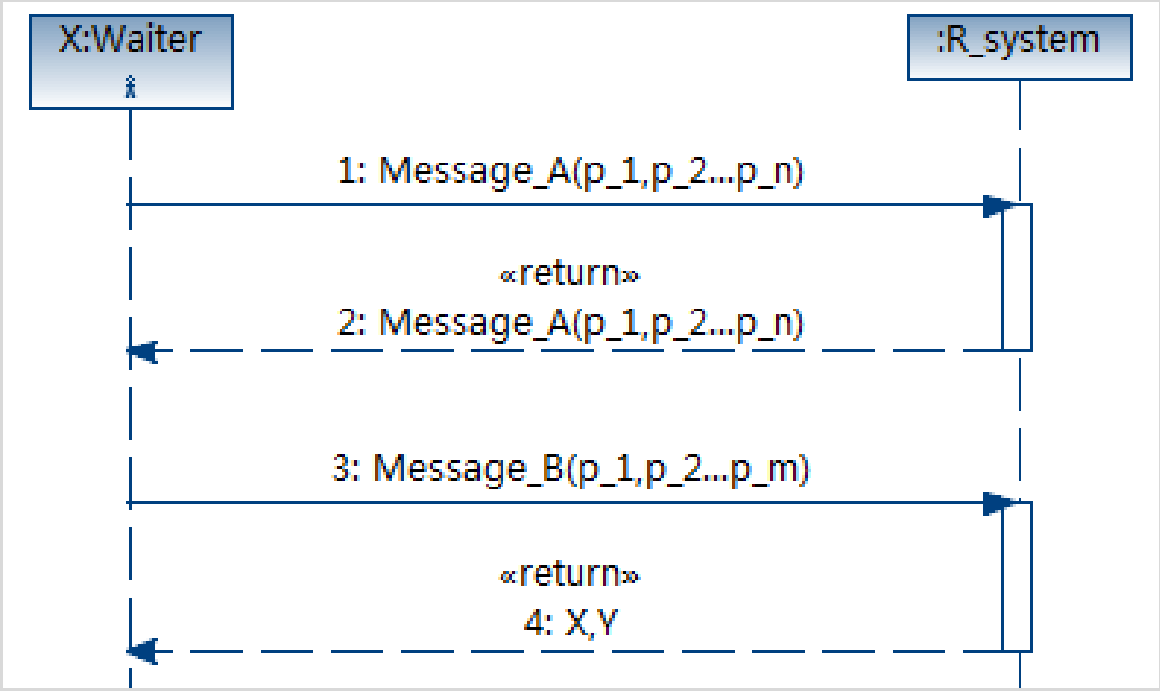
- 基本用例：与角色直接相关的用例，表示系统的功能需求；
- 子用例：通过场景描述分析归纳出的用例，也表示了系统的功能，但这些用例与角色无直接关系，而与基本用例存在关联关系；
  - 包含子用例：多个基本用例中的某个与角色交互的场景具有相同的操作，且这些场景都是基本用例中必须执行的步骤，可以将其抽取出来作为基本用例的子用例；

  - 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；



系统顺序图

- 在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；
- 系统顺序图中“一般”只需要三个UML的符号元素
  - 角色；
  - 代表软件系统的对象，一般使用system或者系统命名；
  - 角色与system之间的交互信息，简称消息或操作；



操作契约

- 系统操作：处理系统事件的操作，也称为系统事件；
- 操作契约是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，以便软件设计时进行参考。模板如下表所示：

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；

创建操作契约

- 创建操作契约的指导原则如下：
  - 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以下内容描述并确定对象的状态变化，即后置条件；
    - 对象实例创建和删除。
    - 对象属性修改。
    - 对象关联形成和断开。
- 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。

第五章 结构化需求分析方法

要点:

- 数据建模
- 功能建模
- 行为建模

数据建模

- 概念性数据模型是一种面向问题的数据模型，是按照用户的观点来对数据和信息建模。其表示方法称为实体-关系（Entity-Relation）法，也称为实体关系模型。
- 它描述了从用户角度看到的数据，反映了用户的现实环境，但与在软件系统中的实现方法无关。
- 软件系统本质上是信息处理系统，即对数据进行处理的系统，因此在开发过程中必须考虑以下两方面的问题：
  - “数据”
    - 需要有哪些数据?
    - 数据之间有什么联系?
    - 数据本身有什么性质?
    - 数据结构等
  - 对数据的“处理”
    - 对数据进行哪些处理?
    - 每个处理的逻辑功能是什么?

数据对象

- 数据对象描述包括了数据对象的名称及其所有属性。通常将数据对象简称为“实体”，其具体表现可以是：
  - 外部实体：产生或使用消息的任何事物；
  - 事物：例如建筑物、汽车等物体；
  - 事件：例如警报；
  - 角色：例如老师、学生、销售等；
  - 组织单位：例如学校教务处、财务处等；
  - 地点：例如仓库、停车场等；
  - 结构：例如文件、档案等。

数据对象的属性和关系

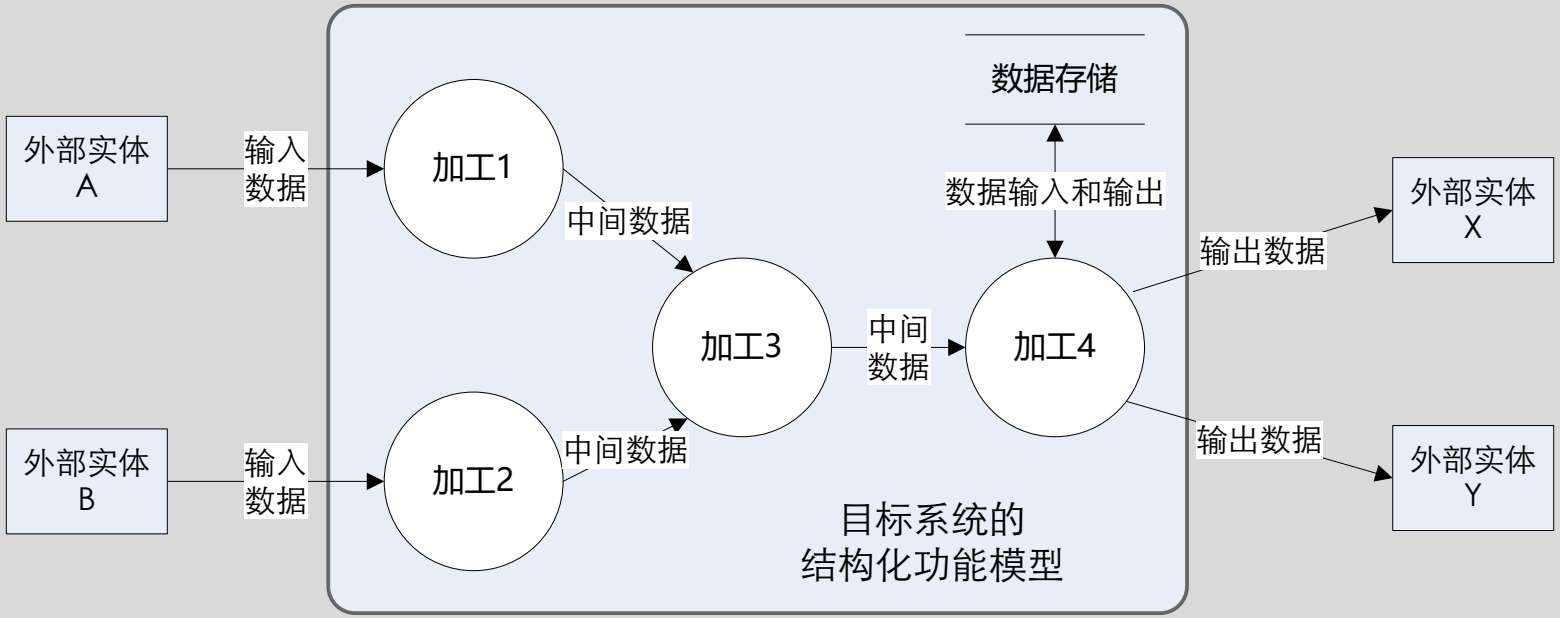
- 每个数据对象都具有一些区别于其他数据对象的特征和性质，这些特征称为数据对象的属性。它可用于：
  - 命名数据对象；
  - 描述数据对象实例；
  - 建立与其它数据对象的联系；
- 数据对象之间可以存在某种特定的连接，称之为数据对象的关系。
- 关系是由被分析问题的语境定义的。

数据对象的基数

- 数据建模的基本元素：数据对象、属性和关系提供了理解问题信息域的基础，但还必须了解数据对象之间出现的次数有无必然的联系，即实体-关系对的基数。
- 基数通常简单地表达为“一”或“多”。考虑到“一”和“多”的所有组合，两个实体可能的关联如下：
  - 一对一：例如人和身份证件的关系；
  - 一对多：例如父母与孩子的关系；
  - 多对多：例如学生和老师的关系；

功能建模

- 当数据或信息“流”过计算机系统时将会被系统的功能所处理、加工或变换后再将处理或变换后的数据从系统输出。
- 基于计算机的系统可被表示为数据流图的基本结构：



数据流图的结构

- 数据流图可以被用来抽象地表示系统或软件，既能提供功能建模的机制。
- 也可提供数据流建模的机制，并可以自顶向下的机制表示层级的功能细节和数据变换细节。
- 从数据流图中可知，数据流图有四种基本元素：

分层的数据流图

- 为表达复杂的实际问题（图形符号过多），需要按照问题的层次结构进行逐步分解，并以分层的数据流图反映这种结构关系。
  - 顶层数据流图：顶层流图仅包含一个加工，它代表被开发系统，其作用在于表明被开发系统的范围，以及它和周围环境的数据交换关系。
  - 中间层数据流图：表示对其上层父图的细化。它的每一加工可以继续细化，形成子图。中间层次的多少视系统的复杂程度而定。
  - 底层数据流图：是指加工不须再做分解的数据流图，称为“原子加工”。

行为建模

- 为了直观地分析系统的动作，从特定的视角出发描述系统的行为，需要采用动态分析的方法。
- 其中最为常用的结构化动态分析方法有：
  - 状态迁移图
  - 时序图
  - Petri网等。

第六章 软件设计的概念及原则

要点：

- 软件设计过程
  - 概要设计
  - 详细设计
- 软件设计模型
- **模块、模块的独立性**
- **面向对象的设计原则**
- 软件体系结构风格

软件概要设计

- 制定设计规范
- 软件系统结构的总体设计
- 处理方式设计（性能设计）
- 数据结构设计
- 可靠性设计（质量设计）
- 界面设计（需求的直接表达方式）
- 编写软件概要设计说明书
- 概要设计评审

软件详细设计

- 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。
- 选定某种表达形式来描述各种算法。
- 编写软件详细设计说明书
- 进行详细设计的评审。

软件设计模型

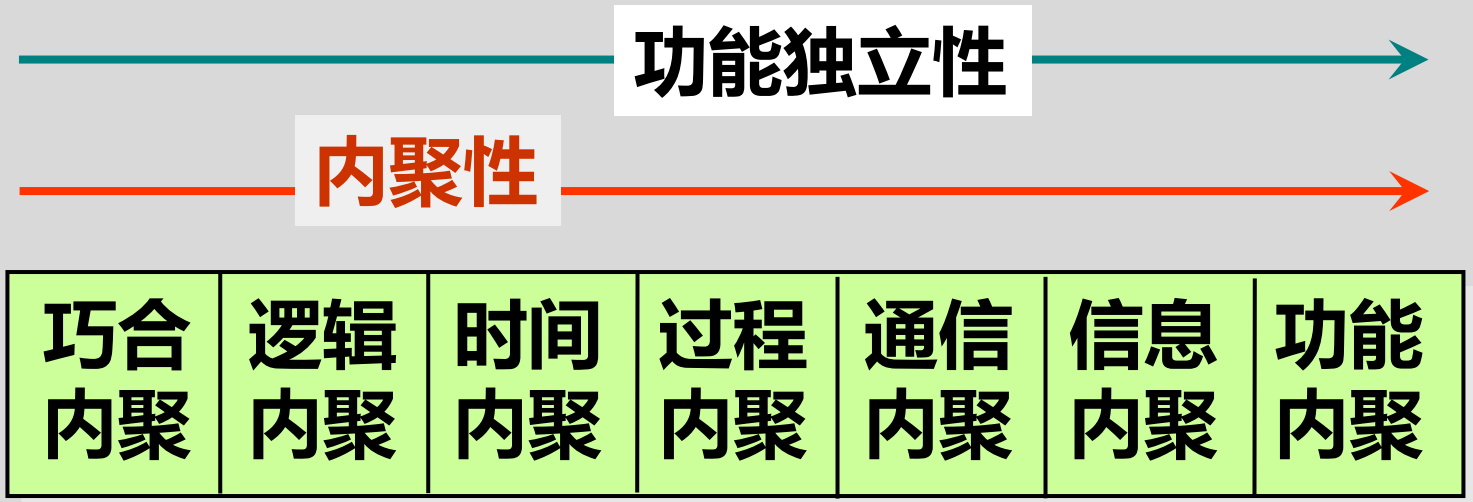
- 软件设计既是过程又是模型。
- 软件设计模型由两个部分构成：
  - 动态结构设计：以某种方式表示功能响应客户请求时处理数据的过程或条件，用于进一步解释软件结构中各功能之间是如何协调工作的机制。
  - 静态结构设计：由软件的功能结构和数据结构组成，展示软件系统能够满足所有需求的框架结构；
- 软件的设计活动
  - 系统结构设计及数据结构设计；
  - 接口设计和过程设计；
  - 界面设计、组件设计及优化设计等；

软件的模块化

- 模块(module)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。
- 模块具有如下三个基本属性：
  - 功能：实现什么功能，做什么事情。
  - 逻辑：描述模块内部怎么做。
  - 状态：该模块使用时的环境和条件。
- 模块的表示
  - 模块的外部特性：是指模块的模块名、参数表、以及给程序以至整个系统造成的影响。
  - 模块的内部特性：是指完成其功能的程序代码和仅供该模块内部使用的数据。

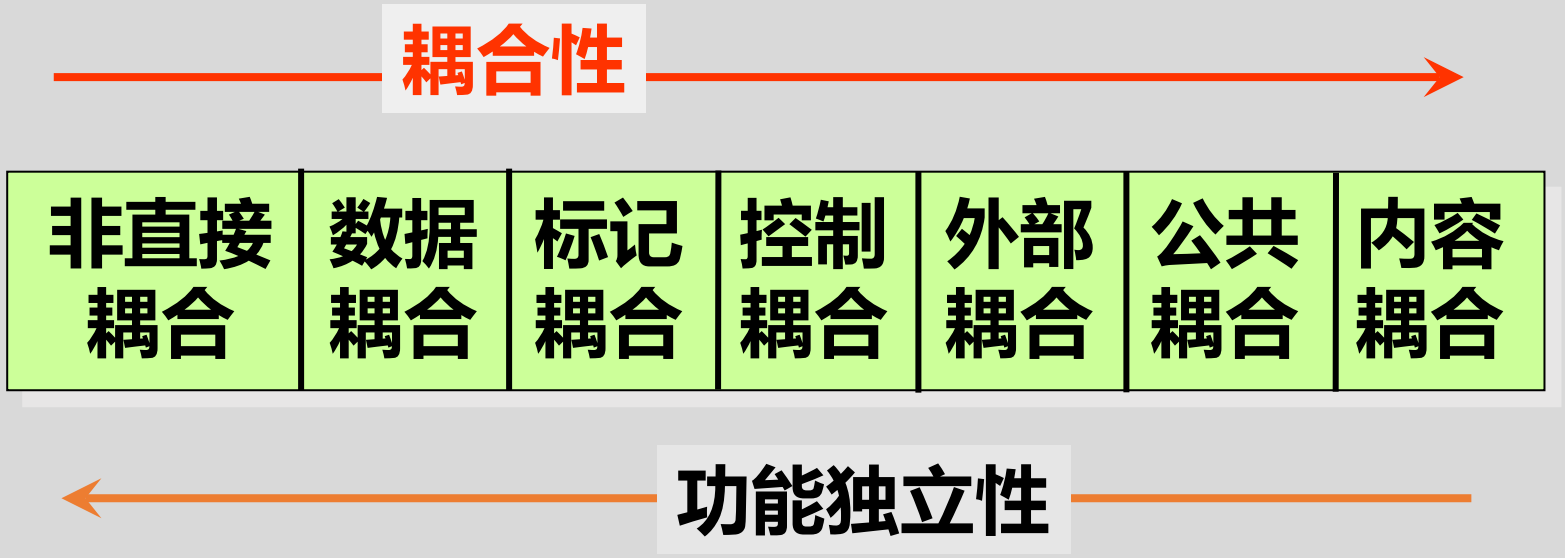
模块的内聚性

- 内聚是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，相对地，它与其他模块之间的耦合性就会减低，而模块独立性就越强。



模块的耦合性

- 耦合是模块之间互相连接的紧密程度的度量。模块之间的连接越紧密，联系越多，耦合性就越高，而其模块独立性就越弱。



面向对象的设计原则

- **单一职责**（Single Responsibility）
- **开闭原则**（Open Closed）
- **里氏替换原则**（Liskov Substitution）
- **依赖倒置原则**（Dependency Inversion）
- 接口隔离原则（Interface Segregation）
- 组合/聚合复用（Composite/Aggregation Reuse）
- 迪米特法则（Law of Demeter）

软件体系结构风格

- 软件体系结构设计的一个核心问题是能否使用重复的体系结构模式。
- 基于这个目的，学者们开始研究和实践软件体系结构的风格和类型问题。
- 软件体系结构风格是描述某一特定应用领域中系统组织方式的惯用模式。
  - 管道和过滤器风格
  - 调用和返回风格
  - 主程序/子程序风格
  - **对象风格**
  - **分层风格**
  - 基于事件的风格
  - 客户端/服务器风格
  - **MVC风格**
  - 黑板风格

BUPT
TSEG

第七章 面向对象设计方法

要点:

- 面向对象设计步骤
- 模型层次化
- 面向对象的设计模式

面向对象的设计与UML

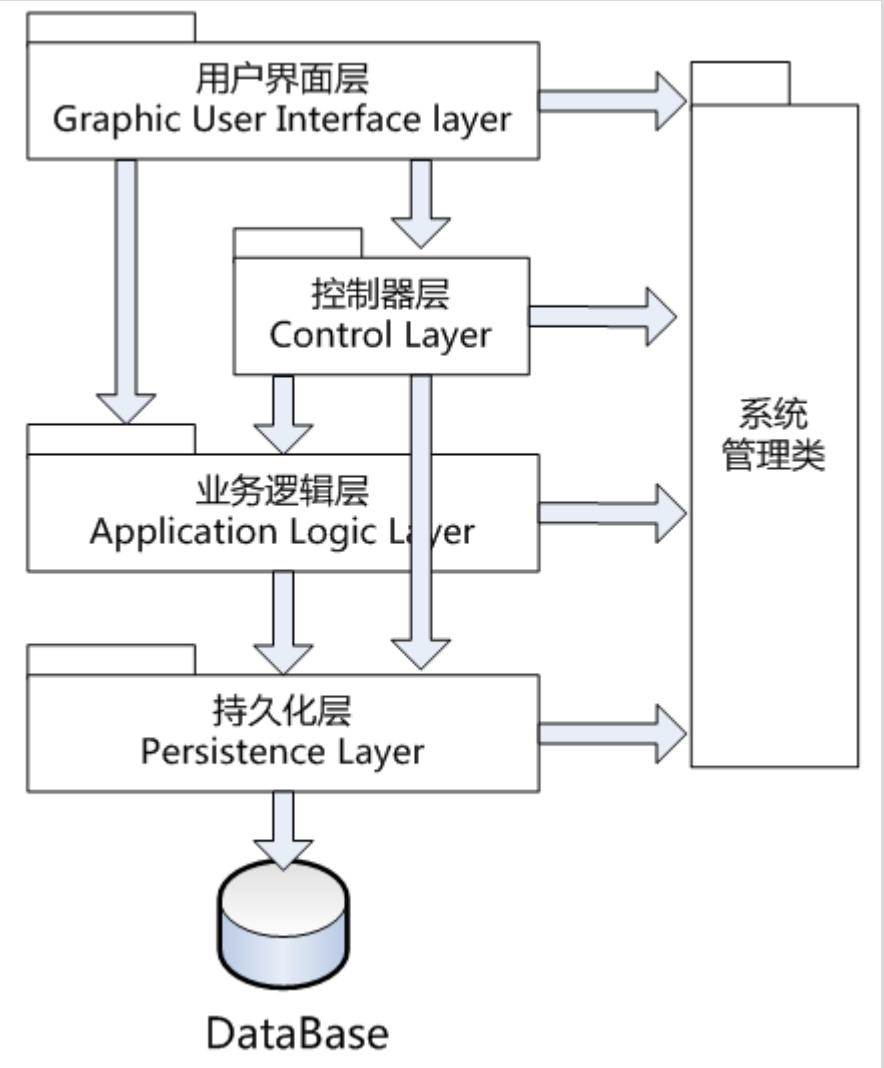
- 软件概要设计步骤
  - 选择合适的软件架构；
  - 系统的动态结构设计：
    - 用例实现过程设计，针对用例对应的SSD中的每个系统事件，运用UML的 sequence diagram / collaboration diagram 给出符合该系统事件定义的操作契约的内容；
    - 如果软件对象具有多种不同的职责（主要考虑对应于不同的用例）的情况下，需要运用 state machines diagram 对该软件对象进行状态迁移的设计；
  - 系统的静态结构设计
    - 对所有用例或者子系统级别的用例的交互图进行归纳，运用UML的 Class diagram 给出系统的静态结构；
- 软件详细设计
  - 针对系统静态结构中每个对象的方法，运用UML activity diagram 对其进行逻辑结构的设计

面向对象设计的关键步骤

- 在确定软件框架结构的基础上，进行以下内容的设计
  - 发现对象（发现软件类）：根据需求和选择的架构和模式确定系统由哪些对象构成；
  - 确定对象属性：明确该对象应该具有的特征属性；
  - 确定对象行为：明确对象应具有的功能和职责；
  - 确定对象之间的关系：根据系统顺序图及操作契约以及选择的架构和模式明确系统是如何相互协作完成功能需求的交互过程；

（基于BS结构）模型的层次化

- 层次化的设计模型是面向对象方法基于软件体系结构风格的一种方案选择。层次化的设计模型符合面向对象的设计原则，并使系统易于扩展和维护。
  - 用户界面层：（用例）系统功能的各种界面表现形式。
  - 控制器层：用于协调、控制其他类共同完成用例规定的功能或行为。
  - 业务/应用层：实现用例要求的各种系统级功能；
  - 持久化层：用于保存需要持久化存储的数据对象；
  - 系统层：为应用提供操作系统相关的功能，通过把特定于操作系统的特性包装起来，使软件与操作系统分离，增加应用的可移植性。



面向对象的设计模式

- 对象的职责通过调用对象的方法来实现。将职责分配给一个对象还是多个对象，是分配给一个方法还是多个方法要受到职责粒度的影响。
- 面向对象设计最关键的活动是正确地给对象分配职责。
- 模式是面向对象软件的设计经验，是可重用的设计思想，它描述了在特定环境中反复出现的一类设计问题，并提供经过实践检验的解决这类问题的通用模式。
- 模式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。

类职责分配(Grasp)模式

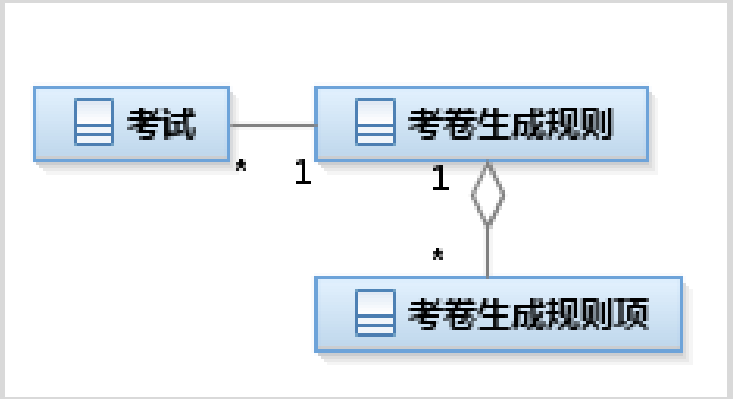
- 设计类的来源有两部分。
  - 核心逻辑由领域模型中的概念类转换而来
  - 另一部分则是为实现而新增的一些类，如负责对象持久化的类、负责通信的类。
- 每一个设计类都有明确的职责，分为两种类型：
  - 了解型（knowing）职责（自己干自己的事）细分为三类：对象要了解自己私有的封装数据；了解相关联的对象；了解能够派生或者计算的事物。
  - 行为型（doing）职责（自己干自己能干的事）。细分为三类：对象自身要能执行一些行为，如创建一个对象或者进行计算；对象要能启动其他对象中的动作；对象要能控制或协调其他对象中的活动。
  - 职责的内聚（自己只干自己的事）：目的是提高内聚降低耦合，减少不必要的关联关系

控制器(Controller)模式

- 问题来源：第一个接收系统事件的软件对象是什么？哪个软件对象负责接收和处理一个系统输入事件?
- 解决方案：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
  - 它代表整个系统（系统简单且不复杂），称为外观（facade）控制器;
  - 它代表一个发生系统事件的用例场景，这个类通常命名为“<用例名>控制器”，称为用例控制器或者会话控制器。
  - 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;
  - 一次会话是与一个参与者进行交谈的一个实例。

创建者(Creator)模式

- 问题来源：哪个对象应该负责产生类的实例？（操作契约中对象实例的创建）
- 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)。
  - B聚合（aggregate）或包含（contain）对象A；
  - B记录（record）对象A；
  - B密切使用对象A；
  - 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）。
- 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。



信息专家(Information Expert)模式

- 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。
- 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，选取类的方法：
  - 如果在设计模型中存在相关的类，先到设计模型中查看；
  - 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。
- 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。

创建设计类图

- 通过类职责分配，找出了实现用例的类，以及类的职责。结合分析阶段的领域模型，可以得到设计阶段的类图，简称设计类图。
- 设计类图中主要定义类、类的属性和操作，但是不定义实现操作的算法。

1、通过扫描所有的交互图以及领域模型中涉及的类，识别软件类。

2、将领域模型中已经识别出来的部分属性添加到类中。

3、根据交互图为软件类添加方法。忽略软件类的构造函数和get/set方法；

4、添加更多的类型信息。包括属性类型、方法参数类型以及返回类型。

5、添加关联和导航。定义A到B带导航修饰关联的常见情况有以下几种：

- A发送一个消息到B；
- A创建一个B的实例；
- A需要维护到B的一个连接。

6、类成员的细节表示（可选）。如成员的属性可见性，方法体的描述等。

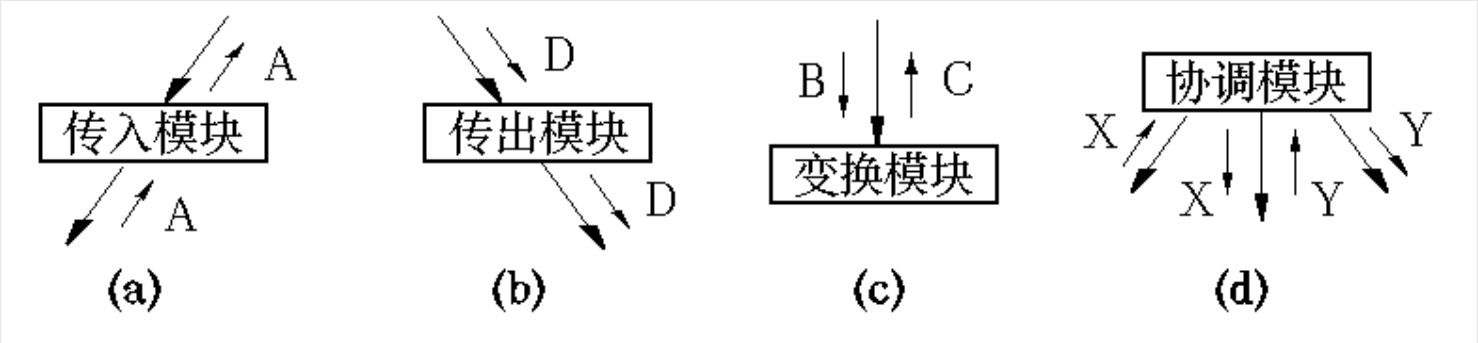
第八章 结构化设计方法

要点：

- 功能结构图基本结构
- **变换型映射**
- **事务型映射**

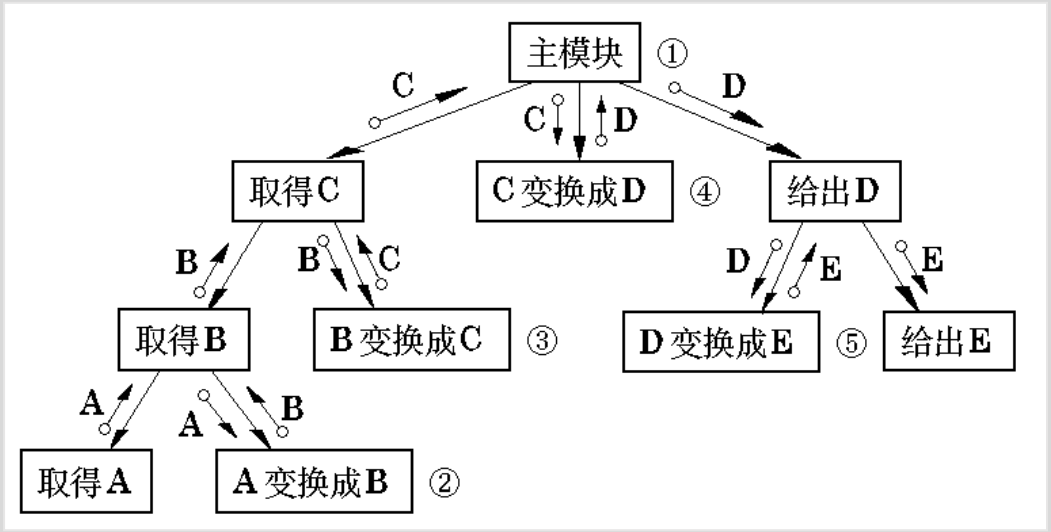
功能结构图的基本结构

- 四种基本类型的模块
  - 传入模块： 从下属模块取得数据，经过某些处理，再将其传送给上级模块。
  - 传出模块： 从上级模块获得数据，进行某些处理，再将其传送给下属模块。
  - 变换模块： 即加工模块。它从上级模块取得数据，进行处理，转换成其它形式，再传送回上级模块。
  - 协调模块： 对所有下属模块进行协调和管理的模块。



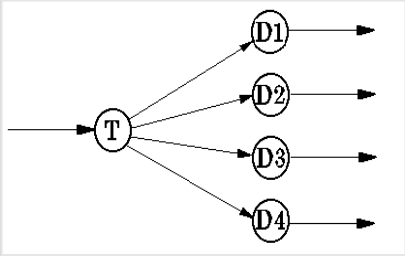
变换型结构

- 变换型数据处理问题的工作过程大致分为三步：
  - 取得数据
  - 变换数据
  - 给出数据

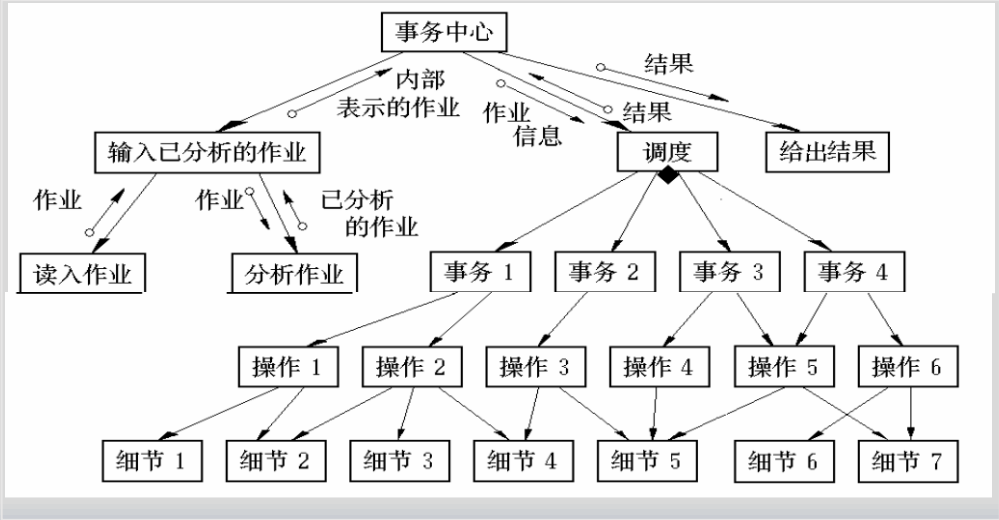


事务型结构

- 存在某一个数据流处理节点，引发一个或多个相同的处理，并将处理结果返回给该节点，则该数据流就叫做事务，该节点称为事务处理中心。
  - 事务处理中心
  - 事务处理加工



- 事务是最小的工作单元，不论成功与否都作为一个整体进行工作。
  - 原子性
  - 一致性
  - 隔离性
  - 持久性



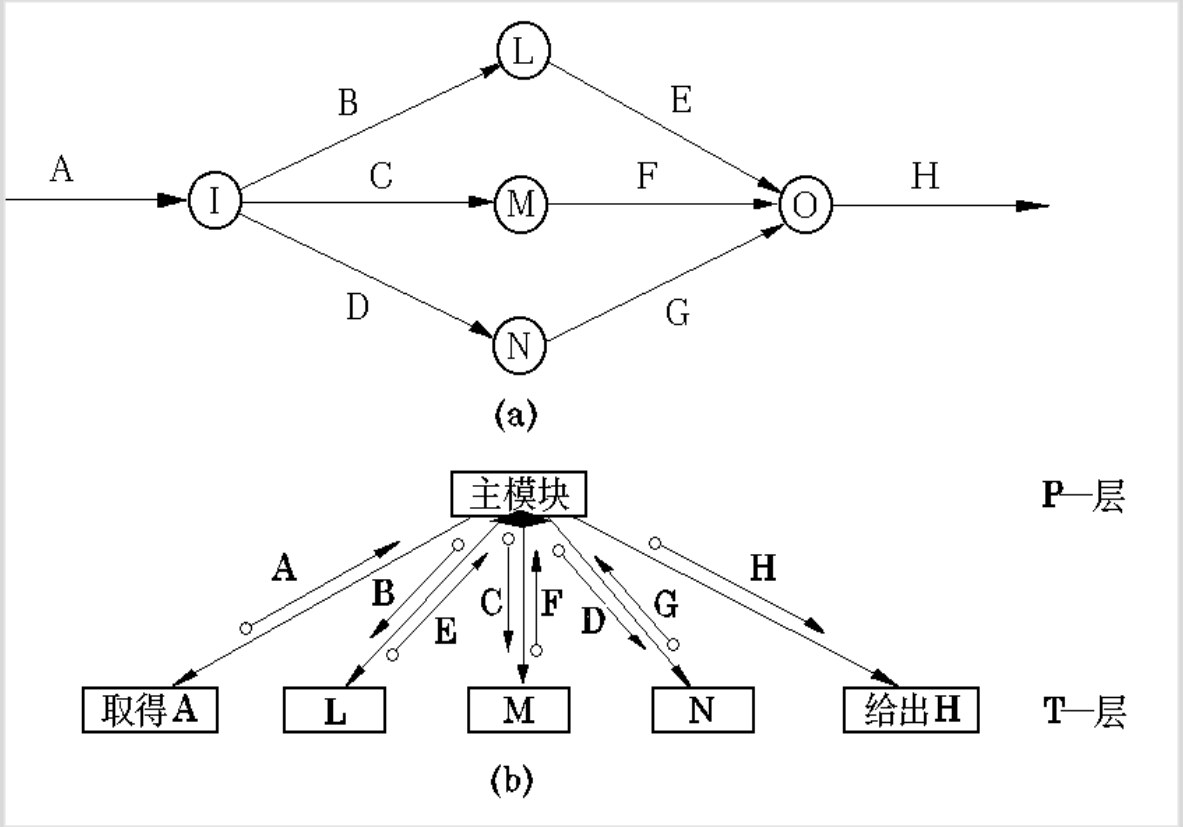
变换映射

- 变换映射是一组设计步骤，将具有变换流特征的数据流图映射为一个预定义的程序结构模版。
- 运用变换映射方法建立初始的系统功能结构图，然后进行多次改进或优化，得到系统的最终结构图。
  - 复审并评估分析模型；
  - 复审并重画数据流图；
  - 确定数据流图中的变换和事务特征；
  - 区分输入流、输出流和中心变换部分，即标明数据流的边界；
  - 进行一级“因子化”分解，设计顶层和第一层模块；
  - 进行二级“因子化”分解，设计中、下层模块；
  - 利用一些启发式原则来改进系统的初始结构图，直到得到符合要求的结构图为止。

事务映射

- 事务映射也从分析数据流图开始，自顶向下，逐步分解，建立事务型系统结构图。

1. 复审系统分析模型
2. 重画数据流图
3. 确定是否具有事务流特征
4. 确定事务中心及流特征
5. 进行事务映射
6. 因子化分解和细化
7. 优化系统结构

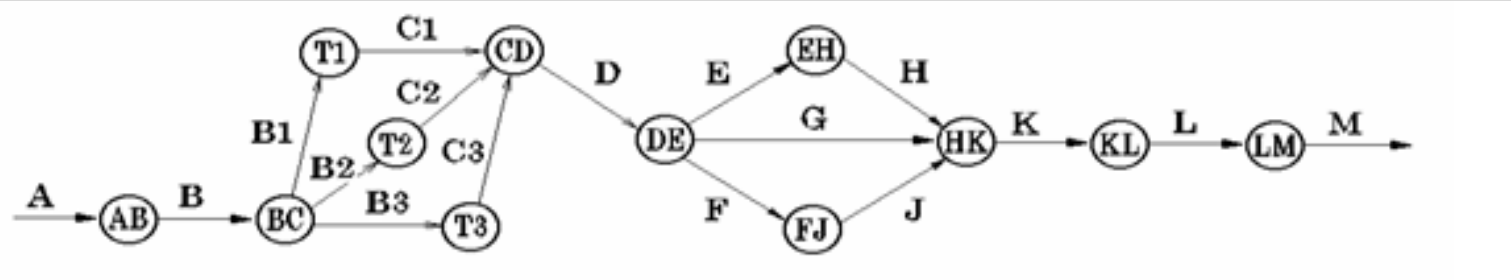


事务映射

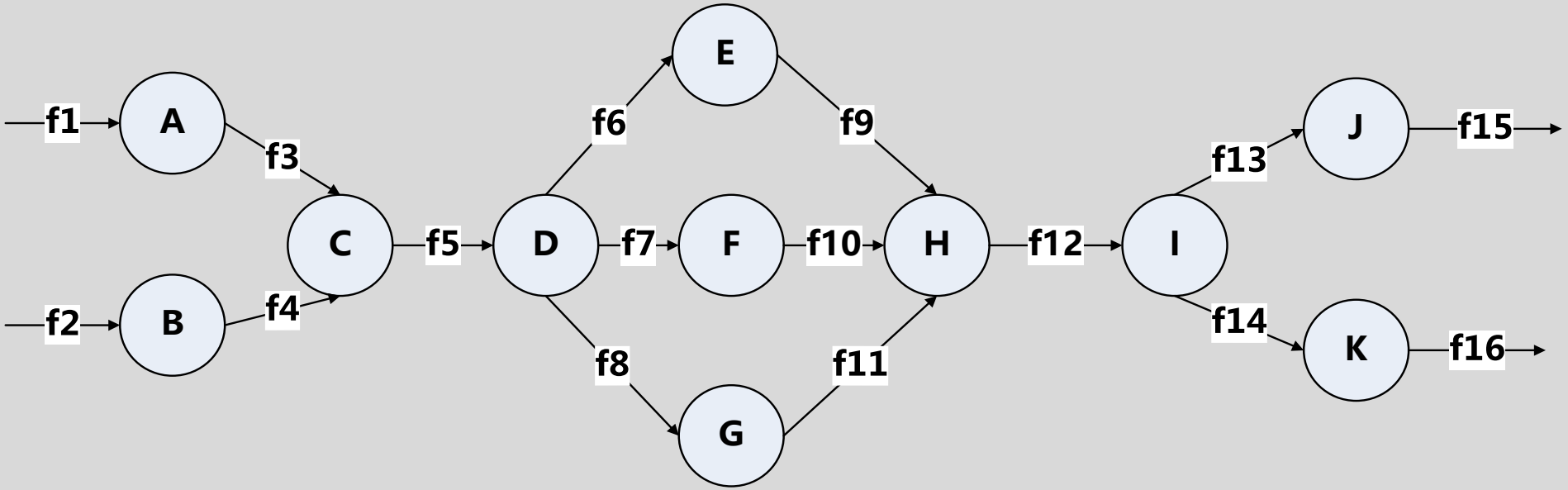
- 事务流应映射到包含一个输入分支和一个分类事务处理分支的程序结构上。
  - 输入分支结构的开发与变换流的方法类似
  - 分类事务处理分支结构包含一个调度模块，它调度和控制下属的事务处理模块。
    - 建立一个主模块用以代表整个加工，P层
    - 然后考虑被称为事务层的第二层模块，T层
    - 第二层模块只能是三类：取得事务、处理事务和给出结果。
    - 处理事务模块的下层为操作模块，A层
    - 操作模块之下为细节模块，D层

变换-事务混和型结构

- 一般来讲，一个大型的软件系统不可能是单一的数据变换型，也不可能是单一的事务型，通常是变换型结构和事务型结构的混合体。
- 在具体的应用中一般以变换型为主，事务型为辅的方式进行软件结构设计。



- 假定DH为以下数据流图的中心，请说明该中心的处理类型，并写出该中心的逻辑输入流和逻辑输出流，再将数据流图转换为对应的系统功能结构图。



第九章 程序实现

要点:

- 源程序文档化

源程序文档化

- 从软件工程的角度，对于源程序除了质量要求之外，为了后期代码的维护和更改，还必须从提高可阅读性，即达到源程序文档化。
  - 标识符命名：模块名（类名及方法名），变量/常量名
    - 名称需清楚表示具体的含义，采用添加前缀和后缀增加可阅读性；
    - 使用专业术语（业务词汇），注意编码语言的关键字冲突；
    - 注意大小写及长度；
  - 源程序布局
    - 编码之前定义统一的编码规范；
    - 规定合理的注释、缩进、空格、空行等方式；
  - 程序注释
    - 序言性注释
    - 功能性注释

序言性注释

- 注释的目的：解释程序的主要内容及难点说明；
- 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
- 主要包括以下内容：
  - 程序标题：模块名称
  - 模块描述：该模块的功能和目的说明
  - 主要算法：（option）说明算法结构
  - 接口说明：说明该模块与其他模块的调用关系
  - 开发简历：
    - 创建者、创建时间；
    - 修改者、修改时间、修改内容；
    - 版本

功能性注释

- 在序言性注释的基础上，对于程序体中复杂难于理解的程序结构进行局部说明；
  - 主要描述一段程序，必要时对某一段进行说明；
  - 修改代码的同时，对应的功能性注释也要进行修改；

第十章 软件测试

要点：

- **白盒测试**
- **黑盒测试**
- 测试基本类型

两种常用的测试方法

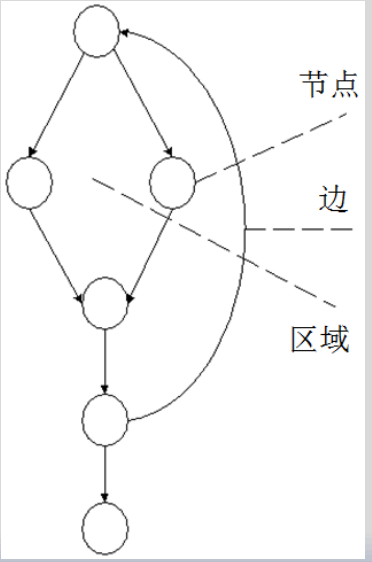
- 白盒测试：将测试对象看做一个透明的盒子，允许利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序所有逻辑路径进行测试。通过在不同点检查程序的状态，确定实际的状态是否与预期的状态一致，又称为结构测试或逻辑驱动测试。
- 黑盒测试：这种方法完全不考虑程序内部的逻辑结构和内部特性，只依据程序的需求规格说明书和概要设计说明，检查程序的功能是否符合它的功能说明，又称为功能测试或数据驱动测试

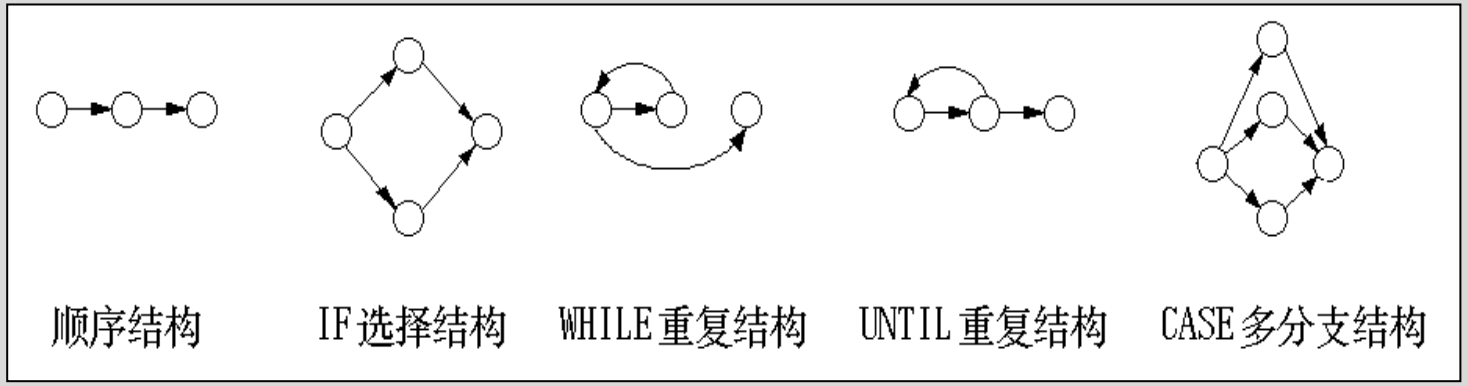
白盒测试方法

- 白盒测试主要应用于单元测试，是检查程序逻辑错误的主要方法。
- 使用白盒测试方法，主要对程序模块进行如下的检查：
  - 程序模块的所有独立的执行路径至少测试一次；
  - 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；
  - 在循环的边界和运行界限内执行循环体；
  - 测试内部数据结构的有效性，等。
- 逻辑覆盖：逻辑覆盖是以程序内部的逻辑结构为基础设计的测试用例技术
  - 语句覆盖
  - 判定覆盖
  - 条件覆盖
  - 判定+条件覆盖
  - 条件组合覆盖
  - 路径覆盖

基本路径测试

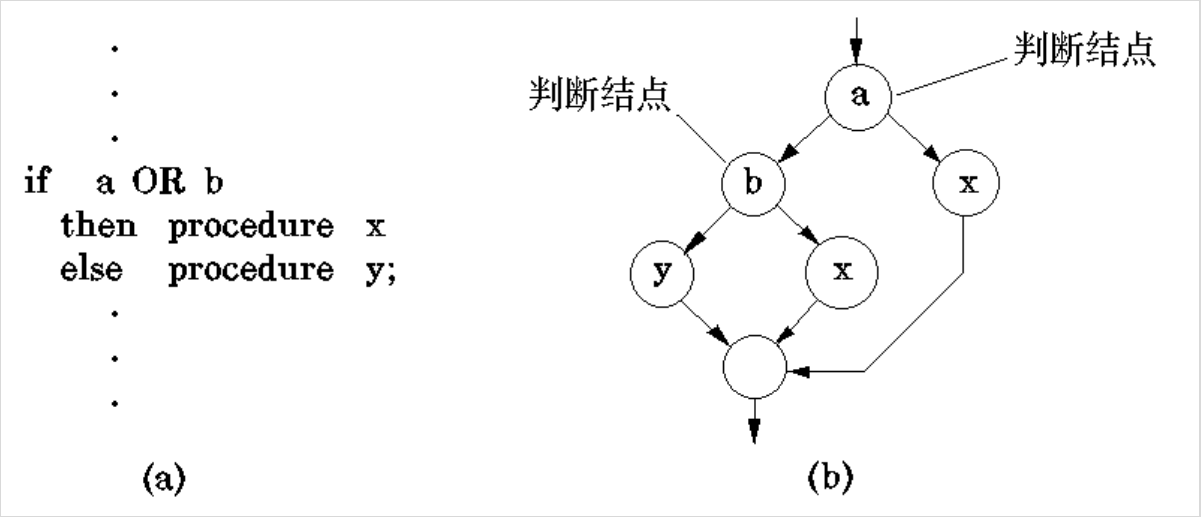
- 对于具有循环结构的程序而言，其路径数有可能很多，要求做到路径覆盖有难度。基本路径测试方法力图把覆盖的路径数压缩到一定限度内，使得程序中的循环体最多只执行一次。
- 这个方法需引入程序控制流图：基于程序流程图进行简化，得到程序的控制结构。
- 进而分析控制结构的环路复杂性，导出基本可执行路径集合，设计测试用例的方法。设计出的测试用例要保证在测试中，程序的每一个可执行语句至少要执行一次。





控制流图的转换

- 顺序结构的多个结点可以合并为一个结点。
- 在选择或多分支结构中，分支的汇聚处应有一个虚拟汇聚结点。
- 边和结点圈定的范围叫做区域，当对区域计数时，图形外的范围也应记为一个区域。
- 如果判断中的条件表达式是由一个或多个逻辑运算符 (OR，AND, NAND，NOR) 连接的复合条件表达式，则需要改为一系列只有单个条件的嵌套的判断。



控制流图的环路复杂度计算

- 控制流图的环路复杂度（也称为McCabe复杂度）确定了程序中独立路径的上界，以此为依据可以找出程序中的全部独立路径。
- 环路复杂度有三种计算方法：
  - 等于控制流图中的区域数，包括封闭区域和开放区域；
  - 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；
  - 若设P为控制流图中的判定结点数，则有 V(G) = P + 1。
- 基本路径集：指程序的控制流图中，从入口到出口的路径，该路径至少经历一个从未走过的边。
  - 基本路径集不是唯一的，对于给定的控制流图，可以得到不同的基本路径集。
  - 最大的基本路径条数就是环路复杂度。

导出测试用例

- 根据控制流图的基本路径导出测试用例，确保基本路径集中每一条路径的执行。
- 根据判断结点给出的条件，选择适当的数据以保证每一条路径可以被测试到，考虑使用逻辑覆盖方法。
- 每个测试用例执行之后，与预期结果进行比较。
- 如果所有测试用例都执行完毕，则可以确信程序中所有的可执行语句至少被执行了一次。

控制流图转换举例

```
void Func(int iRecordNum,int iType)
1{
2    int x=0;
3    int y=0;
4    while (iRecordNum > 0)
5     {
6        if(0= =iType)
7          {x=y+2; break;}
8        else
9            if (1= =iType)
10                {x=y+10; iRecordNum--;}
11           else
12               {x=y+20; iRecordNum--;}
13    }
14 }
```

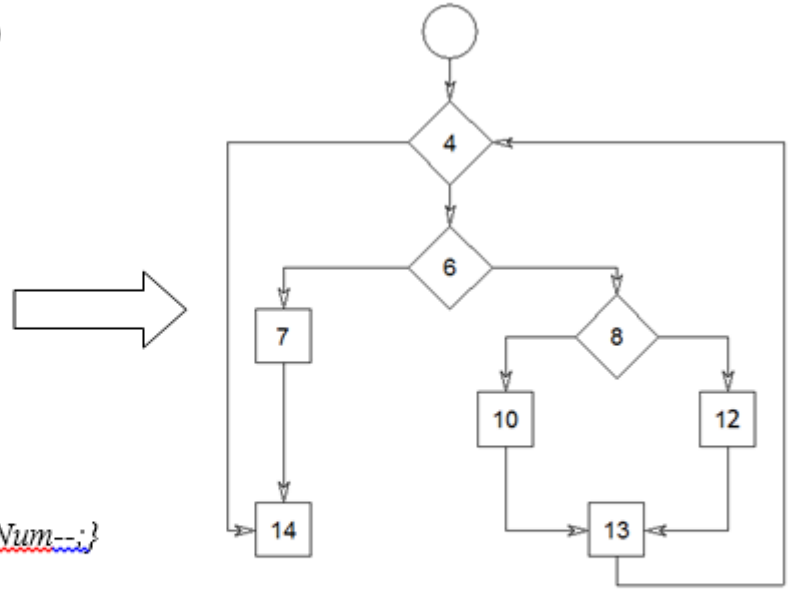
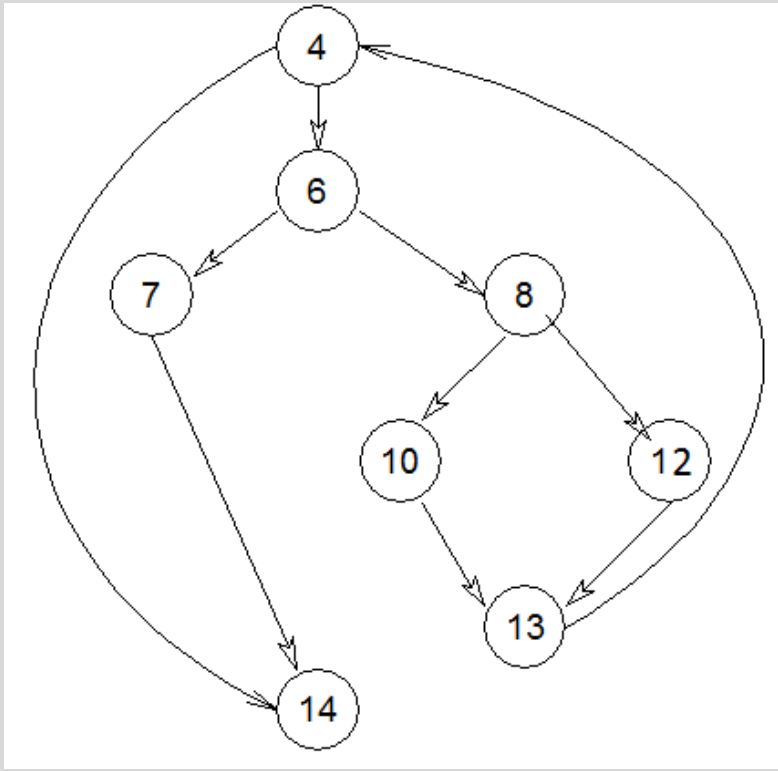


图 10-18 程序流程图



- 路径1：4-14　　输入数据：iRecordNum = 0；预期结果：x = 0；
- 路径2：4-6-7-14　　输入数据：iRecordNum = 1,iType = 0；预期结果：x = 2；
- 路径3：4-6-8-10-13-4-14　　输入数据：iRecordNum = 1,iType = 1；预期结果：x = 10；
- 路径4：4-6-8-12-13-4-14　　输入数据：iRecordNum = 1,iType = 2；预期结果：x = 20。

黑盒测试方法

- 相对于白盒测试，黑盒测试是在不需要了解程序结构的基础上，根据概要设计或者需求分析的结果进行测试用例的设计。常用的方法有：
  - **等价类划分**；
  - 边界值分析；
  - 因果图；
- 黑盒测试方法一般用于集成测试、系统测试和验收测试，某些特殊情况也会用到单元测试。
- 黑盒测试方法用于测试程序接口，主要是为了发现以下错误:
  - 是否有不正确或遗漏了的功能?
  - 在接口上，输入能否正确地接受? 能否输出正确的结果?
  - 是否有数据结构错误或外部信息(例如数据文件)访问错误?
  - 性能上是否能够满足要求?
  - 是否有初始化或终止性错误?

五个软件测试的基本类型

- 单元测试：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；
- 集成测试：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；
- 确认测试：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；
- 系统测试：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；
- 验收测试：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。