1. 尽管敏捷开发强调“可工作的软件高于详尽的文档”和“响应变化高于遵循计划”，但这绝不意味着完全抛弃文档、设计和计划。相反，敏捷开发以一种更精益、更适应变化的方式来处理它们。敏捷开发鼓励“恰到好处”的文档。它不追求过度的、僵化的文档，而是关注能为团队和客户带来价值的文档。用户故事 (User Stories)： 这是敏捷中常见的“文档”形式，简短、清晰地描述了用户需求。它们是需求文档的替代品，但更侧重于对话和协作。测试用例：在敏捷中，测试是持续进行的，测试用例本身也是一种重要的“文档”，用于确保质量和验证功能。产品待办列表 (Product Backlog) 和冲刺待办列表 (Sprint Backlog)： 这些列表是经过优先级排序的需求和任务的“计划文档”，用于指导开发。代码注释和自文档化代码： 敏捷推崇清晰、易读的代码，代码本身就是最好的文档。提倡“持续设计”或“演进式设计”，而不是瀑布模型中一次性完成所有设计。设计是随着项目的进展不断演进和细化的。TDD (测试驱动开发) 和 BDD (行为驱动开发)： 这些实践在编写代码之前定义行为和测试，实际上也是一种“设计”的过程，引导开发者思考代码的接口和行为。结对编程 和代码审查 ： 这些协作实践有助于团队成员共同探讨和改进设计。计划：每日站会：团队每天都会进行简短的同步，实际上也是一种微观的、实时的“计划”调整和问题识别。迭代回顾 (Sprint Retrospective)： 在每个迭代结束时，团队会反思并规划如何改进下一次迭代的工作方式。//完全没有文档会导致知识流失、新成员难以融入、维护困难，以及与利益相关者沟通障碍。没有设计，软件将变得杂乱无章、难以扩展和维护，最终导致技术债务缠身。没有计划，项目将失去方向，无法预测交付，也无法有效分配资源。

2.软件维护：在软件生存期的各个阶段都应该高度重视并采取有效措施来提高软件的可维护性。需求分析阶段：清晰、完整、无歧义的需求规格说明：明确的需求是设计出可维护系统的基础。模糊的需求会导致频繁的修改和错误的理解，增加维护难度。非功能性需求（如可维护性、可扩展性、性能等）的明确： 在需求阶段就将可维护性作为一项重要的非功能性需求提出，并进行量化或明确描述。业务领域知识的充分获取与建模：利益相关者沟通。设计阶段：模块化和解耦设计： 采用高内聚、低耦合的设计原则。将系统分解为独立的功能模块，减少模块间的依赖，使得修改一个模块时对其他模块的影响最小化。架构设计评审期进行架构评审，确保设计符合可维护性要求，并预留扩展性接口。设计模式的应用恰当应用设计模式（如工厂模式、观察者模式、策略模式等）可以提供成熟的解决方案，使系统结构更清晰、更易于理解和修改。编码阶段：清晰的注释和文档：对复杂逻辑、接口、算法等进行必要的注释，尤其是公共API的文档，便于其他开发者理解和使用。代码规范：遵循统一的编码规范（命名规范、注释规范、代码格式等），提高代码的可读性。文档化代码：编写清晰、意图明确的代码，让代码本身就能解释其功能。使用有意义的变量名、函数名和类名。测试阶段缺陷跟踪与管理：建立有效的缺陷管理流程，确保缺陷被及时发现、记录、修复和验证。可维护性测试：可以通过工具或人工评审来评估代码的复杂度、耦合度等指标，间接反映可维护性。编写全面的自动化测试用例。测试是维护的基础，有了充分的测试，在修改代码时才能更有信心，减少引入新错误的风险。错误处理和日志记录： 完善的错误处理机制和清晰的日志记录，有助于快速定位和诊断问题。部署与运行维护阶段：监控和日志分析： 建立完善的系统监控和日志分析机制，及时发现和预警潜在问题，便于快速响应和修复。版本控制：使用版本控制系统（如Git）管理所有代码、文档和配置，便于追溯历史、协同开发和回滚。配置管理：对软件的运行环境、依赖库、参数等进行统一管理，确保环境的一致性。

3.科学地进行软件项目计划(非确定性 计划赶不上变化）采用迭代和增量式规划采纳敏捷思想，将项目计划分解为多个短周期的迭代（如Sprint）。 高层路线图 (Roadmap)： 制定一个相对稳定的、高层次的、长期的产品愿景和主要里程碑，但不包含具体细节。这为团队提供了方向感。迭代计划 (Sprint Planning)： 每个迭代开始前，团队共同评估待办事项，选择能在当前迭代中完成的功能，并分解成具体的任务。这是最详细的计划，但只针对短周期。基于风险的规划：在计划初期就识别、评估和规划应对风险的策略。强调团队协作和透明度 以价值为导向的优先级排序：产品待办列表 (Product Backlog)： 维护一个所有需求（用户故事）的列表，并根据业务价值、风险、依赖性等因素进行优先级排序。MoSCoW 方法将需求分为Must-have, Should-have, Could-have, Won't-have，确保核心功能得到优先开发。采用迭代式估算和持续调整速率在几个迭代后，根据团队实际完成的故事点数计算出团队的平均速率，以此作为未来迭代计划的依据。这是一种经验驱动的估算。持续跟踪与调整： 每日站会 跟踪进度，发现偏差及时调整。在迭代回顾 中总结经验，改进估算和计划流程。强调团队协作和透明度共同参与让所有团队成员（包括开发、测试、产品）共同参与计划制定过程，特别是迭代计划。可视化管理使用看板或Scrum板等工具，将任务进度、状态可视化，让所有人都清楚项目的当前状况。

5.螺旋模型主架构与敏捷开发思想螺旋模型将瀑布模型的系统性和迭代模型的灵活性结合起来，尤其适合于需求模糊、大型、复杂、高风险的项目开发。它强调在项目的每个迭代周期中进行风险评估和风险规避。强调风险驱动：螺旋模型在每个迭代阶段都强制进行风险识别、分析和管理，能够有效降低项目风险。适应需求变化螺旋模型以迭代的方式进行开发，每个迭代结束都可以根据反馈和风险评估调整下一阶段的计划，这使得它比传统的瀑布模型更具灵活性，能更好地适应需求变化。支持原型开发通过不断的原型验证，可以及早发现设计缺陷、优化交互，并验证硬件与软件的集成效果。例如，可以先开发一个（最小可行产品），让少数用户体验，收集反馈。结合敏捷方法实践虽然选择了螺旋模型作为整体框架，但在每个螺旋周期内部的开发阶段，可以深度融入敏捷开发的方法和实践如：Scrum或Kanban小周期迭代： 螺旋模型的一个“圈”可能较大，但每个圈内的开发、测试阶段可以进一步细化为Scrum的短迭代（Sprint）。持续集成与交付强调代码的持续集成和测试，确保软件质量。选择螺旋模型作为主体框架能够解决高风险和不确定性，在螺旋模型的每个迭代内部融入敏捷实践，能够提高开发效率、增强团队协作，并快速响应变化。这种组合策略既能保持项目整体的风险控制和规划，又能保证开发过程的灵活性和高效性。

6.关于新兴技术这些工具集在很大程度上提升了软件项目的成功率，并在一定程度上缓解了（而非彻底消除）软件危机。关于软件危机本质复杂性： 指软件本身固有的、难以消除的复杂性，如需求理解的困难、并发处理、通信协议、数据结构等。这种复杂性是软件固有的，无法通过工具或方法论简单消除。偶然复杂性： 指由于工具、编程语言、开发环境等非本质因素带来的复杂性。例如，早期汇编语言的低效、糟糕的调试工具、不成熟的操作系统等。新工具带来新挑战云计算、微服务等虽然解决了旧问题，但也引入了新的复杂性（如分布式系统协调、数据一致性、服务治理、复杂的运维监控）。AI工具也需要人类的指导和验证。软件的本质复杂性，即理解人类需求、逻辑抽象和精确表达的困难，并未被根本解决。人与人之间的沟通障碍、需求模糊性依然是软件项目失败的重要原因。最终，软件的成功与否，很大程度上取决于开发团队的技能、经验、沟通和管理能力。再好的工具和方法，也需要人去正确地理解和运用。

7.版本交付规范严格，强调过程控制、审计、可追溯性在需求明确的情况下选择V型模型V强调验证与确认过程。如果在验证和确认期间发现问题，可以重新执行V左侧，以在重新执行右侧的 测试步骤之前修复和改进需求、设计和代码。

8.团队协作按模块分工的方式组织团队成员，即每位成员负责一个相对独立的子模块，如前端界面、后端接口、数据库设计等。提高效率，便于并行开发；挑：过早且严格地固定角色有时反而增加了跨部门沟通的复杂性，导致信息传递失真或效率低下。如果初期模块接口定义不清，后期可能导致集成困难。合作方式的体会线上线下相结合的沟通机制避免了碎片化沟通带来的遗漏和误解。版本控制工具：分支管理代码、提交规范、定期进行代码审查。团队合作过程中最重要的是沟通机制的建立。明确“谁负责什么、截止日期、对接人”极其关键。项目早期沟通不足导致部分成员理解方向偏差，后期需要花时间统一思路，影响进度。

9.关于风险项目风险 影响项目计划、进度、成本、资源和人员的风险。技术风险影响软件质量和交付性能的风险，如技术难题、兼容性问题、性能瓶颈、架构不合理等。业务风险影响项目商业价值或市场接受度的风险，如需求不明确、市场变化、用户接受度低等。人员风险团队成员流失、技能不足、沟通不畅等。外部风险 不可控的外部因素，如政策变化、竞争对手、硬件故障等。例需求频繁变更（业务项目）建立严格的需求变更控制流程，需求优先级排序与版本规划、原型迭代与用户确认，将需求分解为更小的、可管理的部分，并在每个迭代结束后进行交付和评审，以便在早期发现并控制需求变更。技术学习曲线陡峭内部技术分享与学习、代码审查与结对编程、寻求外部资源

10.这里说明一下不是所有高风险的都是要去选择螺旋模型，螺旋模型所解决的风险更倾向于需求不确定、技术不确定性带来的风险。例如医疗设备，它需求较为完善，并且它需要更为严格的监管和认证要求。更需要的是环节的追溯性所以这里选择V模型。我觉得是更为合理的。其结构化、阶段性的特性使得文档和可追溯性管理变得更为容易和规范。在螺旋模型中，通过原型和迭代发现问题并“返工”是其设计的一部分。但对于医疗设备代价是巨大的，甚至可能危及生命。V模型的目标是尽早且系统性地发现和消除缺陷，尽量避免大规模的后期返工。

11.然后最后，这里说一些作图的看法。Emm问了一下老师像结构化分析的那些数据流图都是不会考的.然后对于UML 在项目里我给出了几个我认为比较重要的图。然后它们具体会在哪里用？我们可以结合软件生存周期去看一下。首先是需求分析：需求获取（用例分析）（快速原型）需求分析与细化（用例图、类图、活动图、顺序图）需求优先级排序（Kano 模型： 区分基本型需求、期望型需求、兴奋型需求、无差异型需求和反向型需求）需求验证与确认（原型验证，需求可追溯性： 建立需求与设计、开发、测试之间的关联）需求文档化然后需求管理就是之前提到的规定更改范围和对新的需求尽心分析是否加入。然后软件设计这里是分概要和详细的。一般的设计题目都会说对于概要的类的设计（重要的是类之间的关系UML类图）详细的设计会让类的属性和方法写出来。下面概要设计明确系统架构（这个比较重要）确定模块间关系识别主要接口（组件图）评估系统性能和安全性： 从宏观层面考虑系统的性能指标（如响应时间、吞吐量）和安全策略。关键技术选型与非功能性需求考虑（在架构层面考虑如何满足这些非功能性需求：扩展呀维护呀之类的）详细设计：明确模块内部逻辑（定义每个模块内部的功能、算法、数据结构和处理流程）详细设计是软件从抽象概念走向具体实现的关键一步。它要求设计者具备扎实的编程功底和对业务逻辑的深入理解。活动图,状态图，顺序图。软件维护问题（完善性维护，适应性维护，纠错性维护，预防性维护）估算与计划、风险管理和质量保证 估算是预测项目完成所需的资源（人力、时间、成本）和工作量。准确的估算对于制定合理的计划至关重要（类比估算：基于类似历史项目的经验数据进行估算。适用于项目相似度高的情况。参数估算：使用历史数据和统计关系来计算估算值|例如，每行代码多少工时，每功能点多少工时|需要有量化指标。自下而上估算 ： 将项目分解为最小可管理任务，对每个小任务单独估算，然后汇总得到总估算。最精确但最耗时。计划:计划是在估算的基础上，将项目目标分解为具体任务，并安排这些任务的执行顺序、时间表、资源分配和责任人。