ChatGPT

软件工程课程复习速查笔记

需求建模

- **软件需求定义及分类**：软件需求是利益相关方对目标系统的**要求和期望** [1] 。主要分为**功能需求**（系统应提供的功能，例如“控制月球车移动”）和**非功能需求**（性能、可靠性、安全性等质量需求，以及开发约束等） [2] 。功能需求描述系统应**做什么**，非功能需求描述系统应**如何**（质量、约束）运行。通常功能需求通过用例等表述，非功能需求包括性能指标、可靠性指标、平台限制等。
- **需求工程过程**：包含一系列活动以获取和管理需求 [3] ：① **需求获取/抽取**（Elicitation）：与干系人协作收集需求，识别冲突并确定系统边界 [4] ；② **需求分析**（Analysis）：对收集的需求进行分类、澄清和冲突协调，可能进行**概念建模**（使用建模语言表示功能需求和领域概念） [5] ；③ **需求规约**（Specification）：将需求编写成规范文档（软件需求说明书 SRS），要求需求**准确、完整、一致**等 [6] ；④ **需求验证**（Validation）：检查需求正确性和可行性，确保需求满足干系人意图（可通过评审、原型或验收测试验证） [7] ；⑤ **需求管理**（Management）：对需求变更进行控制和跟踪，包括需求属性记录、版本控制、需求跟踪矩阵等 [8] 。
- **需求获取技术**：常用需求抽取方法包括：**访谈**（与用户和客户交流获取需求，要善于提问引导），**问卷调查**、**研讨会/焦点小组**（召集用户和开发团队讨论需求），**原型**（构建原型系统以澄清需求），**文档研究**（分析现有系统或业务资料），**观察**（观察用户实际操作）等。协同工作会议也是有效手段；使用**检查表**核对常见非功能需求；还需**冲突识别与协商**来解决不同干系人之间需求矛盾 [9] 。需求获取时应定位真正的业务问题，避免只记录表面需求 [10] 。
- **强调常考概念**：需求按**来源**可来自用户、客户、法律法规等；按**重要性**可分为必备、期望和可选需求。**利益相关者**（Stakeholders）指影响或被系统影响的所有人（用户、客户、开发者、维护人员等）。**需求变更**频繁且难免，要有变更管理机制（例如需求基线和变更控制）。考试常考功能 vs 非功能需求区别、需求工程活动步骤、获取需求的方法等概念区分。

UML建模

- **UML概述**：统一建模语言（UML）是一种用于**可视化、描述、构造和文档化**软件系统蓝图的标准建模语言 [11] 。UML提供多种图表来从不同视角表示系统，如用例图表示功能需求，类图表示静态结构，顺序图表示动态交互等。
- **用例图 (Use Case Diagram)**：用于描述系统与外部参与者（Actor）的交互及系统提供的功能。**参与者**代表与系统交互的角色（人或外部系统），**用例**表示参与者利用系统实现的一个功能场景。用例图包含参与者、用例以及它们之间的关系（关联线表示参与者参与用例）和系统边界。常见关系有：**泛化**（继承，参与者或用例的分类层次）、**包含**（include，将多个用例的公共行为提取为被包含用例）和**扩展**（extend，将用例中特殊或可选流程拆分为扩展用例）。绘制步骤包括：确定系统边界，找出外部参与者及其期望的系统功能，命名用例，并识别用例间的包含/扩展关系 [12] 。用例图有助于从用户视角梳理需求，确保所有功能点都被定义且无多余功能 [13] 。**常考点**：参与者 vs 用例区别、包含与扩展关系区别，用例图如何表示系统边界和交互等。
- **类图 (Class Diagram)**：表示系统的静态结构。类图由**类**（具有属性和操作的抽象数据类型）构成，展示类与类之间的关系。关键元素包括：**类（Class）**及其属性（成员变量）和方法（操作）；**接口（Interface）**（定义一组操作，无实现，由类实现）。类之间常见关系：① **关联**（Association）：表示类之间语义上的连接，通常有角色名和多重性（如1对多）；特例包括**聚合**（Aggregation，空心菱形，表示整体与部分的松散组合）和**组合**（Composition，实心菱形，表示严格的部分-整体关系，部件生命周期依赖整体）。② **泛化**（Generalization）：表示继承/一般化关系（空心三角箭头指向父类），子类继承父类属性和操作。③ **实现**（Realization）：类实现接口（空心三角虚线箭头）。④ **依赖**（Dependency）：临时使用关系（虚线箭头）。类图着重体现系统的模块结构和对象之间的静态依赖

关系，例如类的层次结构和关联关系等。**常考概念**：聚合 vs 组合区别、泛化和实现的语义、多重性符号的意义（如1..* 等）。

- **顺序图 (Sequence Diagram)**：属于交互图，按时间顺序显示对象之间的消息交换，用于描述**特定场景**下的行为流程。顺序图中的垂直方向表示时间流逝，水平方向列出不同对象（参与者或类实例）的**生命线**，生命线上的**激活条**表示对象执行过程 [14]。对象之间以箭头表示消息（调用），实线实箭头表示同步调用，虚线箭头表示返回，亦可标注消息名称。顺序图直观展现**用例**场景中对象协作的过程（一个用例通常可用一个顺序图来表示其主要交互流程 [15] [16]）。它能帮助分析系统行为、验证用例逻辑，并可细化为后续设计阶段的通信图或代码架构。**重点**：会识别顺序图元素（对象生命线、激活、消息）以及它如何对应用例场景，区分顺序图与其他交互图（如协作图）的侧重点（顺序图强调时间顺序，协作图强调对象关系）。
- **其他UML图简述**：除了上述常用模型图，UML还包括**状态图**（Statechart，用状态迁移描绘对象生命周期）、**活动图**（Activity，用流程图表示算法或业务流程）、**组件图**（Component，展示软件物理组件依赖）、**部署图**（Deployment，表示运行时节点和部署关系）等。状态图与活动图用于动态行为建模（状态图偏对象状态变化，活动图偏流程顺序），组件图和部署图用于体现体系结构的实现视图。这些图在特定场景下可能出现，但考试一般侧重用例图、类图、顺序图等核心模型。

软件测试

- **测试基础与原则**：软件测试是为了**发现程序中的错误**而执行程序的过程 [17]。测试的直接目标是尽早找出并修复缺陷，评估软件质量 [18]。**测试基本原则**包括 [19] [20]：1）测试应有预期的输出检查（必须为每个测试用例定义期望结果）；2）**避免由开发者自己测试其代码**（独立的第三方测试更易发现问题）；3）开发组织不应测试自己开发的软件（避免偏见）；4）**彻底检查测试结果**，不能忽略任何异常；5）用例设计既要考虑**有效输入**也要涵盖**无效输入**（包括意外或非法情况）；6）要检查程序**“应做未做”**和**“不该做却做了”**两方面；7）测试用例应**可重用**，除非一次性软件，避免用后即弃；8）计划测试时默认会发现缺陷，不应假定“没有错误”；此外，“缺陷的集群”现象表明错误往往集中在少数模块（80/20原则） [21]，需要对高风险区重点测试；还要注意“杀虫剂悖论”，重复使用相同测试用例将逐渐无法发现新缺陷 [22]，因此需要定期评审并增加新用例。测试只能证明存在缺陷，**不能证明没有缺陷**，即经过测试的软件仍可能存在隐藏错误。
- **测试级别（V模型）**：软件测试分多个层次，与开发活动对应：① **单元测试**：针对最小单元（函数/模块）的测试，通常由开发人员采用白盒方法进行，验证模块内部逻辑的正确性。② **集成测试**：在单元测试基础上，将模块按设计要求组装成子系统进行测试，重点检查模块间接口和交互问题 [23]。验证数据在模块接口传递是否丢失或不良影响，发现模块调用关系、全局数据等集成缺陷。③ **系统测试**：针对完整的集成系统进行测试，验证系统功能和非功能需求是否满足 [24]。系统测试在尽可能逼真的环境下执行，包括软硬件、网络、用户等，属于**黑盒测试**为主，检查系统整体行为（功能性正确性、性能、安全性等）。④ **验收测试**：也称交付测试，由客户或独立测试方执行。包括**Alpha测试**（由用户在开发环境或内部模拟真实场景测试）和**Beta测试**（在部分真实用户中发布试用收集反馈），目的是确保软件在实际使用中可被接受。考试常要求比较这些测试级别的**对象、目的**区别，注意：单元测试面向模块代码细节，集成测试关注接口交互，系统测试验证整体需求，实现逐级扩大测试范围。
- **白盒 vs 黑盒测试**：按照测试方法对比：**白盒测试**（结构测试）把被测软件视为透明盒子，利用程序内部逻辑来设计测试用例，检查内部路径是否按预期运行 [25]。白盒测试要求熟悉代码，典型技术有**语句覆盖**、**分支覆盖**、**路径覆盖**等度量执行路径的覆盖率。**黑盒测试**（功能测试）把被测对象视为黑箱，不考虑内部实现，只依据需求规格说明来设计输入和检查输出 [26]。黑盒侧重验证功能行为和用户可见性能，例如给定输入是否产生期望输出。简言之，白盒测试关注程序结构正确性，黑盒测试关注功能满足需求。考试中常要求比较两者异同：黑盒能发现**遗漏的功能或界面错误**，对白盒难以察觉的内部边界值敏感错误也有效；白盒能检查**代码实现细节**，覆盖隐藏在代码中的逻辑路径。实际测试中应综合采用白盒和黑盒方法 [27]：单元/集成测试阶段偏重白盒，系统/验收测试偏重黑盒。
- **黑盒测试方法**：为了系统地设计黑盒测试用例，有多种经典方法：
- **等价类划分**：将输入域划分为若干等价类（子集），使得每个类内输入预期引起相似行为 [28]。选取每个有效等价类的代表值作为测试用例，尽可能覆盖所有有效类；同样针对每个无效等价类各取一代表值测试异常处理 [29] [30]。这样可用有限用例覆盖无限输入空间。等价类包括有效类（符合规格的输入集

合）和无效类（不符合规格，如越界值）。需注意同时涵盖有效和无效等价类测试，软件应对不合理输入有适当处理 [30]。

- **边界值分析**：针对倾向于在边界处出现错误的情况，选择等价类边界附近（含临界值、刚过界值等）作为测试数据。通常包括最小、最大值及其临近值。边界值测试能有效发现例如数组越界、取值溢出等错误。
- **决策表测试**：将复杂业务规则的各种条件组合与对应动作以表格形式表示，每一列代表一种条件组合下的结果，确保测试用例覆盖所有规则情形。适用于多条件逻辑，例如权限判定、多重业务规则等。
- **因果图**：先画出条件与结果的因果关系图，再转换为判定表生成测试用例。它考虑条件间的逻辑关系（与或等）和约束，确保测试逻辑完整性。适用于复杂逻辑要求，根据输入条件推导输出。
- **场景法**：基于真实使用场景设计测试用例，即按照用户使用软件的流程和业务场景编写测试。侧重于跨功能的场景串联测试，可发现集成问题或流程逻辑错误。典型如电商下单场景测试涵盖浏览、下单、支付各环节。
- **正交试验法**：运用正交表（拉丁方）系统地挑选参数组合进行测试，以较少用例覆盖多因子组合。适合于参数较多、每个参数取值有限的情况，通过有代表性的组合减少测试总量又保证覆盖广泛。

（黑盒测试重点在于覆盖输入的各种等价情况和组合，考试常要求说明等价类和边界值方法，并正确划分例子中的等价类或识别边界值。） - **集成测试**：在模块单元测试通过后，逐步将模块按设计集成为更大的子系统并测试接口。【定义】集成测试（Integration Testing）是将经过单体测试的模块组装起来按总体设计要求进行测试，以发现**模块接口之间**存在的问题 [23]。重点关注：跨模块调用时数据是否正确传递，有无接口失配，一个模块的行为是否会对另一个产生不良影响等 [31]。**集成策略**常考有：① **自顶向下**集成：从系统顶层主控模块开始，逐步替换桩模块向下集成真实模块 [32]。需要**桩模块**（stub）模拟下层未实现模块，被测模块调用桩以测试接口 [33]。优点是主要框架先搭建，验证高层逻辑；缺点是底层模块晚集成可能缺陷定位延后。② **自底向上**集成：从最低层模块开始，逐步向上组装，需要**驱动模块**（driver）模拟调用上层 [33]。优点是底层单元测试完成即可立即集成，缺点是系统架构只有在高层加入后才能验证整体流程。③ **大爆炸**集成：所有模块单元测试后一次性全部集成测试 [34]。实现快但缺陷排查困难，不推荐。④ **三明治（混合）集成**：结合自顶向下和自底向上，分层同时集成，减轻双端依赖。**集成测试原则**：尽早优先集成/测试**关键模块**（承担核心功能的模块） [35]；公共接口都应被测试到 [36]；集成应分阶段分层次，不要一次性集成过多；每次增量集成后执行回归测试确保新缺陷未引入。集成测试通常采用**组合的测试方法**（接口逻辑用白盒，功能用黑盒） [37]。考试可能要求解释桩模块和驱动的作用、比较不同集成策略优劣等。 - **系统测试**：在集成完成后，对整个软件系统进行的全面测试。系统测试以**需求规格说明书**为依据，从最终用户视角验证软件整体满足功能和非功能需求。包括：**功能测试**（验证各功能是否实现，通常以用例或用户场景为基础的黑盒测试 [38]）和**非功能测试**（针对性能、负载、安全、易用性等品质特性）。例如性能测试检查系统在高负载下响应时间和资源占用；安全测试评估系统防护能力等。系统测试环境应模拟实际部署环境（硬件、网络、用户权限等）以发现环境相关问题。**验收测试**作为系统测试的最后阶段，由客户进行，以确认软件是否达到了可交付的质量标准。典型包括Alpha测试（开发场所，由用户和开发共同参与，发现潜在问题）和Beta测试（有限用户群试用，在真实环境下使用软件，反馈缺陷和改进意见）。系统测试和验收测试强调**从用户需求出发**进行验证，是对整个软件生命周期工作的最终检验。常见考点：区分功能测试与非功能测试，Alpha vs Beta测试意义等。 - **持续集成（CI）**：一种软件开发实践，要求团队成员频繁地将代码集成到主干，每次集成都通过自动构建来验证 [39]。通常每位开发者**每日至少集成一次**代码变更 [40]，由持续集成服务器自动完成编译、部署并运行自动化测试，以快速发现集成错误。要点：开发者在本地先构建和测试再提交版本库，保持主干稳定；持续集成服务器检测到代码更新后触发构建和回归测试，及时报告问题。持续集成有助于及早发现模块集成兼容性问题，防止“大爆炸集成”导致的调试困难，并支持快速迭代的开发模式。很多敏捷开发流程（如XP）都强调持续集成。考试可能将持续集成与传统集成方式对比，重点在于其**高频次、小增量、自动化测试**的特征，以及带来的质量改进和效率提升 [41]。

软件设计原则与方法

- **模块化设计与高内聚低耦合**：将软件系统划分为**模块**可以降低复杂度，模块化的核心原则是**关注点分离**，即将不相关的功能职责分开 [42]。衡量模块划分质量的关键是**内聚性**和**耦合度**。**内聚**指模块内部元素彼此关联的紧密程度，一般划分有偶然内聚、逻辑内聚、时间内聚、过程内聚、通信内聚、功能内聚等层次，**功能内聚**（模块内所有元素共同完成单一功能）最理想 [43] [44]。**耦合**指模块间依赖强度，包括内容耦合、公共耦合、控制耦合、标记（数据结构）耦合、数据耦合等，**数据耦合**（模块仅通过简单

参数传递数据）是最松散、优良的耦合类型 [45] [43]。理想设计追求**高内聚、低耦合** [45]：高内聚让模块内部职责单一明确，低耦合使模块之间松散独立，修改一个模块对其他的影响最小。实现手段包括**信息隐藏**（隐藏模块内部实现细节，仅通过接口交互 [46]）、**抽象**（提取共性特征形成模块接口或父类，屏蔽具体实现 [47]）、以及**接口设计简洁**（减少模块对环境的假设和依赖 [46]）。这些原则提高代码灵活性、可维护性和可扩展性 [48]。考试常考：定义内聚和耦合、举例说明高低耦合情况、信息隐藏的意义等。

- **设计原则 (SOLID)**：面向对象设计中，有若干遵循高内聚低耦合思想的经典原则，经常简称为SOLID：
- **单一职责原则**（SRP）：一个类只负责一项职责，即**仅有一个导致其修改的原因** [49]。遵循SRP可增强内聚，降低变更影响。
- **开放封闭原则**（OCP）：软件实体应**对扩展开放，对修改封闭**。通过抽象和多态，在无需修改已有代码情况下扩展新功能，提高系统灵活性和稳定性。
- **里氏替换原则**（LSP）：所有基类能够使用的地方，子类都可以透明替换且保证程序行为正确 [50] [51]。即子类不应破坏基类的约定（契约），确保继承的正确性和可靠性。
- **接口隔离原则**（ISP）：使用多个专门的接口，而不使用一个庞大臃肿接口。客户端不应被迫依赖不需要的方法。该原则减少依赖，实现**高内聚接口**设计。
- **依赖倒置原则**（DIP）：高层模块不应依赖低层模块，二者都应依赖其抽象；抽象不应依赖细节，细节应依赖抽象。通过引入抽象接口，使得具体实现变化不影响高层逻辑，实现模块解耦。
  这些原则指导创建更易维护和扩展的OO设计。考题可能让解释某一原则或识别违反该原则的设计例子。
- **其他设计方法与实践**：软件设计过程中，可采用**迭代增量式设计**（逐步细化完善设计 [52]），每次增量开发高优先级功能，逐步演进系统架构。**设计复用**也是重要原则：尽量利用已有组件、库或框架，提高开发效率和质量 [53]。在模块划分时，可选择**自顶向下**逐步分解问题（先高层抽象设计，再细化子问题）或者**自底向上**组合已有组件。此外还有**KISS原则**（Keep It Simple, Stupid，设计尽量简单），**DRY原则**（Don't Repeat Yourself，避免重复代码，通过抽取公共模块来复用），**YAGNI**（You Ain't Gonna Need It，不做当前不需要的设计超前实现）等，这些敏捷理念强调简洁和适应变化。设计过程中应平衡**质量属性**（如性能 vs 易读性），综合考虑扩展、维护、安全等需求，避免过度设计和不足设计。考试关注点：理解并举例说明高内聚/低耦合、信息隐藏等原则的作用；能列出SOLID原则的含义。

软件体系结构风格

- **概念**：软件体系结构是关于系统高层结构和全局组织的设计，包括组件划分和组件交互方式。**体系结构风格（架构模式）**是可复用的架构设计范式，描述了特定环境下系统组件组织和交互的通用方式。掌握常见架构风格有助于选择合适的系统设计方案。
- **分层架构 (Layered)**：将系统按抽象层次分为层级，每层只与相邻层交互。典型是**三层架构**：表示层（UI）、业务逻辑层、数据层，各层职责单一。例如表示层处理界面交互，业务层处理业务规则，数据层负责数据存储访问。高层可以调用低层服务，低层不依赖高层，实现良好解耦和演化。优点：层次清晰，低层可以被复用和独立替换；缺点：可能降低性能（增加通信开销）。考试常让描述三层架构的各层职责。
- **客户端-服务器架构 (C/S)**：分为**客户端**和**服务器**两类组件，客户端发起请求，服务器处理请求并返回结果。常见两层C/S如典型数据库应用：客户端实现UI和部分业务逻辑，服务器提供数据管理服务。扩展的**多层架构**（如三层C/S）在客户端和服务器间加入中间层（应用服务器）分担业务逻辑。优点：职责分离，方便分布式部署；需考虑服务器并发和安全。变体包括**微服务架构**（将服务器端功能进一步划分为可独立部署的服务）。
- **管道-过滤器架构 (Pipe-and-Filter)**：将系统处理流程拆分为一系列**过滤器**（Filter，处理数据的独立组件）并通过**管道**连接，数据流经各过滤器逐步转换。类似Unix的管道命令链，每个过滤器只关注单一步处理。优点：模块独立，易扩展和重组处理流程；可以并行处理流；缺点：数据格式需统一，调试不易定位。常用于编译器（词法->语法->语义分析）或数据流转换系统。
- **事件驱动架构**：以**发布-订阅**模式为代表，组件通过事件进行松耦合通信。**消息总线/事件总线**介导通信，事件源组件发布事件，感兴趣的组件（订阅者）收到通知并触发相应动作。典型如GUI的事件处理、分布式系统的消息队列架构。优点：组件解耦，扩展订阅者方便；缺点：异步通信调试困难，系统状态跟踪复杂。

- **共享存储架构**：如**黑板模型 (Blackboard)**，一组独立的子系统通过共享的全局数据存储（黑板）协作[54]。子系统监视黑板中数据变化并做出响应决策，反复更新黑板直到求解完成。该模式用于问题求解（如AI专家系统），优点是不同算法可异步协作，缺点是共享数据成为瓶颈且一致性维护复杂。
- **模型-视图-控制器 (MVC)**：一种用于交互式应用的架构模式，将软件分成**模型**（核心数据和业务逻辑）、**视图**（用户界面）和**控制器**（视图与模型间交互协调）。视图负责显示模型数据，控制器接收用户输入并转换为对模型的操作，模型更新后通知视图刷新。MVC实现了表示层和业务逻辑的分离，易于维护和扩展不同视图。Web开发中常见MVC框架将请求路由到控制器，控制器更新模型并选择视图输出响应。
  （除上述外，还有面向服务架构(SOA)强调通过服务接口提供松耦合服务，以及对等网络(P2P)架构等，视课程要求掌握。）
  考试要点：能够列举几种架构风格并描述其特点和示例场景，比较不同架构风格的适用性和优缺点，例如分层 vs 客户端服务器、事件驱动 vs 管道过滤的区别等。

设计模式

- **创建型模式**（共5种）：解决对象**创建**相关问题，提供灵活的创建机制以解耦对象创建与使用。主要包括：
- **工厂方法模式**：定义一个创建对象的接口，由子类决定实例化哪种具体产品类[55]。客户端仅通过工厂接口获取对象，不需了解具体构造逻辑，方便扩展新的产品类型。
- **抽象工厂模式**：提供一个接口以创建**一系列相关联**的对象族（产品族），无需指定具体类。例如UI工具库的抽象工厂可生成一整套按钮、文本框等不同风格的组件。保证产品族的一致性，易于切换产品族。
- **单例模式**：确保一个类在应用中**只有一个实例**，并提供全局访问点[56]。通过将构造函数私有化、提供静态方法返回唯一实例实现。常用于全局管理类（如配置管理、线程池）。注意懒汉/饿汉式实现及线程安全问题。
- **建造者模式**（生成器）：将一个复杂对象的构造与其表示分离，通过逐步构建的方法来创建对象。建造者按步骤构建产品，并允许不同的建造者创建不同表示。适用于需要按步骤构造、组合部件的对象（如构建器生成不同配置的车辆对象）。
- **原型模式**：通过拷贝已有实例来创建新对象，而不是通过构造函数。先定义一个原型接口，包含克隆自身的方法。当创建新对象开销大或需避开构造过程时，可利用原型快速复制。注意深拷贝与浅拷贝的实现。

创建型模式强调封装实例化过程，减少依赖。考试可能要求辨识工厂方法 vs 抽象工厂区别，单例的应用场景等。 - **结构型模式**（共7种）：解决**类或对象组合**构成更大结构的问题，通过继承或组合来实现模块间的灵活组织：
- **适配器模式**：将一个类的接口转换为客户端期望的另一接口，弥合接口不兼容的问题。分对象适配器（组合）和类适配器（多重继承）两种实现。典型应用是旧系统接口适配新系统，使得原本不能一起工作的类可以协同[57]。
- **桥接模式**：将抽象部分与实现部分分离，使它们可以独立变化。通过组合关系桥接，抽象类包含实现接口的引用。适用于跨多个维度变化的类，如形状和颜色，通过桥接可独立扩展形状种类和颜色种类，而不产生笛卡尔积子类。
- **组合模式**：将对象组织成树形的**部分-整体**层次结构，使客户端对单个对象和组合对象的使用具有一致性。组件可以包含子组件（递归定义）。典型如文件系统目录-文件结构，组合模式允许统一处理叶子和容器。实现上常有抽象组件接口，叶节点和复合节点实现接口，复合节点存储子节点。
- **装饰模式**：动态地给对象添加新的职责，而不改变其接口。通过创建装饰类包装真实对象，装饰类实现与真实对象相同接口，并持有真实对象引用，在调用时增添额外行为。可以叠加多个装饰。例子：给流对象添加缓冲、加密等功能。相对于继承更灵活，避免子类爆炸。
- **外观模式**：提供一个高级接口，封装子系统的一组接口，从而简化客户端使用。外观类对内协调调用多个组件，向外提供统一功能调用接口。常用于复杂子系统（如编译器子系统），客户端通过外观完成常见任务，无需了解内部细节。
- **享元模式**：运用共享技术复用大量细粒度对象，避免重复开销。将对象分为内部状态（可共享）与外部状态（不共享，由客户端维护）。享元工厂负责维护池，当需要对象时返回已有实例或创建新实例。典型应用是字

符对象在文档中的重复出现，通过共享减少内存。
- **代理模式**：为某对象提供一个**代理对象**，由代理控制对真实对象的访问。代理可以在调用真实对象前后附加行为，例如远程代理（负责网络通信）、虚拟代理（延迟加载实际对象）、保护代理（控制访问权限）等。代理模式保持了接口一致性，客户端感觉就像在直接使用真实对象。

结构型模式考点：适配器vs桥接的区别（适配器改接口、桥接分离抽象实现）、装饰 vs 代理区别（装饰添加功能、代理控制访问）、组合模式的树形结构等。 - **行为型模式**（共11种）：关注**算法和对象职责的分配**，以及对象之间的交互通信：
- **模板方法模式**：在抽象类中定义算法骨架，将某些步骤的实现延迟到子类。父类模板方法按固定流程调用基本方法，子类可重写这些基本方法以定制步骤细节。保证算法整体结构稳定，又允许细节变化。常用于算法由多步组成且部分步骤易变的情况（如排序算法模板，具体比较策略由子类实现）。
- **策略模式**：定义一系列可互换的算法，将每个算法封装到独立的策略类中，使它们可以在运行时替换。客户端持有抽象策略引用，根据需要选择不同具体策略实现。典型例子是排序策略、缓存策略等。策略模式避免了在客户端使用大量条件语句选择算法，提高灵活性。
- **观察者模式**：又称发布-订阅模式，定义对象间一对多依赖，当**主题对象**状态发生变化时，自动通知所有依赖的**观察者对象**，使它们能够更新自己 [58] 。主题提供注册/移除观察者的接口，维护观察者列表，触发变化时调用观察者接口方法 [59] [60] 。典型应用：GUI事件处理（按钮为主题，监听器为观察者）或模型-视图更新（模型变化通知视图刷新）。该模式松耦合地连接消息发布者和订阅者。
- **状态模式**：允许对象在内部状态改变时改变其行为，看起来像更改了其类。通过将状态封装为独立的状态类，并由状态子类实现不同状态下的行为。环境（Context）对象将行为请求委托给其当前状态对象来执行。避免了大量if/else或switch根据状态执行不同动作，使状态转换和动作封装在状态类内部，增加新状态更方便。
- **职责链模式**：将请求沿着一条**处理者链**传递，直到有处理者处理它。每个处理者包含对下一个处理者的引用，请求在链上传递时每个处理者有机会处理或传递下去。典型如事件冒泡或拦截机制（多个对象按顺序尝试处理事件），或者请假审批流程逐级上报。优点：请求发送者无需知道最终处理者是哪一个，实现松耦合和灵活的职责分配。
- **命令模式**：将请求封装为命令对象，包含执行该请求所需的全部信息（接收者对象和操作细节）。命令对象提供执行方法 `execute()` ，调用接收者实现具体操作。可将命令排队、记录日志、支持撤销操作等。典型如GUI中菜单/按钮的点击操作封装为命令类，可在不同时间执行或撤销。命令模式将行为请求者与执行者解耦，增加灵活性。
- **中介者模式**：引入一个中介对象（Mediator）封装对象之间的复杂交互。各对象（同事）不再直接互相引用，而是通过中介者发送消息。中介者负责协调并调度同事间的交互逻辑。常用于聊天室（中介者为聊天室服务器，同事为用户）、GUI窗体控件交互等。优点：简化了对象关系网络，减少耦合；缺点：中介者本身可能变得复杂。
- **备忘录模式**：在不暴露对象内部细节的情况下，保存对象某一时刻的状态（快照）以便日后恢复。通过备忘录类存储状态，发起人（Originator）创建备忘录，负责人（Caretaker）持有备忘录以在需要时恢复。常用于实现撤销操作，将历史状态保存起来。注意控制备忘录的生命周期和开销。
- **访问者模式**：将作用于对象结构的操作封装在访问者对象中，使得在不修改元素类的前提下增加新操作。对象结构包含很多元素类，各元素接受访问者到访并调用其针对该元素类的处理方法。这样新增加操作只需新增访问者子类即可。适用于元素种类固定、新操作频繁的情况（如编译器AST操作）。缺点是增加新元素类型比较困难，需要修改所有访问者。
- **解释器模式**：为特定领域语言定义文法表示，并设计解释器解释该语言的句子。使用抽象语法树表示语言的语法结构，解释器递归地解释节点。典型用于简单命令语言、正则表达式等的实现。由于应用场景特殊，一般在需要构建DSL时考虑，用途相对较少。

（以上行为型模式众多，复习时重点掌握常用的几个：观察者、策略、模板方法、状态、命令等，理解它们解决的问题和基本类图结构。考试可能让识别某段场景描述适用的模式，或画出模式类图。）

敏捷开发核心思想与流程

- **敏捷宣言与理念**：敏捷开发是一种**适应变化、快速交付**的软件开发思想。2001年发布的**敏捷宣言**阐述了核心价值观 61：“**个体和交互**高于流程和工具，**工作的软件**高于详尽的文档，**客户合作**高于合同谈判，**响应变化**高于遵循计划”。也就是说，更重视人的协作、交付真正可用的软件、客户的参与反馈，以及对需求变化的积极响应，而不是僵化地遵守既定计划。敏捷提倡**迭代增量**开发：将项目划分为多个小的迭代周期，每次迭代都产出可工作的软件增量 62。通过持续交付价值，及早收集反馈来改进产品。敏捷原则（12条）强调如：以**满足客户**为最高目标，欢迎需求晚变，频繁交付可用软件，业务人员与开发者**每日合作**，建立可信环境并面谈沟通，工作的软件是进度主要度量，保持可持续开发速度，不断关注技术卓越和简洁设计，团队定期反省调整 63。这些理念指导团队更灵活高效地开发，适应不确定和变化。考试常要求理解敏捷价值观的内涵，例如“响应变化胜过遵循计划”意味着什么。
- **敏捷开发流程（Scrum为例）**：敏捷并非无序，而是通过简化的过程管理来快速迭代。以**Scrum框架**为代表的流程：Scrum将开发分成短**冲刺（Sprint）**，每个冲刺通常2～4周 64。在每次冲刺中，团队完成**小型的瀑布循环**（需求分析、设计、编码、测试等）并交付一个可用的产品增量 65。Scrum角色包括**产品负责人**（Product Owner，负责需求优先级和验收标准）、**Scrum主管**（Scrum Master，移除障碍、确保团队按敏捷原则运作 66）和**团队成员**（跨职能的开发团队）。主要工件有**产品待办列表**（Product Backlog，按优先级排列的需求条目）和**迭代待办**（Sprint Backlog，本次冲刺要完成的任务）。关键活动有：冲刺计划会（确定本迭代目标和待办项）、**每日站会**（每日15分钟团队同步进度和解决障碍 67）、冲刺评审（演示增量给客户并收集反馈 68）、冲刺回顾（团队反思改进流程）。另一常见敏捷方法**XP（极限编程）**强调工程实践，如**结对编程**（两人一组共同编写同一代码，提高代码质量和知识共享）、**持续集成**（前述CI实践，每日集成代码并运行测试 69）、**重构**（不断改进代码设计以优化质量 70）、**简单设计**（YAGNI原则，不做超前设计）等。Scrum偏重项目管理框架，而XP偏重具体编码实践 71。敏捷团队通过短周期迭代和持续反馈，实现需求的快速演进和高质量交付。
  常考点：传统瀑布模型 vs 敏捷开发区别（敏捷迭代、拥抱变化、客户参与等优势），Scrum流程各要素的作用，每日站会三个问题（昨日完成了什么、今天计划做什么、有何障碍）等细节，以及如何应对需求变化。敏捷核心在于**快速响应和持续改进**，理解这些有助于应对相关考题。

---

1 2 3 4 5 6 7 8 9 10 2025-4-需求获取技术与需求工程.pdf
file://file_0000000038e87206b424852574446f82

11 12 13 14 15 16 57 UML.pdf
file://file_00000000ae307206a3471c344e0bfc2d

17 18 19 20 21 22 25 26 27 37 38 软件测试基础.pdf
file://file_000000005f687206ab6c5d764f08d087

23 24 31 32 33 34 35 36 39 40 41 软件系统测试.pdf
file://file_000000002f9872068f297cb89137a65e

28 29 30 软件黑盒测试.pdf
file://file_00000000ab047206adff9506f34f617a

42 43 44 45 46 47 48 49 50 51 52 53 2025-10-软件设计.pdf
file://file_000000009e54720683ae8a0359219b0e

54 2025-11-软件体系结构.pdf
file://file_00000000db14720698a8a8efd74f975d

55 56 58 59 60 2025-12-设计模式.pdf
file://file_0000000027e07206bd2b3bf2a6703b2e

61 62 63 64 65 66 67 68 69 70 71 2025-2-敏捷开发方法.pdf
file://file_00000000ccdc7206a1f1ed3fb4cf88e3