

本文档对《软件工程》课程作出简明复习。

本文约定

墨蓝色粗体表示首次出现的概念，蓝色下划线表示一定程度上的重点，灰色表示较琐碎的知识点。

目录

1	概述	3
1.1	软件工程、软件危机	3
1.2	软件生命周期等	3
2	软件定义·可行性研究	4
2.1	何为可行性研究	4
2.2	各种图表	4
2.3	成本/效益分析	5
3	软件定义·需求分析	5
3.1	何为需求分析	5
3.2	结构化分析方法 (SA)	7
3.3	需求分析常用图形	7
3.4	本章实例	8
4	软件开发·总体设计	8
4.1	设计过程	8
4.2	设计原理	8
4.2.1	耦合	9
4.2.2	内聚	10
4.3	启发规则	11
4.4	图形工具、面向数据流设计方法	11

5	软件开发·详细设计	12
5.1	结构程序设计	12
5.2	人机界面设计	12
5.3	详细设计的工具	12
6	软件开发·编码	13
7	软件开发与维护·测试	13
7.1	单元测试	13
7.2	集成测试与确认测试	14
7.3	白盒测试与黑盒测试	14
8	软件维护·维护	15
9	面向对象方法学	16
9.1	何为面向对象方法学	16
9.2	对象模型	18
9.3	动态模型、功能模型	19
10	统一建模语言 (UML)	19

1 概述

1.1 软件工程、软件危机

软件的构成：

$$\text{软件} = \text{程序} + \text{数据} + \text{文档} \quad (1)$$

软件的特点：

1. 软件，逻辑的实体；
2. 开发，智力的发挥；
3. 维护，不同于维修；
4. 成本，昂贵而非凡；
5. 开发，尚未摆脱手工艺过程，过程复杂。

软件工程和编程的区别：软件工程是一门学科，考虑分解一个系统；编程只是一种写代码行为，占据很少时空。

软件危机是指软件开发和维护中存在的一系列问题。它主要面临的问题是：进展难衡量、质量难评估、管理难控制。

软件工程是指开发、运行、维护、修复软件的系统方法（1983IEEE 定义）。

软件工程的本质特性（共 7 条）：主要是由一种有文化背景的人替另一种有文化背景的人创造产品。

软件工程 7 条基本原理：

周期严管、阶段评审、产品控制、现代设计、结果可视、团队精炼、持续改进。

软件工程方法学是指软件在生命周期全过程中使用的一整套技术的集合，也叫范型。

$$\text{软件工程（方法学）} = \text{方法} + \text{工具} + \text{过程} \quad (2)$$

目前使用得最广泛的两种软件工程方法学：传统方法学、面向对象方法学。

1.2 软件生命周期等

软件生命周期划分：

- 软件定义/孕育期——(1) 可行性研究与计划、(2) 需求分析；
- 软件开发/成长期——(3) 总体设计、(4) 详细设计、(5) 编码实现、(6) 集成测试；
- 软件维护/成熟期——(7) 确认测试、(8) 使用与维护。

软件过程与质量评价 **软件开发模型**是指软件开发全过程的结构框架，明确的是主要活动、任务、开发策略。

软件开发模型举例：

「瀑布模型」、「快速原型模型」、「螺旋模型」。

质量和生产率的优先级：质量第一，生产率第二。

2 软件定义·可行性研究

2.1 何为可行性研究

可行性研究的任务是确定问题是否能解决，用最小代价和尽可能短的时间。（可行性研究成本 5% – 10%）

可行性研究具体包括：[技术可行性](#)、[经济可行性](#)、[操作可行性](#)、社会因素等。

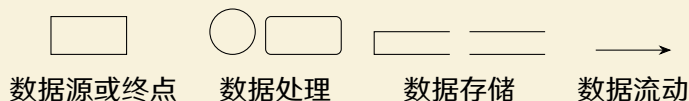
可行性研究的过程：

复查问题、研究现有系统、导出新模型、重定义问题、评价方案、推荐方案、草拟计划、书写文档、提交审查。

2.2 各种图表

系统流程图 **系统流程图**就是我们熟悉的流程图，是描绘[物理系统](#)的传统工具。

数据流图 (DFD) **数据流图** (Data Flow Diagram) 是一种描述[逻辑模型](#)的图形工具。DFD 可以表示系统和软件在[任何一个](#)层次上的抽象。



画 DFD 的方法：

1. 确定系统的输入输出；
2. [由外向里](#)画顶层数据流图；
3. 自顶向下逐层分解。

实例：仓库订货系统、「口算高手」、客房管理系统（详见课件）。

数据字典 (DD) **数据字典** (Data Dictionary) 用来对数据流图的所有被命名的元素加以定义。它也是一种描述[逻辑模型](#)的工具。

在数据字典中，我们会自顶向下地对一个字段的规则进行分解。常用规则：

- $a + b$ ——由 a 和 b 构成；
- $[a|b]$ ——由 a 或 b 构成；
- (a) ——元素 a 出现零次或一次；
- $\{a\}$ ——元素 a 出现零次或多次；
- $m\{a\}n$ ——元素 a 出现 $m \sim n$ 次（当 $m = n$ 时即次数固定）；
- $a \dots b$ ——取值为 $a \sim b$ 的任一个值；
- “ a ”——取值 a 已经是最基本的，无需再度定义。

例如在学生一课程数据库中的数据字典可能的情况如下所示：

- 学号/Sno = "2022" + 7{数字}7, 数字 = "0" ... "9"
- 姓名/Sname = $\left[2\{\text{汉字}\}6 \mid 1\{\text{汉字}\}8 + " \cdot " + 1\{\text{汉字}\}8 \right]$ （考虑少数民族名）
- 性别/Ssex = ["男"|"女"]
- 成绩/Grade = "0" ... "100"

2.3 成本/效益分析

成本/效益分析是为了从经济角度评估开发一个新项目的可行性。

成本估计一般用代码行技术、任务分解技术。

效益估计一般用货币的时间价值、投资回收期、纯投入等。

可行性报告参考格式：GB8567-88。

3 软件定义·需求分析

《原神》中的传奇工匠希诺宁，几乎什么工具都能打造。但对她来说，最难面对的客户是不知道自己想要的东西有多复杂的人。因此在开始打造以前，希诺宁总需要花费大量的精力帮助客户捋清需求。可见，需求分析是十分关键的一环。

3.1 何为需求分析

需求分析的目的是澄清客户需求，并把开发者和客户双方的理解表述成一份《软件需求规格说明书》。

客户包括提出要求、支付款项、选择、具体说明或使用软件产品的项目风险承担者（stakeholder）或是获得产品所产生结果的人。

需求分析的具体任务包括：

1. 确定综合需求；
2. 分析数据需求；
3. 导出逻辑模型；
4. 修正开发计划；
5. 验证分析正确性；
6. 编写需求说明书。

软件的综合需求一般包括：

1. [功能](#)需求
2. [性能](#)需求
3. [可靠性可用性](#)需求
4. 出错处理需求
5. 接口需求
6. [约束](#)（环境约束、界面约束、用户因素约束、文档约束、数据约束、资源约束、安全保密要求约束、软件成本消耗与开发进度需求约束）
7. 逆向需求
8. 未来潜在需求

需求获取的常见方法：

1. 访谈
2. 面向数据流自顶向下求精
3. 应用规格说明

需求分析实例：「口算高手」软件、远程路灯照明系统（详见课件）。

需求分析的四个阶段：

1. [调查研究](#)——需求调查，补充数据字典，修改 IPO 图
2. [分析与综合](#)——追踪数据流图，复查系统逻辑模型
3. [书写需求分析文档](#)——书写系统规格、数据要求、用户系统描述等文档
4. [需求分析评审](#)

需求分析的步骤如图 1 所示。

需求分析阶段常用的逻辑模型：[数据流图 \(DFD\)](#)、[E-R 图](#)、类图、时序图、状态图、协作图等。

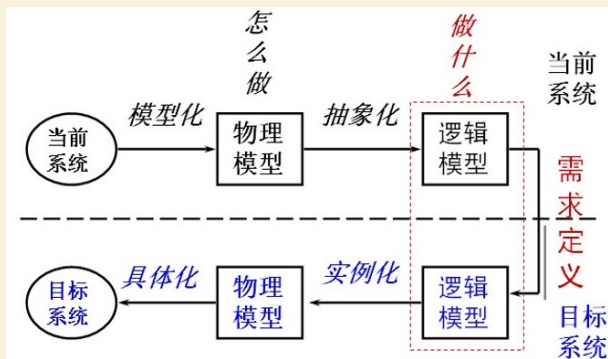





图 1: 需求分析的步骤

3.2 结构化分析方法 (SA)

结构化分析方法 (Structured Analysis) 是一种面向数据流、自顶向下、逐步求精的方法。这种方法使用的建模工具有[数据流图 \(DFD\)](#)、[数据字典 \(DD\)](#)、结构化英语、判定表、判定树等。


3.3 需求分析常用图形

E-R 图 实体—联系图，即 **E-R 图**，也就是「数据模型」¹，是我们熟悉的一种概念模型，它是逻辑模型的一种。

E-R 图的三要素： **实体对象**、 **属性**、 **联系**。

数据规范化用范式表示，分为 1NF 至 5NF，其中 1NF 冗余程度最大，5NF 冗余程度最小。

状态转换图 **状态转换图**是描绘系统状态及其转换关系的图，它是逻辑模型的一种。这种图所包含的初始状态**有且只有 1 个**，终止状态有 0 个到多个，中间状态若干。我们在《数字逻辑》《编译原理》等课程已经见过状态转换图了。

其符号为：• 初态、● 终态、 中间态。

层次方框图 层次方框图是一种用树形结构描述数据层级结构的逻辑模型。



IPO 图 输入—处理—输出图，即**IPO 图**，是一种描述数据的输入、处理、输出关系的图形工具，同样属于逻辑模型。

¹第四章课件的第三章回顾部分提到了，数据模型也称 E-R 模型。

3.4 本章实例

需求分析报告书写同样遵循自顶向下原则。参考 GB856T-88。

本章实例：考务处理系统（详见课件）。其绘制过程如下：

1. 明确系统功能
2. 绘制顶层数据流图 (DFD)：三个  数据源和一个  数据处理枢纽。（从上数第一层）
3. 细化 0 层数据流图。（从上数第二层）
4. 细化 1 层数据流图。（从上数第三层）

4 软件开发·总体设计

4.1 设计过程

总体设计也称为概要设计、初步设计，要考虑「怎么做」的问题。（上一章的需求分析考虑的是「做什么」的问题。）

总体设计的主要任务是：划分物理元素、确定软件结构。

总体设计两个阶段：[系统设计](#)、[结构设计](#)。

总体设计总共九个步骤：（其中 1-3 为系统设计）

1. 设想供选择的方案；
2. 选取合理的方案；
3. 推荐最佳方案；
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. 检查和复审。

4.2 设计原理

模块化 **模块化设计**就是将大型软件划分成一个个小模块的设计方法。重要指导思想是功能分解、信息隐藏和模块独立性。

成本与模块的关系呈「V 形」曲线，即合理的模块数目可以使成本最小化，过多或过少都不合适。

控制结构 **控制结构**描绘了软件模块间的相连关系，用图结构表示。常见度量：

- **扇出**：一个模块直接调用的模块数（图的最大出度）
- **扇入**：调用一个模块的上层模块数（图的最大入度）
- **深度**：模块的层数（图的深度）
- **宽度**：同一层的最大模块数

抽象 抽象指反映本质特征而忽略细节。可能分为多层次抽象。

信息隐藏 模块内部包含的信息，对于不需要这些信息的模块来说是不能访问的。

模块独立性 以内聚与耦合定性度量模块独立性。

4.2.1 耦合

耦合用于衡量不同模块之间相互依赖的紧密程度，**越低越好**。应该追求「松散耦合」的系统。从低耦合到高耦合的七种类型如下所示，示意图如图 2 所示。

1. **非直接耦合/无耦合**：两个模块没有直接关系，最低耦合。
2. **数据耦合**：两个模块传递的都是简单的数据，松散耦合。
3. **特征耦合/标记耦合**：两个模块传递的是数据结构（数组、记录等）或都与一个数据结构有关。特征耦合可以修改为数据耦合。
4. **控制耦合**：上层模块向下层模块传递的信息会控制下层模块的调用逻辑。去除控制耦合的好方法是，将下层模块的判定逻辑上移到上层模块中，「先判定再调用」；以及把下层模块分解。
5. **外部耦合**：模块与系统外部（I/O 设备等）关联。外部耦合**必不可少**，但是数量应尽量少。
6. **公共耦合**：一组模块引用同一个公用数据区，如全局变量、共享内存等。这种耦合须慎用。
7. **内容耦合**：两个模块之间互相访问内部信息或代码重叠，或者一个模块多入口。这种耦合是最糟糕的。

耦合设计原则：

- 多用数据耦合
- 少用控制耦合
- 限制公共耦合

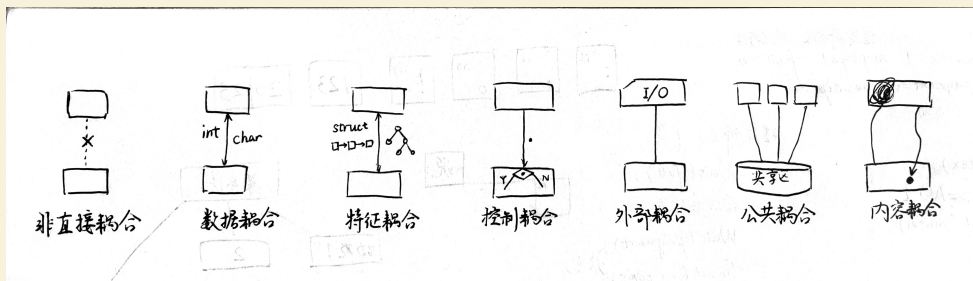


图 2：七种耦合类型示意图

- 避免内容耦合

4.2.2 内聚

内聚用于衡量一个模块内部各元素彼此结合的紧密程度，越高越好。从低内聚到高内聚的七种类型如下：

1. **偶然内聚/巧合内聚**：模块内的语句没有任何联系。
2. **逻辑内聚**：将逻辑相似的功能组合在一个模块内。缺点是增加控制耦合，修改困难。
3. **时间内聚**：模块完成的功能必须在同一时间段内执行。
4. **过程内聚**：模块内各处理成分相关，且必须顺序执行。
5. **通信内聚**：模块内各部分使用相同的输入，或产生相同的输出。
6. **顺序内聚/信息内聚**：模块完成的功能都在同一数据结构上操作，且每个功能有唯一入口。
7. **功能内聚/理想内聚**：模块所有成分共同完成一个功能，缺一不可，也无冗余。最棒的内聚。

笔者小记：判断何种「内聚」相当于是判断以什么理由将各个成分绑定在一个模块内。偶然内聚的理由根本没有，逻辑内聚的理由只是「看起来」逻辑相似，时间内聚的理由只是要在同一个时间段处理——这三者都属低内聚。过程内聚的理由是需要有先后关系，通信内聚的理由是需要处理同样的输入输出——这两者属于中内聚。顺序内聚的理由是操作对象数据结构相同，而功能内聚的理由即为实现的功能本身——这两种属于高内聚。

内聚设计原则：

- 力求高内聚
- 可用中内聚
- 避免低内聚

内聚和耦合优先级 实践表明，**内聚**更重要些，高内聚意味着松耦合。

4.3 启发规则

常用的启发规则有：

1. 提高模块独立性——提高内聚，降低耦合。
2. 控制模块规模适中——过大则分解不充分，过小则接口复杂。
3. 深度、宽度、扇入、扇出应适中。
4. 模块的**作用域在控制域之内**——保证模块可控。
5. 降低接口的复杂程度。
6. 设计单入口、单出口模块。
7. 模块功能应该可预测。

模块的**作用域**是指受该模块的一个判断所影响的模块之集合；**控制域**是指控制结构图中该模块本身与其子模块的集合，也就是「子树」。改变作用域和控制域方法：判断点上移，作用域对象下移。

4.4 图形工具、面向数据流设计方法

层次图 **层次图**用于描绘软件的层次结构，矩形代表模块，连接代表调用关系。

HIPO 图 **HIPO 图**就是「层次 (Hierarchy) 图 + 输入/处理/输出 (IPO) 图」的变种。相当于外层是层次图，每个模块加以编号；而内层即每个模块内部各有一个 IPO 图。

结构图 (SC) **结构图** (Structure Chart) 用于描绘模块间的信息传递关系，箭头表示调用关系，带注释的箭头表示传递的信息。层次图有传入、传出、变换、协调共四种模块，有选择调用和循环调用机制。

面向数据流设计方法 数据流图分为「**变换型**」和「**事务型**」。变换型结构由输入、变换中心、输出三部分组成，而事务型结构由接受路径、事务中心、动作路径等组成。

变换型数据流图 (DFD) 经过变换分析可以得到结构图 (SC)。事务型数据流图经过事务分析也可以得到结构图。

体系结构设计优化 改进软件结构设计有 9 条原则。例如，消除重复功能，将作用域限制在控制域之内，模块大小适中等。

5 软件开发·详细设计

详细设计的目的是确定具体编程方案。

5.1 结构程序设计

结构程序设计采用自顶向下、逐步求精的设计方法和单入口、单出口的控制结构。

- 经典结构程序设计：顺序、选择、循环。理论上最基本的控制结构为顺序和循环。
- 扩展结构程序设计：do-case（分支）、do-until
- 修正结构程序设计：break、continue、goto（少用）

5.2 人机界面设计

人机界面设计的 4 个设计问题：

1. 系统响应时间
2. 用户帮助设施
3. 出错信息处理
4. 命令交互

人机界面设计指南：

1. 一般交互指南
2. 信息显示指南
3. 数据输入指南

5.3 详细设计的工具

详细设计的典型工具有：

- [程序流程图](#)
- [盒图/N-S 图](#)
- [PAD 图](#)（问题分析图）
- 结构化英语
- 判定表、判定树
- 过程设计语言 (PDL)

其中盒图中的有些符号需要留意，如分支判断的结果为假写在左边，为真写在右边；直到型循环和当型循环的折角方向也不一样。

判定表由四部分组成：条件桩、操作桩、条件条目、操作条目。

可能需要掌握例子如一元二次方程求解过程的程序流程图、盒图画法。

6 软件开发·编码

由于课件的原型不晚于 2010 年，大约在 2003 年前后，因此课件内举的一些例子（如程序设计语言）略显陈旧。

编码就是用程序设计语言书写程序。

程序设计语言可分为机器语言、汇编语言、高级语言、第四代语言。

选择语言要求：用户要求、编译程序要求、软件工具要求、工程规模要求、程序员知识、可移植性、应用领域等。

编码风格规则：程序内部的文档、数据说明、语句构造、输入输出、效率等。

7 软件开发与维护·测试

测试是为了发现程序中的错误而执行程序的过程。然而测试成功并不能证明程序一定没有 bug。

测试的基本步骤：模块测试、子系统测试、系统测试、验收测试、平行运行。

7.1 单元测试

单元测试用详细设计描述作为指南，对重要的数据通路进行测试。可以使用[白盒测试](#)法。

单元测试的重点：

- 模块接口调试
- 局部数据结构测试
- 重要的执行通路测试
- 出错处理通路测试
- 边界条件测试（参加过 CSP 考试的应该深有体会）

7.2 集成测试与确认测试

集成测试是指测试和组装软件的系统化技术，主要目标是发现与[接口](#)相关的问题。两种方法：非渐增式测试、渐增式测试。两种结合方式：自顶向下结合、自底向上结合。

假设有 4 个模块，分别编号为 1,2,3,4，那么两种方法的测试流程可能如下：

- 非渐增式测试：分别测试 1, 2, 3, 4, 1234。
- 渐增式测试：分别测试 1, 12, 123, 1234。

确认测试也称**验收测试**，是为了检测软件的有效性。一般使用[黑盒测试](#)法。在需求分析阶段产生的文档是软件有效的标准，也是验收测试的基础。

以某游戏的 Alpha 和 Beta 测试为例：

- Alpha 测试：也称「内测」，环境相对封闭，通常是公司内部工作人员参与，对外界保密。
- Beta 测试：也称「公测」，环境相对开放，通常需要一些玩家的游玩体验与反馈意见。

7.3 白盒测试与黑盒测试

白盒测试 **白盒测试**也称**结构测试**，主要用于检测软件编码过程中的错误，用于前期测试。

白盒测试的测试用例设计，采用[逻辑覆盖法](#)，由弱到强的偏序关系分为：

1. [语句覆盖](#)：每条语句至少执行一次。
2. [判定覆盖/分支覆盖](#)：每次判定的真假分支至少执行一次。
3. [条件覆盖](#)：每次判定的每个条件的可能取值至少执行一次。例如，对 $(x > 3) \text{AND} (y < 2)?$ 的条件，测试用例至少要包括 $x > 3$ 、 $x \leq 3$ 、 $y < 2$ 、 $y \geq 2$ 的取值。
4. [判定一条件覆盖](#)：判定覆盖 + 条件覆盖。
5. [条件组合覆盖](#)：每个判定表达式的各种可能组合至少出现一次。
6. 点覆盖、边覆盖、路径覆盖：至少经过流图的每个点/边/路径一次。

黑盒测试 **黑盒测试**也称**功能测试**，主要检测软件的每一个功能是否能够正常使用，用于后期测试。

划分「等价类」的几条启发式规则，这里用形式化定义表述，请结合课件观看：

1. 规定输入值范围为连续值 $[x, y]$ ，假设输入值为 I ，则划分出三个等价类：
 - 有效等价类： $I \in [x, y]$ ；
 - 无效等价类： $I \in (-\infty, x)$ 、 $I \in (y, \infty)$ 。
2. 规定输入数据个数 $a[0] \sim a[x]$ ，假设测试的下标为 I ，则划分出三个等价类：
 - 有效等价类： $I \in [0, x]$
 - 无效等价类： $I \in (-\infty, 0)$ 、 $I \in [x, \infty)$
3. 规定输入值范围为离散值 $A = \{a_1, a_2, \dots, a_n\}$ ，假设输入值为 I ，则划分出两个等价类：
 - 有效等价类： $I \in A$
 - 无效等价类： $I \notin A$
4. 规定输入值的数据类型或规则 r_0 ，假设输入数据的数据类型或规则满足 I ，则划分出若干个等价类：
 - 有效等价类： $I = r_0$
 - 无效等价类： $I = r_1, r_2, r_3 \dots$ （总之 $I \neq r_0$ ）
5. 规定输入值范围为连续值 $[x, y]$ ，其中 $x < 0$ ， $y > 0$ ，假设输入值为 I ，则划分为五个等价类：
 - 有效等价类： $I \in [x, 0)$ 、 $I = \{0\}$ 、 $I \in (0, y]$
 - 无效等价类： $I \in (-\infty, x)$ 、 $I \in (y, \infty)$ 。

总的来说，无效等价类就是有效等价类的补集。

在黑盒测试中，需要为有效等价类设计测试用例，还需要为每个无效等价类分别设计测试用例，因此理论上最少测试用例数量 = 无效等价类数量 + 1。

8 软件维护 · 维护

软件**维护**是指为了改正错误（bug）或满足新的需要而修改软件的过程。

维护的种类：

- **纠错性/改正性**维护——例如修正一个在开发阶段已发现的 bug
- **适应性**维护——例如本来在 Win10 操作系统下运行的软件，开发 Win11 的 API
- **完善性**维护——例如电商平台改进搜索算法，使结果更贴合用户意图

- **预防性**维护——例如对老旧项目代码进行重构（如更新依赖库）

【小测试】以下是游戏《原神》5.5 版本「众火溯还之日」更新公告节录，请对以下每一条更新判别分类：

1. 「纳塔」地区开放新地图「安饶之野」「远古圣山」——???
2. 修复了角色「绮梦缙缙·梦见月瑞希（风）」施放下落攻击后，武器显示异常的问题——???
3. 「神瞳共鸣石」成功寻找到神瞳后的冷却时间由 5 分钟调整为 2 分钟——???
4. 增加了对 iPad Air (M3) 和 iPad (A16) 设备的适配支持——???

答：1. 完善性维护；2. 纠错性维护；3. 完善性维护；4. 适应性维护

软件维护的特点：结构化维护和非结构化维护差别巨大、维护代价高昂、维护的问题很多等。

决定软件可维护性的因素：可理解性、可测试性、可修改性、可移植性、可重用性。

9 面向对象方法学

9.1 何为面向对象方法学

在**面向对象方法学**中，任何事物都是对象，对象分解取代了功能分解。
面向对象方法学的要点（总结为一个等式）：

$$\text{面向对象} = \text{对象} + \text{类} + \text{继承} + \text{消息} \quad (3)$$

面向对象方法学的优点：

- 与人类习惯的思维方式一致
- 稳定性好
- 可重用性好
- 较易开发大型软件产品
- 可维护性好

相关概念 类与对象的基本认识：

- **对象**是对某个实体的抽象，形式化定义为〈对象名, 操作, 数据结构, 受理消息〉。其特点是：

以数据为中心、对象是主动的、实现的数据封装、本质上具有并行性、模块独立性好。

- **类**是对相同属性和行为的一个或多个对象的描述。
- **实例**是某个特定的类描述的一个具体的对象。
- **消息**是对象间交互的核心机制。通过消息，对象可以请求其他对象执行特定操作。一个消息通常由三部分组成：[接收对象](#)、[方法名](#)、[参数列表](#)。
- **方法**是对象所能执行的操作。在 C++ 中，「方法」也叫做「成员函数」。
- **属性**是类中所定义的数据项。

面向对象的三大特征：继承性、封装性、多态性。

- **封装**是指把数据和方法集中放置于对象的内部。
- **继承**是指子类能够获得父类的一些已有性质和特征。
- **多态**是指允许对象以多种形式出现。

多态性似乎有些难以理解，举个例子：

```
1 public class Main{
2     void show(int i){
3         System.out.println("Integer:␣" + i); //输出语句
4     }
5     void show(double d){
6         System.out.println("Double:␣" + d);
7     }
8     void show(String s){
9         System.out.println("String:␣" + s);
10    }
11 }
```

这个 Java 代码块的 Main 类的 show 方法允许接收不同类型的参数，这就体现了多态性，同时也是函数重载的一个例子。

重载是实现多态性的保证，其主要分为两种类型：

- 函数重载：同一个函数名可以有不同的参数列表；
- 运算符重载：允许用户重定义既有运算符的行为。

采用**面向对象建模**开发软件，通常需要建立以下三种模型：

1. [对象模型](#)——描述系统数据结构
2. [动态模型](#)——描述系统控制结构

3. [功能模型](#)——描述系统功能

这三种模型的关系：相互补充，相互配合。（课件 9.7 小节）

- 1. 对象模型——定义了做事情的实体
- 2. 动态模型——规定了什么时候做事情
- 3. 功能模型——指明了系统应该做什么

9.2 对象模型

[对象模型](#)是对对象及其关系的映射，描述了系统的静态结构。通常使用统一建模语言 (UML) 的[类图](#)来建立对象模型。

类图的基本符号，从上到下共三层，如表 1 所示。

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

表 1: UML 类图的画法

其中前面可能冠以符号代表可见性：+ 代表 public，- 代表 private，# 代表 protected。

类与类之间可能存在的关系，如表 2 所示。

需要指出的是，聚合是一种特殊的关联，组合是一种特殊的聚合。

此外对于关联关系，我们可能会在箭头的两端写上重数：

- 0...1 ——表示零个或一个
- 0...* 或 * ——表示零个或多个
- 1...* 或 1+ ——表示一个或多个
- *n* ——表示 *n* 个
- *m*...*n* ——表示 *m* 个到 *n* 个

例如，一个作家使用一台或多台计算机，而一个计算机被零个或多个作家使用。

【小测试】在横线内填写合适的关系。备选：A. 关联；B. 聚合；C. 泛化；D. 实现；E. 依赖

关系类型	强度	生命周期依赖	举例（UML 画法）	解释
泛化/继承	最强	子类继承父类	狗 \rightarrow 动物	狗「是一种 (is-a)」动物
组合	强	部分随整体销毁	窗口 \blacktriangleleft 菜单	窗口销毁时菜单也不存在
聚合/聚集	中	整体部分关系	教室 \circ 学生	学生独立于教室存在
关联	弱	语义上有关系	教师 — 课程	教师与课程长期关联
依赖	最弱	临时使用	报告生成器 - \rightarrow 数据	报告生成器类临时依赖数据类
细化	最弱	描述层次更细	设计类 - \rightarrow 分析类	设计类在分析类基础上详细描述

表 2: 类与类的关系表

1. 在图书管理系统中，读者包含教工和学生，其中教工可以借 10 本书，本科生可以借 6 本书，则本科生和书之间是____关系，读者和本科生是____关系。
2. 交通工具与卡车之间是____关系，卡车与发动机之间是____关系。
3. 公司与部门之间是____关系。
4. 人借助船渡河，人与船之间是____关系。

答：1. A. 关联、C. 泛化；2. C. 泛化、B. 聚合（或组合聚集）；3. B. 聚合；4. E. 依赖

9.3 动态模型、功能模型

动态模型是一种特别的[状态图](#)，描述对象在其生命周期的各种状态，以及状态之间的转换关系。

功能模型直接反映用户对目标系统的要求，由一组[数据流图](#)(DFD) 或者 UML 中的[用例图](#)组成。以用例图建立的功能模型，也叫做[用例模型](#)。一幅用例图所包含的模型元素有：系统、行为者、用例及用例之间的关系。

10 统一建模语言 (UML)

统一建模语言 (Unified Modeling Language) 是一个通用的可视化建模语言，是用于对软件进行描述、可视化处理、构造和建立软件系统支配的文档。

UML 包含 5 种视图：

用例视图、逻辑视图、组件视图、实现视图、部署视图

课件给出的 UML 的图例：

- 用例图
- 活动图
- 顺序图
- 协作图
- 类图
- 状态图
- 组件图
- 部署图

E-R 图, 7

HIPO 图, 11

IPO 图, 7

事务型, 11

人机界面设计, 12

作用域, 11

内聚, 10

功能模型, 19

功能测试, 14

动态模型, 19

单元测试, 13

变换型, 11

可行性研究, 4

多态, 17

实体—联系图, 7

实例, 17

客户, 6

对象, 16

对象模型, 18

封装, 17

层次图, 11

属性, 17

总体设计, 8

成本/效益分析, 5

控制域, 11

控制结构, 9

数据字典, 5

数据流图, 4

方法, 17

模块化设计, 8

测试, 13

消息, 17

状态转换图, 7

用例模型, 19

白盒测试, 14

确认测试, 14

类, 17

系统流程图, 4

结构化分析方法, 7

结构图, 11

结构测试, 14

结构程序设计, 12

统一建模语言, 19

继承, 17

维护, 15

编码, 13

耦合, 9

详细设计, 12

软件危机, 3

软件工程, 3

软件工程方法学, 3

软件开发模型, 4

软件生命周期, 3

重载, 17

集成测试, 14

需求分析, 5

面向对象建模, 17

面向对象方法学, 16

验收测试, 14

黑盒测试, 14