本文档**仅供学习交流使用**，内容可能存在疏漏、错误或个人理解偏差，请以课本和老师讲解为准。如果你觉得它对你有帮助，我们会非常开心！预祝大家考试顺利！

一、 软件开发方法（简答）

1.软件的概念

(1) 软件的概念

是以能够完成预定功能和性能要求的可执行的计算机程序为核心，以使程序正常执行所需要的数据为基础，以描述与程序开发、维护和使用有关文档为指南，从而方便相关涉众理解、使用与维护的人工制品。

(2) 构成和特点

●构成：由程序（指令序列）、文档（图文材料）和数据（程序处理的信息）三部分组成。

●特点：**复杂性**：规模庞大、逻辑复杂。**一致性**：需遵循内外部统一的接口和规范。**易变性**：需求和环境易变，导致软件经常需要变更。**不可见性**：软件的运行逻辑是无形的、不可直接观察的。

(3) 软件的分类

●系统软件：负责控制和协调计算机硬件，为其他软件提供基础运行环境。例如：操作系统、数据库管理系统、驱动程序。

●应用软件：为满足用户特定需求而开发的软件。例如：办公软件、Web 应用、游戏等。

●支撑软件（工具软件）：介于系统软件和应用软件之间，辅助软件的开发、维护和运行。例如：编译器、集成开发环境(IDE)。

●嵌入式软件：指与硬件设备深度耦合的，用于控制特定硬件设备的软件系统。

●信息安全软件：用于保护计算机系统和数据安全的软件

2.软件的生存周期（概念/阶段）

●概念：指软件从最初构思、开发、运行、维护直到最终被淘汰的整个生命过程。

●主要阶段：① 定义时期（软件计划、需求分析）；② 开发时期（设计、编码、测试）；③ 使用和维护时期。

3.软件质量

(1) McCall 模型

●核心思想：从三个维度来衡量软件质量。

●三维度：**产品运行**（操作特性）：包括正确性、可靠性、效率、易用性。**产品修改**（承受变更的能力）：包括可维护性、灵活性、可测试性。**产品转移**（对新环境的适应能力）：包括可移植性、可复用性、互操作性。

(2) Garvin 质量维度

●核心思想：从更偏向于用户和市场的角度，用 8 个维度来评价产品质量。

●八维度：性能、特征、可靠性、符合性、耐用性、适用性、美学、感知质量。

(3) ISO/IEC25010 质量因素

●核心思想：目前较新的国际标准，定义了 8 个主要质量特性。

●八特性：功能性、性能效率、兼容性、易用性、可靠性、安全性、可维护性、可移植性。

4.软件质量保证

(1) 质量保证思想

通过对软件开发的全过程进行一系列的计划、评审和测试活动，来预防和发现缺陷，以确保最终产品符合质量标准。它强调的是"过程控制"。

(2) 方法

●制定质量标准：为编程设计测试等活动设立需要遵守的规范。

●软件评审：组织人员对需求、设计、代码等产物进行检查，如代码走查、设计评审。

●软件测试：通过执行程序来发现缺陷，包含单元、集成、系统、验收等多种测试活动。

●静态分析：在不运行代码的情况下，使用工具分析其结构和逻辑以发现潜在问题。

●代码重用：通过复用高质量、经过测试的代码来降低引入新错误的风险。

二、 软件工程概述

1. 软件工程概念（发展、目标、原则）

●概念：一门用系统化、规范化、数量化的工程原则和方法，来进行软件开发和维护的学科。

●发展特点：抽象层次越来越高，重用粒度越来越大（从代码片段 -> 组件 -> 服务）。开发理念不断演化（如从瀑布到敏捷的演化）。

●目标：在有限的成本和时间内，创造出"足够好"的、能满足用户需求的软件。

●核心原则：**分阶段管理**：使用分阶段的生命周期计划来严格管理项目。**阶段评审**：坚持在每个阶段结束时进行评审，及早发现问题。**产品控制**：实行严格的版本和配置管理。**抽象与封装**：通过抽象隐藏实现细节，通过封装将数据和功能打包。**关注点分离**：将不同功能和职责分解到独立的模块中。

2. 软件危机

(1) 软件开发的特殊性

软件开发从早期的"作坊式"个体编程，发展到如今的大规模、大批量协作开发。这种转变带来了巨大的管理和技术挑战，导致了一系列问题。

(2) 软件危机的表现和根源

●表现：用户对交付软件不满意、软件质量不可靠、项目超预算或延误。

●根源：**软件自身**的高复杂度。**技术问题**：沿用落后的开发方法，难以应对新的技术挑战。**组织管理问题**：需求不清晰、沟通不畅、项目进度失控。**市场需求问题**：市场变化快，要求软件快速迭代，开发周期被压缩，缺乏标准和规范。

●计算机辅助软件工程 (CASE)。**核心定义**：一套辅助软件开发全生命周期的工具和方法的集合，旨在提高开发效率和软件质量，实现开发的自动化。**CASE 工具分类**：**分析与设计工具**：如可视化建模工具（画 UML 图的工具）。**程序设计工具**：如编译器、调试器。测试**与质保工具**：如自动化测试、性能分析工具。**项目管理工具**：如代码托管（Git）、团队协作工具（Jira）。**软件运维工具**：如自动化部署、系统监控工具。**集成开发环境** (IDE)：将以上多种工具集于一身的综合性平台，如 Visual Studio, Eclipse。

三、 软件过程

1. 软件过程基础（概念和模型）

●软件过程概念：为构建高质量软件而实施的一系列活动、任务和步骤的集合。它规定了在软件生命周期中要做什么、由谁做、需要什么资源、如何保证质量等。

●软件过程模型概念：一种开发策略，为软件开发过程提供了一套范式（即模板）。它为如何安排需求、设计、实现、测试等基本活动提供了指导。

2. 经典的软件过程模型

(1) 瀑布模型

●核心思想：基于确定性假设、严格线性的、阶段化的开发，每个阶段完全结束后才能进入下一阶段。

●特点：阶段严格顺序、文档驱动，阶段清晰，像瀑布一样不可逆。

●适用：需求非常明确且稳定的项目。

(2) V 模型（瀑布变体）

●编码为顶点，测试与分析和设计的对应相关，每个开发阶段都有一个对应的测试阶段（如单元测试对应详细设计）。特点：文档驱动，阶段清晰，像瀑布一样不可逆。

●适用：测试贯穿始终，验证与确认并重。

●对可靠性和安全性要求极高的项目。

(3) 原型模型

●核心思想：先快速构建一个可运行的原型给用户看，通过不断收集反馈来明确需求，最终再开发正式系统。2 种策略：废弃策略(原型仅用于需求探索和验证，最终系统从头开发，原型不被保留)、追加策略(原型经过不断完善和扩展，逐步演化为最终系统)

●特点：“先做再改”，用户参与度高。

●适用：需求模糊、需要反复验证的探索性项目。

(4) 增量模型(瀑布+原型迭代)

●核心思想：将软件按功能模块化，每次只开发和交付一个或多个模块（增量），像搭积木一样逐步构建整个系统。

●特点：分批次交付，资金回笼快，风险分散。

●适用：需求较清晰、可分批次实现的项目。

(5) 迭代模型(瀑布的缩小与循环)

●核心思想：将整个开发过程划分为多次短周期的“小瀑布”，每次迭代都完整地经历需求、设计、编码、测试，并交付一个可用的、功能更丰富的版本。

●特点：多次循环，快速反馈，逐步求精。

●适用：需求可能频繁变更、需要快速适应的项目。

(6) 螺旋模型(瀑布+演化+风险分析)

●核心思想：依次进行制定计划、风险分析、实施开发、客户评估。

●特点:风险驱动，每一圈都包含计划、风险分析、开发与评估。

●适用：大型、复杂、高风险的项目。

(7) 演化模型

●核心思想：先开发一个简化的核心系统，然后根据用户反馈和新需求，使其逐步演化和完善。

●特点：“先生长，再变壮”，强调系统对变化的适应能力。

●适用:无法在初期完整定义需求，需要系统持续演进的项目。

(8) 统一过程模型

●核心思想：一个重量级的、综合性的过程框架，其特点是用例驱动，以架构为中心、迭代和增量。

●特点：综合全面，纪律性强，文档和角色定义严格。

●适用：大型、复杂的软件项目和团队。

3. 软件过程模型选择示例

(1) 示例 1(需求明确)：开发一个需求非常明确、几乎不会变更的“企业内部信息管理系统”，可以选择瀑布模型，按部就班地进行。

●示例 2(需求模糊)：开发一个全新的、连客户自己都想不清楚具体功能的“创新社交 App”，可以选择原型模型，先做出一个可交互的样子来收集反馈。

(2) 适用性

●核心思想：没有“最好”的模型，只有“最合适”的模型。选择哪种模型，取决于项目的具体情况。

●主要考量因素：需求明确度、需求变化的可能性、项目风险、资金情况、团队能力等。

(3) 特定类型的软件开发适用的过程模型

1.基础软件开发时的过程模型:(开发技术复杂度高、需求明确、可靠性要求高)瀑布、V、螺旋模型(高风险模块)2.工业软件开发时的过程模型:(需求可能不完全明确，用户参与度高、需持续优化)原型、增量、迭代模型、RUP 模型（规模复杂，实现系统化管理与持续演进）。

(4) 软件过程模型的选择建议

●如果需求在前期就已非常明确、稳定不变 → 瀑布模型。

●如果用户对需求描述不清、需要通过原型来探索和验证 → 原型模型、迭代模型、螺旋模型。

●如果项目风险高、不确定性大，难以在早期制定详细计划 → 螺旋模型、增量模型、迭代模型。

●需求在开发中频繁变更 → 增量模型、迭代模型、演化模型。

●项目资金无法一次性到位，需要分批看到成果 → 增量模型。

●如果团队经验或技能不足，难以驾驭复杂流程 → 避免使用管理难度大的迭代式模型 (如 RUP、迭代模型)。

●如果需要综合多种模型的优点来开发大型复杂项目 → 统一过程模型 (RUP)，并为每次迭代设定清晰的验收标准。

●项目协调可以清晰地拆分成多个独立模块 → 可以在整体上采用增量模型，对每个独立的模块内部再采用瀑布模型。

四、软件开发方法（简答/分析设计）

1.面向对象开发方法

(1)面向对象开发方法的概念和思想

●对象(Object):系统基本单位。代表一个客观事物，包含属性(静态特征)和操作(动态行为)。

●类(Class):对象的模板/蓝图。是具有相同属性和操作的对象的集合。

●封装(Encapsulation):隐藏内部细节。将对象的属性和操作捆绑成一个独立的单元。

●继承(Inheritance):实现代码复用。子类自动拥有父类的属性和操作。

●多态(Polymorphism):同一接口，多种实现。父类的操作在不同子类中有不同的行为表现。

●消息(Message):对象间的通信方式或操作请求。

●核心：从现实世界客观存在的对象出发，用人类的自然思维来构建软件系统。

●公式：面向对象=对象+类+继承+通信(Coad&Yourdon)

(2)用例图

●参与者(Actor)：与应用程序或系统进行交互的用户、组织或外部系统。用一个小人表示。

●用例(Use Case)：用例就是外部可见的系统功能，对系统提供的服务进行描述。用椭圆表示。

●系统边界：系统边界是指系统与系统之间的界限。用方形容器+系统名称表示。

●参与者之间的关系：用例图中的关系有包含、扩展和泛化 3 种。

关系	符号	解释
包含	<<include>>	一个用例的执行必须包含另一个用例的功能。箭头从基础用例指向被包含用例。（登录指向下订单）
扩展	<<extend>>	一个用例的功能可以扩展(增加)另一个用例。箭头从扩展用例指向 基础用例。(用优惠券指向支付)
泛化	————▷	“是一种”的关系，类似继承。箭头从子类(研究生)指向父类(学生)。

(3)类图

●访问修饰符

+ public, - private, # protected, ~ package

●类之间的关系

种类	关系	UML 符号	记忆
泛化	最强	————▷ (实线空心三角)	继承关系，子类指向父类
实现	强	- - - -▷ (虚线空心三角)	类实现接口/抽象类，实现类指向接口
组合	强	◆———> (实心菱形指向整体)	不可分割，生命周期相同（车和引擎）
聚合	中	◇———> (空心菱形指向整体)	可分割，生命周期不同（班级和学生）
关联	中	———> (实箭头指向被拥有者)	普通成员变量关系(学生和课程)
依赖	最弱	<- - - (虚线箭头指向被使用者)	临时使用，如方法参数（人开车）

●多重性：标在关联/聚合/组合关系的连线两端，表示一个类的对象可以与另一个类的多少个对象相关联。

1 ：只有一个　　0..1 ：0 个或 1 个

* 或 0..* ：0 个或多个　　1..* ：1 个或多个

m..n ：m 到 n 个

(4)状态图

●基本组件

组件	符号	解释说明
状态	方框	对象在其生命周期中的一种状况或条件。框内写状态名，如“待支付”。
初始状态	●	起点。一个图中只能有一个，表示对象刚被创建时的状态。
最终状态	◉	终点。一个图中可以有多个，表示对象生命周期的结束。
转移	——>	状态之间切换，由一个事件触发。

●转移线上标注的文字格式为：事件 [守卫条件] / 动作。示例：确认付款 [金额>0] / 更新库存()。条件和动作可以不写

(5)时序图

●基本组件

组件	符号	解释说明
对象	[类名]	方框里写对象名和类名（对象:类名），通常对象名可省略如[Student]
生命线	¦	对象下方的垂直虚线，表示对象存在的时间。
激活条		生命线上的细长矩形，表示该对象正在执行操作或“忙碌”的时间段。

●消息类型

类型	UML 符号	解释
同步	————▶ (实线实心箭头)	A 调用 B，必须等 B 处理完返回后 A 才能继续
异步	————> (实线空心箭头)	A 调用 B，不等待 B 返回，A 立即继续自己工作
返回	<- - - - - (虚线空心箭头)	从同步消息的调用方返回。
自调用	自己指自己	对象调用自己的方法。

●交互片段

alt 多选一。类似 if-else，框内用虚线分隔不同条件的流程。

opt 可选。类似 if，框内的交互可能发生，也可能不发生。**loop** 循环。框内的交互会重复执行多次。

(6)协作图

●基本组件

组件	符号	解释说明
对象	[:类名]	和时序图一样，一个对象实例。
链接	————	连接两个对象的实线，表示它们之间存在某种关系
消息	1:do () --->	附着在链接上，表示一次通信。这是与时序图最大的不同。

●消息的语法

时间的先后顺序是通过消息编号来体现的，而不是通过垂直位置。**编号格式**：[序号]：[消息名()]**顺序流**：使用数字表示。1, 2, 3... 代表了交互的主流程。**嵌套流**：使用小数点表示。如果消息 2 引发了一个子调用，那么这个子调用的序号就是 2.1。下一个子调用就是 2.2 **例如**：2: processOrder() 内部调用了 2.1: checkStock() 和 2.2: deductPayment()。**方向**：在消息旁边或者下面画一个小箭头 -> 来指明消息的传递方向。

(7)活动图（和流程图很像）

组件	符号	解释说明
动作/活动		流程中的一个步骤。一个圆角矩形，里面写着具体要做的事，比如“用户登录”。
起始	●	流程的开始。实心圆。
终止	◉	流程的结束。同心圆。
控制流	——>	执行顺序。连接各个动作的箭头。
判定	◇	做决策，类似 if-else，每个出口线上要标注条件，如[Yes]。
合并	◇	汇合路径。个路径重新合并成一条。多个入口，一个出口。
分叉	▌(粗黑线)	并行处理。将一个流程拆分成多个可以同时进行的活动。
汇合	▌(粗黑线)	等待同步。将多个并行活动重新合并成一个流程。
泳道	方框框住	划分职责。将活动按负责的角色或部门进行分组。

(8)组件图

组件	符号	解释说明
组件		物理构建块。一个带图标的方框。
组件模块		物理构建块。一个带图标的方框。
提供接口	——o	该组件对外提供的服务或功能。
需求接口	——(	该组件需要依赖的外部服务或功能。
依赖关系	<--	一个组件（箭头起点）的编译或运行需要另一个组件（箭头终点）。
组装连接	-(O-	将一个组件的“需求接口”和另一个组件的“提供接口”连接起来。

(9)部署图

组件	符号	解释说明
节点	(3D 立方体)	核心元素。一个物理硬件设备或软件执行环境。
构件	(文档图标)	软件的物理产物，是实际被部署的东西。它被画在节点内部。
通信路径	————	连接两个节点的实线，表示它们之间的网络连接。可以在线上写明协议。
依赖关系	<-- -->	一个构件对另一个构件的依赖关系。

(10)面向对象开发方法的过程及策略

●面向对象分析 (OOA)。核心任务：理解问题域，回答系统“做什么 (What)”。关注点：业务逻辑、用户需求。主要产出：用例图、分析类图 (如实体类、边界类、控制类)。

●面向对象设计 (OOD)核心任务：构建解决方案 (Solution Domain)，回答系统“怎么做 (How)”。关注点：在 OOA 模型基础上，补充实现细节（如数据库设计、接口设计、UI 交互）。主要产出：设计类图、时序图等详细的交互图。

●核心关系与流程。关系: OOA 与 OOD 之间是无缝连接、平滑过渡的关系。OOD 是对 OOA 模型的精化和扩展，而非推倒重来。过程: 用例驱动，以类和对象为中心。

●UML 建模流程:需求分析(OOA) → 用例图。系统分析 (OOA) → 分析类图。系统设计 (OOD) → 设计类图 + 时序图

2.敏捷开发方法

(1)敏捷开发方法的理念

●核心理念：尊重人、强调沟通、拥抱变化、注重灵活性。

●过程思想：快速迭代、快速反馈、快速修改、增量交付。

●执行策略：整为零、小步快跑，并追求简洁（尽最大可能减少不必要的工作）。

(2)思想和原则

●四大价值观 (思想)：个体与互动胜于过程与工具；可运行的软件胜于面面俱到的文档； 客户合作胜于合同谈判；响应变化胜于遵循计划。

●最高目标是通过尽早和持续交付有价值的软件来满足客户。

●欢迎并善于利用需求变更。

●业务人员与开发者必须每日在一起工作。

●可运行的软件是衡量进度的主要指标。

●最佳的架构、需求和设计出自于自组织的团队。

●团队要定期反省如何能更有效，并相应地调整行为。

(3)XP（极限编程）开发方法

●核心价值观:沟通、简单、反馈、尊重、勇气。

●需求形式:使用轻量级的用户故事来描述用户需求。

●测试驱动开发 (TDD):先写失败的测试用例，再编写代码使其通过，最后重构。

●结对编程: (最具特色的实践) 两个人用一台电脑结对工作，一人编码，一人即时复审。

●持续集成 (CI): 频繁地（一天多次）将所有开发者代码集成到主干，并自动构建测试。

●简单设计:选当前能够解决问题的最简单设计，避免过度工程。

●重构 (Refactoring): 在不改变软件外部行为的前提下，持续地改进其内部结构。

●开发节奏: 通过计划、设计、编码、测试的短迭代周期（通常 1-2 周）进行开发。

(4)Scrum 开发方法

●核心思想：将开发过程划分为多个固定的、短周期的迭代 (Sprint)，每次迭代都产出一个可交付的产品增量。

●3 个核心角色**产品负责人**：负责定义产品功能、管理和优化产品待办列表 (Product Backlog)，并对其进行优先级排序。**Scrum 主持人**：确保团队遵循 Scrum 规则，是团队的服务型领导，负责主持会议、排除开发过程中的任何障碍。**开发团队**：执行开发的自组织团队（通常 5-9 人），负责在每个冲刺中交付可工作的软件。

●3 个核心工件**产品待办列表**：一个动态的、按优先级排序的产品总需求清单。在被开发前可随时变更。**冲刺待办列表**：在一次冲刺中，开发团队承诺要完成的任务清单。**产品增量**：每次冲刺结束时产出的可工作、可交付的软件新功能。

●5 个核心事件**冲刺**：核心事件，一个 2-4 周的固定时间盒，用于创建一个产品增量。一旦启动，冲刺内部的需求不允许变更。**冲刺计划会议**：在冲刺开始时召开，从产品待办列表中选择本次冲刺要完成的任务，并制定冲刺计划。**每日站会**：每天固定的 15 分钟短会，团队成员同步进度，回答三个问题：昨天做了什么？今天要做什么？遇到了什么困难？**冲刺评审会议**：在冲刺结束时，向产品负责人及其他相关者展示本次冲刺的成果（可工作的软件）。**冲刺回顾会议**：在评审会议之后，团队内部反思和总结本次冲刺的工作流程，讨论如何改进。

(5)TDD（测试驱动）开发方法

●核心思想：测试先行。即在编写任何功能代码之前，先为该功能编写一个自动化的测试用例，然后才编写足以让该测试通过的功能代码。

●开发节奏 (红-绿-重构循环): **红**：写一个失败的测试。因为功能还没实现，所以这个新测试理应运行失败（红色）。**绿**：写最少的代码让测试通过。目标是尽快让测试条变绿，这个阶段不追求代码的完美。**重构**：在所有测试都通过的前提下，优化和清理代码（如消除重复，提高可读性），并确保优化后所有测试依然能通过。

●主要优点：驱动出简洁可用、高质量的代码。代码拥有很高的灵活性和健壮性，能快速响应变化。提供了一个可靠的安全网，让开发者有勇气随时重构和优化。

五、需求工程（分析设计）

1. 需求工程概述

(1) 需求工程概念

●什么是需求工程：它是一套系统化、规范化的方法，用来科学地获取、分析、定义、验证和管理软件需求的全过程。

●什么是需求：简单来说，就是用户期望系统“做什么 (What)”，而不是“怎么做 (How)”。它是对系统在功能、性能、约束等方面的期望的描述，是后续所有开发和测试的基石。

●需求的分类 (重点)：**功能需求**：系统必须提供的服务和功能。(例如：用户登录、商品搜索、生成报表)。**非功能需求**：系统的质量属性和约束。(例如：搜索响应时间必须少于 1 秒)。**设计约束**：在设计时必须遵守的限制。(例如：必须使用 C#语言开发)

(2) 需求工程过程模型

●核心活动 (线性过程)：需求工程主要包含以下五个连续的活动：**需求获取**：通过访谈、问卷、观察等方式从用户、文档和环境中挖掘需求。**需求建模与分析**：整理并分析获取到的信息，消除矛盾，建立模型（如用例图）。**需求规约**：将分析好的需求编写成正式、清晰、无歧义的文档（即软件需求规格说明书）。**需求确认与验证**：与用户一起评审需求文档，确保其准确反映了用户的真实意图。**需求管理**：在整个软件生命周期中，跟踪和控制需求的变化。

●迭代与增量模型：上述五个活动不一定只执行一次，也可以是：**瀑布模型**：严格线性的软件开发模型，它要求各个开发阶段（如需求、设计、编码、测试）像瀑布一样依次顺序执行，一个阶段完全结束后才能进入下一个阶段。**迭代模型**：对整个系统的需求进行多轮细化，每一轮都让需求变得更清晰、更完整。**增量模型**：每次只对系统的一部分完成完整的五个活动，分批次地明确整个系统的需求。**敏捷方法**：通常在每个冲刺中，快速地对一小部分需求执行上述五个活动。

2. 需求获取

核心定义：也叫“需求抽取”，是从相关的人员、资料和环境中，系统地获取用户对软件系统期望和约束信息的过程。

(1) 需求获取任务

●明确目标与范围：定义项目要达成的目标，清晰地界定系统的功能边界（即哪些做，哪些不做）。

●识别涉众：找出所有与项目相关的人员（如用户、客户、经理、开发人员），对他们进行分类，明确向谁获取何种需求。

●思考核心问题：在获取时，要带着问题去挖掘，如为谁(Who)、做什么(What)、为何(Why) 做、在何种场景(Where)下、何时(When)使用、如何(How)交互、以及质量多好(How well)。

●收集与计划：到用户现场，通过各种方法实际收集信息，并提前规划好需求获取活动的进度和安排。

●确定非功能需求:专门花时间来确定那些关于性能、安全性、易用性等决定软件“好不好用”的质量要求。

(2) 需求获取的典型方法

●访谈法：与涉众面对面或电话交流，通过提问获取深入、具体的信息，是最直接也是最常用的方法。

●原型法：快速创建一个可交互的系统模型（原型），让用户实际操作并提出反馈，对澄清界面和交互流程特别有效。

●研讨会法：组织一个由关键涉众参加的集中会议，通过头脑风暴和讨论，快速解决分歧，达成共识。

●问卷调查法：向大量的、分散的用户发放问卷，收集量化数据或普遍性意见。

●观察法：在用户的实际工作环境中观察他们的操作，可以发现那些他们自己都习以为常、难以言表的“隐性需求”。

●文档分析：阅读与系统相关的现有文档（如行业法规、公司制度、旧系统手册），从中获取需求和约束。

3. 需求建模与分析

(1) 需求建模与分析的典型方法

a. 面向主体的建模 (以 i* 框架为例，了解即可)

●核心思想：关注“为什么 (Why)”。通过对不同参与者的目标和互相依赖关系进行建模，来理解需求的深层动机。

●参与者：指的是系统中的人或组织，分为两种：**主体** (Agent)：具体的实体，如“张三”、“饿了么公司”。**角色** (Role)：抽象的身份，如“顾客”、“骑手”。

●意图元素：参与者内心想要的东西。**目标**：想要达成的最终状态，如“外卖被送到”。**任务**：为实现目标而要执行的动作，如“支付订单”。**资源**：执行任务需要的东西，如“信用卡”。**质量**：对目标的属性要求，如“快速送达”。

●核心关系 (Relationship)：**依赖关系**：最重要的关系。表示一个参与者 A (依赖者) 需要其自己或其自己某个目标，需要依赖另一个参与者 B (被依赖者)，来提供某项服务 (依赖项)。

b. 面向场景的需求建模与分析方法

●**场景的核心概念**：什么是场景：场景就是一个“小故事”，它描述了某个角色为了达成一个特定目标，与系统进行交互的具体步骤。**场景的基本要素**：角色：与系统交互的人或物。目标：本次交互想要达成的目的。前置条件：场景发生前必须满足的条件。后置条件：场景结束后系统所处的状态。资源：完成场景所需的人、财、物等。场所：场景发生的环境。

●编写场景描述的核心规则：**语言要清晰、直接**：使用一般现在时和主动语态（如：“用户点击按钮”）。使用简单的主谓宾句式。避免使用“应该”“可能”等模糊词语。**结构要专注、有条理**：一个句子只描述一个动作，避免长句。**为每个步骤编号**，使其更有条理。只描述从外部能看到的交互，不要涉及内部实现细节。

●基于 UML 的需求建模（三视图法）。这是用 UML 进行需求分析和设计的核心思路，即从三个不同的视角来观察和构建系统模型：**用例视图**（系统“做什么”）关注点：系统的功能需求。核心模型：**用例图**。通过它来识别系统的所有功能，使用者以及它们之间的关系。**行为视图**（功能“如何动态实现”）关注点：系统中对象之间在运行过程中是如何协作的。核心模型：**时序图**、**状态图**、**活动图**。**结构视图**（系统“由什么静态构成”）关注点：系统的静态组成结构，即有哪些类、模块以及它们之间的关系。核心模型：**类图**、**组件图**。

(2) 需求协商

●核心目标：解决不同涉众（用户、老板、开发者等）之间对需求的矛盾和冲突，最终达成所有人都能接受的一致意见。

●协商的主要过程：**识别冲突**：找出并记录不同人对同一个需求的矛盾点。**分析冲突**：搞清楚冲突的根本原因是什么。**解决冲突**：组织相关人员，采用合适的方法共同解决问题。**记录方案**：把解决方案文档化，避免以后重复讨论。

●常见的冲突原因：**数据冲突**：因信息不足或对信息理解不一致导致。（如：对“快速响应”的时间定义不同）**利益冲突**：因各自的目标和利益不同导致。（如：老板要省钱，用户要功能多）**价值冲突**：因价值观或理念不同导致。（如：有人偏爱开源技术，有人偏爱闭源）

●主要的协商方法：**协商一致法**：通过沟通、争取让各方认同一个观点，或者各退一步达成折中方案。**决定一致法**：当无法达成一致时，通过投票、领导决策或由层级更高的人拍板来做决定。**Win-Win 模型**：不争输赢，而是去理解各方的真实利益诉求，共同探讨能让所有人都满意的解决方案。

(3) 需求优先级排序

●核心目标：在资源（时间、人力、金钱）有限的情况下，决定哪些需求应该最先被开发。

●基本步骤：确定排序标准 -> 列出所有需求 -> 进行排序 -> 定期回顾和更新。

核心方法：Kano 模型

●核心思想：通过分析某个功能“有”或“没有”对用户满意度的影响，来决定其优先级。

●5种需求类型 (重点)：**基本型**“必须有”。用户觉得理所当然。没有它用户会极度不满，但有了也不会更满意。（如：正常登录）。**期望型**“越有越好”。满足得越多，用户就越满意。（如：性能越快越好）。**兴奋型**“惊喜功能”。没有它用户无所谓，但一旦有了，用户满意度会剧增。（如：年度听歌报告）。**无差异型**“有没有都行”。满足与否对用户满意度没影响，应降低优先级。**反向型**“最好没有”。提供了反而会让用户不满。（如：过多的广告弹窗）。

●排序思路：优先满足基本型 -> 尽可能多地实现期望型 -> 适当添加兴奋型 -> 尽量不做无差异型和反向型。

4. 需求规约

(1) 需求规约的结构和内容

●核心产物：《软件需求规格说明书》，这是需求阶段最重要的正式文档。

●主要结构 (参照国标模板)：一份合格的需求文档通常包含三大块：**引言**：说明文档的目的、范围等基本信息。**总体描述**：介绍产品的总体功能、用户特点、设计约束和假设等。**具体需求**（核心部分）：外部接口需求：对用户界面、软硬件接口、通信接口的要求。功能需求：最核心的内容，详细描述系统在各种模式下“做什么”。**性能需求**：量化指标，如响应时间、并发用户数等。**设计约束**：必须遵守的限制，如必须使用哪种数据库、遵循哪个标准。**软件系统属性**：其他非功能需求，如可靠性、可用性、安全性等。

●好的规约应具备的特性：**正确性**：准确地反映了用户的真实需求。**无歧义**：无论谁来读，理解都应是一样的。**完备性**：包含了所有必要的需求，没有遗漏。**一致性**：需求内部、需求之间没有相互矛盾的地方。**可验证性**：需求是可以被测试的，能明确判断是否已实现。**可追踪性**：可以追溯每个需求的来源，并追踪其后续的设计、编码和测试。**可修改性**：结构清晰，易于修改和维护。

(2) 需求规约的描述方法

●非形式化方法（最常用）。**方法**：使用自然语言（如中文、英文）来描述需求。**优点**：通俗易懂，方便与非技术人员沟通。**缺点**：容易产生歧义和不一致。

●半形式化方法（次之）。**方法**：使用工具来描述，如图形化的 UML 图（用例图、活动图等）。**优点**：比自然语言更精确，也相对易懂。**缺点**：需要学习建模语言。

●形式化方法（非常严格）。**方法**：使用严格的数学和逻辑语言来描述需求。**优点**：最精确、无歧义，可通过数学方法验证。**缺点**：非常复杂，难于学习和理解，成本极高。**应用场景**：只用于对安全性和可靠性要求极高的系统（如航空航天）。

5. 需求确认和验证

(1) 需求确认

●核心目标：确保在构建正确的系统，即需求文档准确地反映了用户的真实意图和需要。

●主要原则：让正确的涉众（用户、客户等）参与进来。从不同角度来确认需求，避免单一视角。积极开发原型，让用户看到模型，辅助他们确认需求。反复确认，在整个需求阶段持续进行。

●主要方法：**审查法**：一种正式的会议，组织所有相关人员对需求文档进行逐条、详尽的检查和讨论。**走查法**：一种非正式的讨论，通常在开发团队内部或与少数用户代表进行，旨在初步统一对需求的理解。**原型法**：通过构建可交互的系统原型，让用户直观感受和评判，这是最有效的确认方法之一。

(2) 需求验证

●核心目标：确保构造的系统（或需求文档本身）是正确的，即需求文档自身符合质量标准（如清晰、完整、无矛盾）。

●主要检查内容与**正确性**：需求描述是否正确无误。**完整性**：是否遗漏了任何必要的需求。**一致性**：需求之间是否存在相互矛盾。**无歧义**：需求描述是否清晰、不会产生多种解释。**可测试性**：每条需求是否可以被验证和测试。

●主要方法：**需求测试**：为每条需求编写测试用例，如果某个需求很难或无法写出测试用例，说明该需求可能定义得不清晰。**一致性分析**：使用工具或人工检查，确保整个需求文档中术语使用统一，没有逻辑上的冲突。**形式化验证**：采用数学和逻辑方法对需求进行严格的分析和证明，主要用于安全性、可靠性要求极高的系统。

六、软件设计（分析设计）

1. 软件设计概念

(1) 软件设计概念和思想、目标、过程和原则

●核心思想：软件设计是一个充满决策与权衡的过程，要平衡功能、性能、成本等多个维度，最终形成一套解决方案蓝图。

●设计目标：满足用户明确提出的功能需求，满足用户关心的质量需求（如性能、可靠性），满足开发者关心的系统属性（如可维护性、可扩展性）。

●设计过程：**概要设计**（体系结构设计）：搭建系统的高层“骨架”，定义主要的模块、组件以及它们之间的关系。**详细设计**：深入每个模块，定义其内部实现机制、数据结构和算法。

●核心设计原则：**模块化**：将复杂的系统拆分成独立可管理可重用的小模块。**高内聚、低耦合**：模块内部的各个元素功能联系紧密（高内聚），模块与模块之间的依赖关系要尽可能弱（低耦合）。**单一职责**：一个模块（或类、方法）只负责一件明确定义的事情，**开闭原则**：对扩展是开放的（可以增加新功能），但对修改是关闭的（不修改已有代码）。

(2) 软件设计质量要求

●核心思想：软件的质量（即非功能需求）不是一个模糊的概念，它必须是清晰、可量化、可检验的。

●质量属性分类：**运行时属性**（用户可感知）：如性能（快不快）、可用性（稳不稳定）、安全性。**开发时属性**（开发者关注）：如可修改性（好不好改）、可测试性（好不好测）。

●基于场景的质量描述法（重点）：这是一种精确描述非功能需求的实用方法。一个完整的场景包含 6 个要素：**刺激**（发生了什么事）、**刺激源**（谁/什么触发的）、**环境**（事发时系统状态）、**制品**（系统哪个部分受影响）、**响应**（系统该如何反应）、**响应度量**（用什么具体指标衡量响应的好坏）。

2. 软件体系结构及模式

(1) 软件体系结构的概念

●核心定义：系统的高层骨架，由软件元素（如模块、组件）、它们之间的关系（如依赖）以及两者的属性共同组成。

●核心目标：好的体系结构旨在满足系统的质量属性（如高性能、高可维护性、易扩展性）并能作为后续开发的指导蓝图。

(2) 软件体系结构模式的概念

●核心定义：针对一类常见的设计问题，被实践证明行之有效的、可复用的解决方案。

●核心作用：每个模式都是为了解决特定的质量属性问题。例如，分层模式就是为了提高系统的“可维护性”和“可扩展性”。

(3) 软件体系结构模式的描述方法

●三大基本要素： **上下文** (Context)：描述问题发生的环境和背景。**问题** (Problem)：描述在该环境下需要解决的目标和面临的挑战。**解决方案** (Solution)：描述一个经过验证的设计方案，包括其元素、关系和拓拓扑结构。

(4) 典型软件体系结构模式及其特点（重点）

●分层模式(Layered)。**核心思想**：将系统划分为多个水平层次（如常见的表现层、业务逻辑层、数据访问层），规定上层只能调用其直接下层的服务。**特点**：职责清晰，关注点分离，便于维护和团队分工。

●模型-视图-控制器(MVC)模式。**核心思想**：将应用分为三部分：模型 (Model) 负责数据和业务逻辑，视图 (View) 负责数据显示，控制器 (Controller) 负责接收用户输入并调用模型和视图。**特点**：实现了数据逻辑和界面显示的解耦，支持同一数据有多种不同的展示方式。

●客户端/服务器模式(Client/Server)。**核心思想**：由客户端发起请求，服务器端负责处理请求、管理数据和提供服务。**特点**：集中管理资源，职责分明，是绝大多数网络应用的基础。

●管道-过滤器模式(Pipe-Filter)。**核心思想**：数据像水一样流过一系列独立的处理单元（过滤器），每个过滤器完成一步特定的数据转换任务，并通过管道连接。**特点**：处理步骤清晰，过滤器可重用、可替换，适合数据流处理任务（如编译器、数据分析）。

●发布-订阅模式(Publish-Subscribe)。**核心思想**：发布者发布消息到事件中心，订阅者只接收自己感兴趣的消息，发布者和订阅者之间互不知道对方的存在，完全解耦。**特点**：极度松耦合，扩展性强，非常适合于异步、事件驱动的系统。

3. 软件体系结构设计

(1) 软件体系结构设计任务

●系统分解：核心任务是将整个系统分解成更小可管理的模块或组件。分解时需要考虑： 如何将质量属性（如安全性、性能）合理地分配给不同模块。必须遵循设计约束（如模块间的数据隔离）。可以基于成熟的体系结构模式来进行分解。

●满足关键需求与权衡：体系结构设计必须由最重要的质量需求（如高可用性、低延迟）来驱动，并对这些可能相互冲突的需求进行权衡与决策。

(2) 过程和归档

●设计过程：一个典型的体系结构设计流程包含以下步骤：**理解需求**：从业务角度出发，理解项目的商业目标，并从需求中提炼出对体系结构有决定性影响的关键质量需求。**创建/选择方案**：基于关键需求，设计一个新的体系结构方案，或选择一个合适的、已有的体系结构模式。**分析与评审**：对设计方案进行评估和审查，确保它能满足关键需求。**文档化与沟通**：将最终确定的体系结构设计方案清晰地记录归档（如通过多种视图），并与所有项目涉众进行沟通。**指导实现与演化**：已确定的体系结构将作为后续开发、测试和系统未来演化的指导蓝图。

●设计归档（文档）：体系结构文档至关重要，其主要作用是：作为团队内外部沟通的工具。作为新成员培训和学习的资料。作为后续系统分析、构建和维护的基础。

4. 软件详细设计

(1) 软件详细设计概念

●核心目标：将概要设计的抽象蓝图，转化为具体、可实现的软件内部规格说明，是编码前的最后设计阶段。

●主要任务：**用例设计**：明确一个具体功能是如何由对象间的交互来实现的。**类设计**：定义类的属性方法以及类之间的关系。**数据设计**：确定需要持久化存储的数据结构和访问方式

(2) 用例设计

●核心思想：关注一个具体用例的实现过程，详细描述系统中不同对象是如何协作来完成该用例功能的。

●主要产出：详细的顺序图，它清晰地展示了为实现一个用例，消息在不同对象之间的传递顺序。

(3) 初始类图构建

(4) 类设计

(5) 通用职责分配模式(GRASP)

●核心问题：类设计旨在回答"一个职责（如创建一个订单、计算总价）应该分配给哪个对象来完成？" GRASP 就是一套解决这个问题的指导原则。

●核心 GRASP 模式：**信息专家**：谁拥有信息，谁就负责处理这些信息。这是最基本、最核心的职责分配原则。**创建者**：如果对象 A 聚合或包含对象 B，并掌握创建 B 所需的初始化数据，那么就由 A 来负责创建 B。**控制器**：负责接收和协调系统事件（如用户点击按钮），将工作分配给后续的业务对象，起"总指挥"作用，通常用于连接 UI 和后端逻辑。**高内聚、低耦合**：(最重要的设计思想) 让类（或模块）的职责高度集中和相关（高内聚），同时让类与类之间的依赖尽可能地弱（低耦合）。

(6) 设计原则(必考)

●S - 单一职责原则：一个类只应该有一个引起它变化的原因（即只负责一件事）。

●O - 开闭原则：对扩展开放，对修改关闭（即增加新功能通过扩展实现，而不是修改旧代码）。

●L - 里氏替换原则：子类对象必须可以无缝地替换其父类对象，而不产生任何错误。

●I - 接口隔离原则：客户端不应被迫依赖它用不到的接口（即接口要设计得小而专）。

●D - 依赖倒置原则：高层模块不应依赖底层模块，两者都应依赖于抽象（接口）。细节应依赖于抽象，而不是反过来。

(7) 类的精化

●核心思想：在已有类图的基础上，进一步审查和完善类的关系、属性和方法，使其设计更合理、更具体，为编码做好准备。

●关系精化：审查类之间的关系（继承、关联等）。总原则是：优先使用弱关系（如依赖、关联），慎用强关系（如继承），以保持系统的低耦合。

●属性精化：明确每个属性的名称、类型、可见性（公有+ / 私有-）和默认值等细节。

●方法精化： 命名：使用清晰的"动词+名词"来命名，准确反映其功能。参数与返回值：明确定义每个方法的输入参数和输出返回值的类型。实现逻辑：对于复杂的方法，可以使用活动图来描述其内部的实现步骤和逻辑。

(8) 数据设计概念和过程

●核心思想：确定系统中哪些数据需要被持久化存储（即关掉程序后数据还在），选择合适的存储方式，并设计对这些数据的增删改查（CRUD）操作。

●设计过程：**确定持久化数据**：识别出系统中的实体类，这些类通常是需要被永久保存的。**选择存储方式**：根据需求选择存储介质，如关系型数据库（MySQL）、NoSQL 数据库（Redis）、文件系统等。**设计数据操作**：定义对这些持久化数据的增、删、改、查等具体操作方法。

●对象-关系数据库映射（O-R Mapping）规则（重点）：**基本映射**：一个实体类 通常对应一张数据库表；类的一个属性对应表的一个字段（列）。**一对多关系**：在"多"的那一方的表中，增加一个外键，用来存储"一"那方表的主键。**多对多关系**：需要额外创建一张中间表（关联表），该表至少包含两个外键，分别指向原来两张表的主键，以此来建立关联。

5. 设计模式（补充）

(1) 工厂方法模式

一句话解释：定义一个创建对象的接口，但把具体创建哪种对象的工作，交给不同的子类去做。生活例子：汽车工厂。总厂（抽象工厂）只规定"必须能生产汽车"，至于 A 分厂（具体工厂）是生产"轿车"，还是 B 分厂（具体工厂）生产"SUV"，由分厂自己决定。**优点**：扩展性好。想增加一种新车（新产品），只需要建一个新的分厂（新的具体工厂类）就行，不用改总厂的代码。**缺点**：每增加一种产品，就要多一个对应的工厂类，导致类的数量变多。

(2) 抽象工厂模式

一句话解释: 用来创建一整套相关的对象，而无需指定它们具体的类。生活例子：换皮肤。软件的"夏日清凉"主题（一个具体工厂）会提供一整套蓝色系的按钮、窗口和滚动条。而"冬日暖阳"主题（另一个具体工厂）会提供一套橙色系的。你选择一个主题，所有 UI 组件都会配套换掉。**优点**：能确保你使用的产品是"一套"的，风格统一，兼容性好。**缺点**：如果想给这套产品增加新的种类（比如想给所有皮肤都增加一个新的"音效"组件），会非常困难，需要修改所有工厂。

(3) 单例模式

一句话解释：保证一个类在整个系统中只有一个实例，并提供一个全局唯一的访问点。生活例子：班里的班长。一个班级在同一时间只能有一个班长，任何人提到"我们班班长"，指的都是同一个人。**优点**：节省系统资源，因为实例只有一个；方便全局统一管理。**缺点**：扩展性差；对测试不友好；容易造成职责不清。

(4) 组合模式

一句话解释：把单个对象（叶子）和对象的组合（树枝）用同样的方式来对待。生活例子：电脑里的文件夹。你可以对单个文件（叶子）进行"复制"操作，也可以对整个文件夹（树枝，里面包含文件和子文件夹）进行同样的"复制"操作。你操作的方式是一样的。**优点**：简化了客户端代码，可以用统一的方式处理简单对象和复杂对象。**缺点**：设计变得更抽象，有时会牺牲一定的类型安全。

(5) 装饰模式

一句话解释：在不改变原有对象的基础上，动态地给它套上一层或多层"包装"，来增加新功能。生活例子：拍照用滤镜。你有一张原图（被装饰对象），可以给它套上一个美白滤镜（装饰者），再套上一个复古滤镜（另一个装饰者），每一层都增加了新的效果，而且可以自由组合。**优点**：比继承灵活，可以在运行时动态地给对象增加或删除功能。**缺点**：容易产生很多功能单一的小类，增加系统的复杂性。

(6) 观察者模式

一句话解释：定义了一种一对多的依赖关系，当"被观察者"状态改变时，所有"观察者"都会自动收到通知。生活例子：微信公众号和粉丝。公众号（被观察者）发布新文章，所有关注它的粉丝（观察者们）都会收到推送通知。**优点**：实现了发布者和订阅者之间的解耦，双方不需要知道对方的具体细节。**缺点**：如果观察者太多，通知过程可能会比较耗时。

(7) 适配器模式

一句话解释：像一个"转接头"，把一个不兼容的接口转换成另一个客户端期望的接口。生活例子：插座转换器。你的笔记本电脑插头是国标的，但酒店墙上的插座是英标的，插座转换器（适配器）让你能把它们连接起来。**优点**：提高了类的复用性，让原本不兼容的类可以协同工作。**缺点**：会额外增加一些适配器类，让系统稍微复杂一些。

七、编码实现(简答题)

1. 编码规范及代码风格

(1) 程序代码的质量要求

●健壮性与高性能：程序要稳定、不易崩溃，并且运行效率高。

●可读性与可维护性：代码要像文章一样清晰易懂，方便自己和他人日后修改和维护。

●可扩展性与可重用性：代码要容易增加新功能，并且其中的模块可以方便地在其他地方复用。

●可测试性：代码要易于编写单元测试，以保证其正确性。

(2) 程序代码的编码规范

(3) 程序代码风格

●编码规范。**是什么**：一套需要严格遵守的编码规则，旨在提高代码的正确性、健壮性。通常由组织或行业制定（如谷歌规范、公司内部规范）。**管什么**：变量如何声明、关键字如何使用、异常如何处理等语法和行为层面的规则。

●代码风格（Code Style）。**是什么**：关于代码格式和外观的约定，如命名、缩进、注释等。**管什么**：风格通常没有绝对的对错，但团队内必须保持统一，以提高可读性。

●关键风格要素：**清晰的命名**：变量、函数名要做到望文生义。**合理的缩进与对齐**：清晰地反映代码的层次结构。**适当的空行**：用来分隔不同的逻辑代码块。**有意义的注释**：对复杂的逻辑或重要的前提进行说明，注释量建议在 20%以上。

(4) 代码审查

●核心目标：通过系统化地审查源代码，来发现程序错误、安全漏洞和不规范之处，以提升软件质量和团队技术。

●审查方式：**人工审查**：同行评审是最主要的方式，具体做法包括结对编程（最即时的审查）、正式的审查会议、轻量化的非正式检查（如发一个合并请求 Merge Request 让同事看）。**工具审查**：使用静态代码分析工具自动化地扫描和发现问题。

●审查内容（由浅入深）：是否遵循了基本编码规范。是否存在程序逻辑错误。软件设计是否合理、可扩展。

2. 代码重用

(1) 代码重用的概念

●核心定义：对已有写好的代码进行重复使用以构建新的软件。

●优点与缺点：**优点**：提高开发效率、降低开发成本、使软件更简洁。**缺点**：可能增加不可控的外部依赖，如果重用的代码质量差，会引入风险。

●重用原则：最重要是只重用高质量、经过充分评估的代码。

●可重用代码的特点：好的可重用代码通常具有模块独立性强、接口清晰、易于扩展等特点。

(2) 代码重用的方式和方法

从重用代码的"粒度"从小到大来划分的：

●代码片段重用（粒度最小）：**是什么**：重用具体的函数、类、库文件。**静态库**：代码在编译时被完整地复制并打包进最终的可执行文件中。 优点：部署简单。缺点：程序体积大，可能造成内存浪费。**动态库**：程序在运行时才去加载和链接所需的库文件。 优点：程序体积小，多个程序可共享同一份库，方便更新。缺点：部署时必须带上对应的库文件。

●面向对象的重用机制：**是什么**：利用面向对象语言的特性来实现代码的灵活复用。主要机制：继承（子类复用父类代码）、组合（将一个对象作为另一个对象的成员）、多态（统一接口，不同实现）、接口、重载等。

●组件重用（中等）：重用一个完整的功能模块或服务。例如，在一个新项目中直接使用一个已经开发好的用户认证组件。

●框架重用（粒度最大）：也叫"半成品"式重用。框架提供了一整套软件的骨架和规范，开发者只需在预留的"插槽"中填充自己的业务逻辑代码。

3. 程序调试

(1) 程序调试的概念

●核心目标与流程：**目标**：定位并移除程序中的缺陷、错误和异常。**基本流程**：理解缺陷症状 → 定位缺陷代码 → 修复代码并进行回归测试（确保没有因为修改而引入新的问题）。

●缺陷类型：**语法错误**：不符合语言规范，编译阶段就能发现。运行时错误：程序在运行过程中发生的错误，如空指针引用、除以零等。逻辑错误：程序能正常运行，但结果不符合预期。这是最隐蔽、最难排查的错误。

●定位缺陷的主要方法：直接法：根据错误信息直接推断问题所在。回溯法：从出错的位置开始，反向追踪代码的执行路径和数据变化。排除法（二分法）：通过注释掉部分代码来逐步缩小问题范围，最终定位到有问题的代码块。

(2) 程序调试的技术和工具

●主要技术：**打印/日志**：最简单直接的方法，通过在代码中输出中间变量或日志来观察程序状态，尤其适用于调试线上服务。**使用调试器**：利用专业的调试工具（如 GDB）设置断点，单步执行代码，实时查看变量值内存状态和调用堆栈。

●辅助工具分类：**调试器**：如 GDB，用于断点、单步执行。**性能分析工具**：用于分析程序的性能瓶颈，如哪里耗时最长。**日志分析工具**：如 ELK，用于聚合和分析海量日志。**单元测试框架**：如 JUnit，通过编写测试用例来验证代码模块的正确性，也是调试的重要辅助。**静态分析工具**：在不运行代码的情况下，检查代码是否符合规范，是否存在潜在的缺陷。

八、软件测试(分析设计)

1. 软件错误、缺陷和故障（概念/差异）

软件错误:代码问题(人为错误)。**软件缺陷**:软件未达要求/出现错误/超出范围/未实现不在产品说明书但应实现的功能/不易使用

软件故障:运行时执行错误结果。**差异**：错误(开发)->缺陷(产品)->故障(运行)

2. 软件测试的概念、思想原理、目标准则

●核心思想：在规定的条件下运行或审查一个软件产品，以发现其中的缺陷（即与需求不一致的地方），并对其质量、能否满足用户需求进行评估。

●**静态测试**（不运行代码）。通过评审、走查、静态分析工具等方式，检查代码的语法、结构和逻辑。主要方法：代码走查（非正式）、技术评审、代码审查（正式）。**动态测试**（运行代码）。通过实际运行程序，检查其功能、性能等是否符合预期。**按是否关心内部代码划分**：黑盒测试（不关心）、白盒测试（关心）。**按开发过程划分**（重点）：单元测试 → 集成测试 → 系统测试 → 验收测试。**按实施组织划分**：α 测试（内部用户在开发环境下测试）、β测试（外部真实用户在实际环境测试）。

●**直接目标**：尽早、尽快地发现和预防缺陷。**长期目标**：通过发现缺陷来提高软件质量、提升客户满意度和管理风险。**发布后目标**：通过前期的充分测试来降低后期维护成本，并为改进未来的测试过程提供数据支持。

●**准则 1**：测试只能证明缺陷存在，但不能证明缺陷不存在。**准则 2**：穷尽测试是不可能的。**准则 3**：测试活动应尽早开始，贯穿整个生命周期。**准则 4**：缺陷具有集群性（即"二八定律"，80%的缺陷可能集中在 20%的模块中）。**准则 5**：杀虫剂悖论，同样的测试用例反复使用会"失效"，必须定期更新和评审测试用例。**准则 6**：测试依赖于具体内容，不同类型的系统，测试策略和方法也应不同。**准则 7**："没有发现错误"不等于"系统是有用的"，解决了所有错误但没满足用户真实需求的系统依然是失败的。

3. 软件测试用例（概念/构成/表示）

●核心定义：为确定软件是否满足需求而设计的一组输入数据、执行步骤和预期结果的集合。

●核心构成要素：**测试目标/名称**：要测什么功能。**前置条件**：测试前需要满足的环境或数据状态。**测试步骤**：具体的操作流程。**预期结果**：正常情况下应该出现什么样的结果。**实际结果与状态**：实际执行后得到的结果，以及测试是否"通过"。

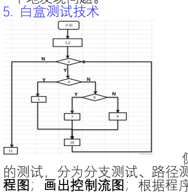
4. 软件测试过程和策略

●基本流程：① 分析测试需求 →② 制定测试计划 →③ 设计测试用例 →④ 执行测试 →⑤ 评估与报告。

●软件测试策略（各测试阶段-重点）：**单元测试**：对软件中最小的可测试单元（如一个函数或类）进行测试。通常由开发者做，采用白盒测试。**集成测试**：在单元测试后，将多个模块组合起来，测试它们之间的接口和协作是否正常。**系统测试**：将整个软硬件系统看作一个整体，对照需求规格说明书，检查系统功能和非功能特性是否达标。主要采用黑盒测试。**验收测试**：软件发布前的最后一道测试，通常由最终用户或客户来进行，以确认软件是否能满足他们的实际使用需求。

●测试过程模型：**V 模型**：强调了测试与开发的对应关系，每一个开发阶段都有一个相应的测试阶段与之对应，但它依然是串行的。**W 模型**：V 模型的演进，强调了测试与开发应并行进行，在每个开发阶段都同步进行验证和确认活动，能更早地发现问题。

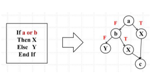
5. 白盒测试技术

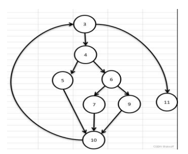


```
void Sort ( int iRecordNum, int iType ) {
1       int x=0;
2       int y=0;
3       while ( iRecordNum-- > 0 ) {
4           if ( iType==0 )
5               x=y+2;
6           else if ( iType==1 )
7               x=y+10;
8           else
9               x=y+20;
10      }
11  }
```

侧重于系统或部件内部机制的测试，分为分支测试、路径测试、语句测试。**画出程序流程图**；**画出控制流图**；根据程序环形复杂度的计算公式，**求出**程序路径集合中的**独立路径数目**；根据环形复杂度的计算结果，源程序的基本路径集合中有多少条**独立路径**；**设计测试用例**

步骤：给每条语句编号->根据代码逻辑画流程图->根据流程图简化为控制流图->公式计算->写路径->根据路径写测试用例





上为(1)流程图和(2)控制流图

注:若判定点有复合条件应按如左方式增加判定点

(3)环复杂度计算:

法一:前提单入口单出口 V(G)=E-N+2 E 为边数,V 为结点数

图中 V(G)=10-8+2=4

法二:前提仅计算二分支判定节点 V(G)=P+1

P 为二分支节点数 图中二分支节点为 3、4、6

V(G)=3+1=4

(4)基本路径测试:所有可能路径 (5)测试用例:根据路径写

path-1: 3 - 11

path-2: 3 - 4 - 5 - 10 - 3 - 11

path-3: 3 - 4 - 6 - 7 - 10 - 3 - 11

path-4: 3 - 4 - 6 - 9 - 10 - 3 - 11

测试用例

编号	输入数据		覆盖路径	预期结果	
	iRecordNum	iType		x	y
test-1	0	0	path-1	0	0
test-2	1	0	path-2	2	0
test-3	1	1	path-3	10	0
test-4	1	2	path-4	20	0

逻辑覆盖:

语句覆盖(点覆盖):设计测试用例时应保证程序中每一条可执行语句至少应执行一次

判定覆盖(边覆盖):保证每个判定的每个分支至少一次(只关注整个表达式结果 T/F)

条件覆盖:保证每个判定的每个条件都至少取真假各一次(如(y==3)||(x>1)T||F/F||T、整体结果随便)

判定-条件覆盖:满足判定节点的取真、取假分支至少执行一次，且每个简单判定条件（子条件）的取真和取假情况也至少执行一次(如(y==3)||(x>1)T||T->T/F||F->F、条件和整体)

条件组合覆盖:满足每个判定节点中，每个判定的每个条件所有可能取值组合至少执行一次。通过列出真值表的方式来得到完全的覆盖(如(y==3)||(x>1)TF/FT/TT/FF)

6. 黑盒测试技术

黑盒测试:检测每个功能是否都能正常使用,不考虑程序内部结构和处理过程

等价类划分法:把所有可能输入数据，即输入域划分成若干互不相交的子集，称为等价类

输入等价类	有效等价类	无效等价类
日期的类型及长度	①6位数字字符	②有非数字字符 ③少于6位数字字符 ④多于6位数字字符
年份范围	⑤在1990~2049之间	⑥小于1990 ⑦大于2049
月份范围	⑧在01~12之间	⑨等于00 ⑩大于12

测试数据	期望结果	覆盖的无效等价类
95June	无效输入	②
20036	无效输入	③
2001006	无效输入	④
198912	无效输入	⑥
200401	无效输入	⑦
200100	无效输入	⑨
200113	无效输入	⑩

边界值分析法:在输入或输出变量范围的边界值上进行测试。某程序要求输入长方体的长 x、宽 y、高 z。x、y、z 的上界为 20，下界为 2。给出健壮性边界值分析的测试用例。

测试用例	x	y	z	预期输出
Test Case 01	10	10	10	1000
Test Case 02	1	10	10	ERROR(x小于2)
Test Case 03	2	10	10	200
Test Case 04	3	10	10	300
Test Case 05	19	10	10	1900
Test Case 06	20	10	10	2000
Test Case 07	21	10	10	ERROR(x大于20)
Test Case 08	10	1	10	ERROR(y小于2)
Test Case 09	10	2	10	200
Test Case 10	10	3	10	300
Test Case 11	10	19	10	1900
Test Case 12	10	20	10	2000
Test Case 13	10	21	10	ERROR(y大于20)
Test Case 14	10	10	1	ERROR(z小于2)
Test Case 15	10	10	2	200
Test Case 16	10	10	3	300
Test Case 17	10	10	19	1900
Test Case 18	10	10	20	2000
Test Case 19	10	10	21	ERROR(z大于20)

错误推测法:列出所有可能有的错误,靠直觉

判定表法、正交实验法、因果图法、功能图法

九、软件部署和维护(简答)

1. 软件维护概念和形式(概念、形式、类别、因素)

(1) 概念

●软件交付使用后，为了修复缺陷、增强功能、提高性能或适应变化的环境而对其进行修改的过程。其主要特点是长期性、复杂性、必要性。

(2) 类别

●**纠错性维护**：修复在运行中发现的错误（改 Bug）。**适应性维护**：为了适应外部环境（如操作系统升级、政策法规变化）而进行的修改。**完善性维护**：根据用户的新需求，增加新功能或改进现有功能，以提升用户满意度。**预防性维护**：为了提高软件未来的可维护性和可靠性而主动进行的修改（如代码重构、优化文档）。

(3) 形式

●形式（影响可维护性的内部因素):软件代码和设计的可理解性。软件的可测试性。软件的可修改性。

(4) 因素

●清晰、合理的软件结构。高质量、同步更新的文档。统一、规范的编码风格。全面、自动化的测试保障。

2. 软件维护过程和技术(任务、过程、原则、策略、技术)

(1) 任务

●主要包括问题诊断、缺陷修复、功能更新、性能优化、兼容性适应、代码重构、文档更新等。

(2) 过程

●问题跟踪与报告，分析问题并定位原因，设计并评审修改方案，实施修改，进行回归测试并验证修复效果

(3) 原则

●在修改前，要充分理解和评估修改的影响范围。对重要的修改，应先评审修改方案再动手。修改后必须进行充分的回归测试，防止引入新的缺陷。

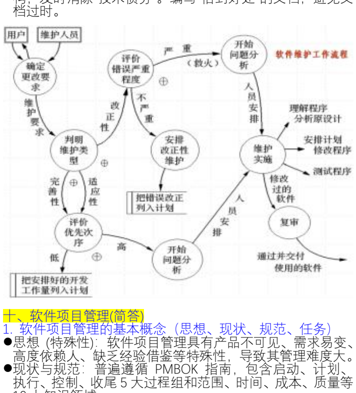
(4) 策略

●反应性维护（被动式）：出现问题后再去修复。前摄性维护（主动式）：在开发阶段就预先考虑和设计系统的可维护性，从源头减少未来的维护成本。

(5) 技术

●**①面向缺陷的维护(DOM):**以发现和修复软件缺陷为主要目标、维护成本较高、适用于高可靠性软件**②面向功能的维护(FOM):**以满足用户需求和改进软件功能为主要目标、适用功能复杂生命周期长的软件**③软件再工程:**对已有的软件系统进行重构和改造,提高影响可维护性的几个因素(代码重构、数据库重构、系统重构、过程重构)**④软件逆工程:**分析已有程序寻求比源代码更高层次的抽象形式（如设计甚至需求）

●**敏捷方法中的可维护性保障:**测试驱动开发（TDD）和持续集成提供了强大的安全网。在每个迭代周期中都进行代码重构，及时消除"技术债务"。编写"恰到好处"的文档，避免文档过时。



十、软件项目管理(简答)

1. 软件项目管理的基本概念（思想、现状、规范、任务）

●思想（特殊性）：软件项目管理具有产品不可见、需求易变、高度依赖人、缺乏经验借鉴等特殊性，导致其管理难度大。

●现状与规范：普遍遵循 PMBOK 指南，包含启动、计划、执行、控制、收尾 5 大过程组和范围、时间、成本、质量等 10 大知识领域。

●任务：其核心任务是在预定的进度、成本和质量要求下，开发出满足用户需求的软件产品。

2. 软件项目估算和计划(工作量估算方法)

●估算基础：先估算项目规模，常用单位有代码行(LOC)和功能点(FP)。常用方法有专家评估法(Delphi)和参数法。

●工作量估算方法:**估算单位**：人月（一个人干一个月的工作量）。**核心方法**： COCOMO 模型，它根据项目类型（有机型、嵌入式等）使用不同公式计算工作量。

●项目计划：是描述如何开展工作的详细文档，内容包括目标、任务分解、进度安排、人员分工、资源分配等。

3. 软件项目风险管理(概念、类别、方法)

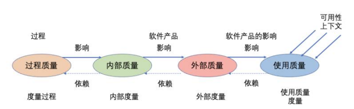
●概念：风险是开发过程中可能对项目造成损失或负面影响的不确定因素。

●类别：**需求风险**（如：需求不清晰、频繁变更）。**人员风险**（如：核心人员流失）。**技术风险**（如：技术方案不可行、性能不达标）。**计划风险**（如：估算过于乐观、计划不具体）。

●方法（风险管理过程）：主要分为风险评估（识别、分析、排序）和风险控制（制定计划、化解、监控风险）。

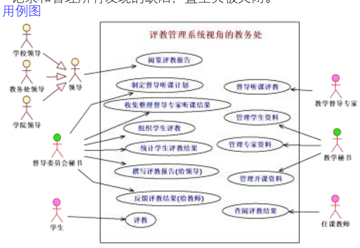
4. 软件项目质量保证(管理内容、质量保证方法)

●管理内容：质量管理不仅关注最终的产品质量（软件本身好不好），也关注过程中的工作质量（开发过程规范不规范），因为后者是前者的保证。

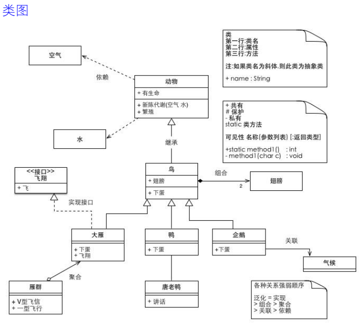


●质量保证方法:**制定质量计划**：为项目设定明确的质量目标和标准。**进行技术评审**：对需求、设计、代码等产物进行评审。**实施软件测试**：通过测试活动发现并修复缺陷。**检查开发过程**：确保团队遵循了既定的开发流程和规范。**缺陷跟踪**：记录和管理所有发现的缺陷，直至其被关闭。

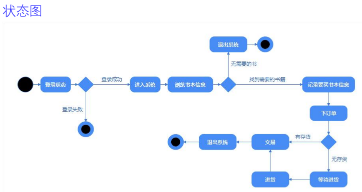
用例图



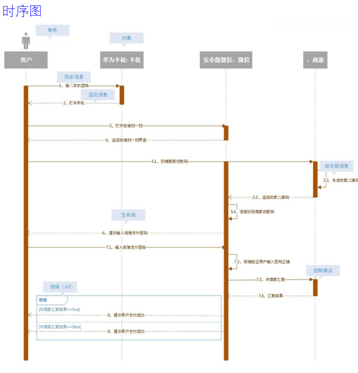
类图



状态图



时序图



Github链接：https://github.com/raitocc/WHUCS-Software-Engineering-A4Paper.git