软件工程期末复习

一般产品：功能、质量、结构

通用工程： 收益-风险；过程-结果；少数-多数

- **风险 vs 收益**
  - 风险可能包括：
    1. 时间延误：开发人员可能会遇到技术难题、人员调整和其他意外情况，导致开发进度延误。
    2. 质量问题：如果开发人员没有遵循最佳实践或者没有进行充分的测试，可能会导致软件存在漏洞或者其他质量问题。
    3. 范围膨胀：如果客户或者其他利益相关者在开发过程中提出了新的需求或者变更，可能会导致项目范围超出原先的计划。
  - 收益可能包括：
    1. 更高的质量：如果开发人员遵循最佳实践并进行充分的测试，可能会产生更高质量的软件。
    2. 更快的开发速度：如果开发人员使用了合适的工具和方法，可能会使开发速度更快。
    3. 更好的用户体验：如果开发人员重视
- **过程 vs 结果**
  - 过程是指在软件开发生命周期中采用的一系列活动、方法、工具和技术，它们可以帮助开发团队规划、设计、实现、测试和交付软件
  - 结果是指软件开发过程中的最终软件产品，包括软件的功能、质量、性能和可靠性等特性
- **少数 vs 多数**
  - 少数是指软件开发团队中的一小部分成员，例如技术领导者、高级开发人员和质量保证人员等，他们通常负责制定技术策略、指导开发过程和保障软件质量等方面。
  - 多数是指软件开发团队中的大多数成员，例如开发人员、测试人员和项目管理人员等，他们通常负责具体的开发、测试和管理工作，通过协作完成软件开发任务。

工程师的特点：

- 人道无害
- 雇主
- 实事求是，恪守公心，严守纪律，蓄意学习

软件特点：复杂性；商品属性；可变性；*功能的契合性

软件家族：版本序列；产品线；产品家族；成品组件

软件的功能

- 包括基本功能、高级功能和定制功能等
- 开发人员需要根据产品需求和设计要求进行开发和测试，以确保软件的功能和特性能够满足业务需求和用户需求。

软件的质量

1. 用户反馈：软件的质量需要通过用户反馈进行评估，包括用户满意度、用户体验等。
2. 软件的自适应：软件需要具备自适应性，即能够根据用户的需求和环境变化进行调整和优化。
3. 移植性：软件需要具备良好的移植性，即能够在不同的平台和操作系统上运行，并且能够兼容各种硬件和软件环境。
4. 功能稳定性：软件的各种功能需要保证稳定性，即能够在各种情况下正常运行，并且不会出现崩溃和异常情况。
5. 性能效率：软件的性能效率需要保证，即能够在各种数据量和负载情况下保持良好的性能表现。

银弹："银弹"是软件工程领域中的一个术语，指的是一种能够解决所有问题的万能解决方案。然而，在软件工程领域中，没有一个单一的银弹能够解决所有问题。

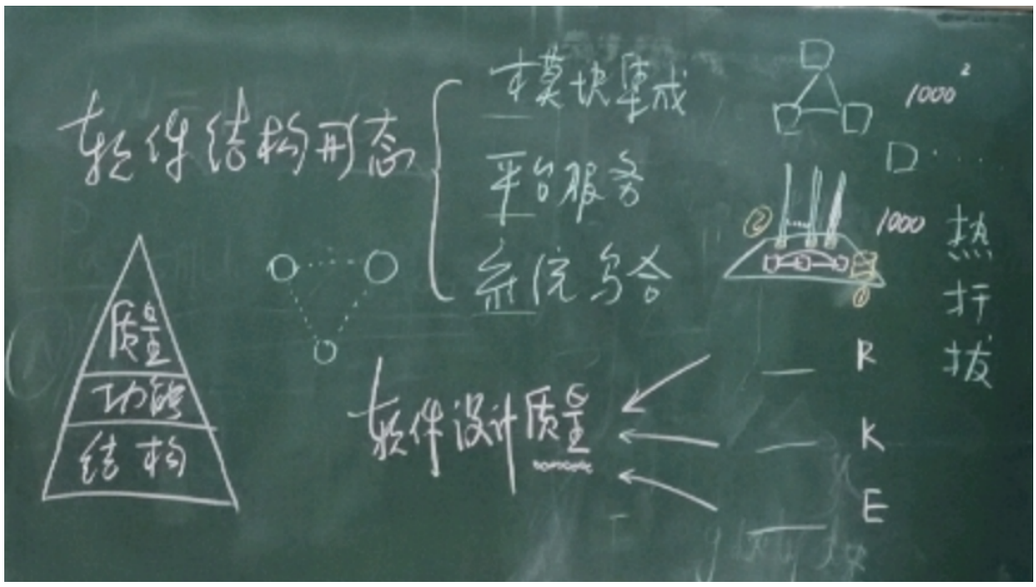
软件工程是一个复杂的领域，需要综合运用多个技术和方法来解决问题。例如，软件开发过程中需要使用多种编程语言、开发工具和技术，如需求分析、设计、编码、测试、部署等。此外，软件开发还需要考虑到项目管理、团队协作、文档管理、版本控制等方面的问题。

- 没有银弹的核心是软工的复杂，不过程更重要的是人

软件天花板：在软件工程中，规模天花板是指在软件开发过程中，由于某些技术或方法的限制，导致软件规模无法继续扩大的情况。

> 软件工程的规模天花板可以来自多个方面，例如：
>
> 1. 技术限制：某些编程语言或开发工具可能无法处理大规模的软件项目，或者在大规模项目中存在性能瓶颈，导致开发效率降低。
> 2. 人力资源限制：大规模的软件项目需要大量的开发人员和项目管理人员，但是人力资源有限，招聘和培训人员需要耗费大量时间和成本。
> 3. 需求管理限制：在大规模软件项目中，需求管理变得更加复杂和困难，需求的变更和管理需要更多的人力和时间成本。
> 4. 组织管理限制：在大规模软件项目中，需要更加严格的组织和管理方法，包括项目管理、人员管理、文档管理等，这需要更多的人力和资源。
>
> **复杂度问题是导致软件工程规模天花板的一个重要原因，这是因为随着软件规模的增加，软件的复杂度也会随之增加。软件的复杂度包括系统需求的复杂度、软件架构的复杂度、代码的复杂度等等。**
>
> **当软件的复杂度达到一定程度时，开发人员的能力可能无法承受复杂度的增长，这就导致了软件规模的上限。如果继续增加软件规模，可能会导致软件开发效率的下降、质量的降低，甚至可能导致软件系统崩溃。**
>
> **例如，微软的重构和DB2代码的上限问题，正是由于软件复杂度太高，随着软件规模的增加，开发人员无法应对其复杂度的增长，导致软件开发效率和质量的下降。为了解决这些问题，需要采用一些有效的软件开发方法和技术，例如模块化设计、自动化测试、重构等方法，以降低软件的复杂度，提高软件开发效率和质量。**
>
> 为了克服规模天花板的问题，软件工程师们可以采用多种方法，例如：
>
> - **模块化设计：**将大型软件系统分解成多个模块，每个模块负责特定的功能或任务，从而降低整个系统的复杂度，提高开发效率和质量。
> - **自动化测试：**使用自动化测试工具和技术，对软件进行全面的测试，以确保软件的正确性和稳定性，减少错误和缺陷的数量，提高软件的质量。
> - **重构：**对已有的代码进行重构，从而改进软件的设计和结构，使其更加清晰、简洁和易于维护，减少代码的复杂度和冗余。
> - **使用现代化的开发工具和技术：**如使用开源软件、云计算、人工智能等技术，以提高软件开发效率和质量，同时降低开发和维护的成本。
> - **采用敏捷开发方法：**通过迭代和增量式的开发方式，及时地获取用户反馈和需求变更，降低开发风险，提高软件开发效率和质量。
> - 采用面向对象的设计和编程方法：通过面向对象的设计和编程方法，将软件系统抽象成对象，降低系统的复杂度和耦合度，提高软件的可维护性和可扩展性。
> - **培养高素质的软件开发人员：**通过培训和技术交流等方式，提高软件开发人员的技术水平和素质，使其能够更好地应对软件规模的增长和复杂度的增加。

- **天花板的核心是复杂度太高**，进而规模的上升导致复杂度更快的增长，而人类的能力无法承受复杂度的增长；
- 天花板无法通过简单的增加人手解决，因为**人月定律。**
- eg：微软的重构，向前兼容困难；DB2代码200w上限也只能重构了
- eg：1层上盖2层，20层上盖21层的困难度是不一样的
- 期望设计一个结构，使得复杂度线性增长而非超线性增长，这属于架构师的工作了
    - 如何设计架构?

- **判断哪个架构最好?**
  - 适合产品的设计
  - 可行性、可构造性
  - 可变性（持续）**【与实体工业的区别】**
  - **因为软件有如下特性**
    - **复杂性**
    - **可变性**
    - **商品属性**
- **软件三层金字塔（从上到下）：质量、功能、结构**
- 

- **软件结构形态**
  - **单体架构**
  - **模块集成**：模块化是一种常见的软件架构设计方法，它能够将系统拆分为多个独立的模块，每个模块之间的关系简单明了。这种方法适用于变更频率较低的中小型软件项目。
    - 缺点：模块化使得关系数量上升，复杂性超线性增长
    - 使用与变更频率低的中小场合
  - **（大型）平台服务【微服务】**：平台服务是一种分布式系统架构，能够实现热插拔和冗余存储，同时具备高可用和高可伸缩性。这种方法适用于大型复杂的软件系统。
    - 分布式
    - 热插拔：可以在不停止或重启整个平台服务的情况下，动态地添加、替换或卸载平台服务中的一个或多个组件。这种能力使得平台服务更加灵活和可扩展，能够在运行时对平台服务进行动态修改和优化。
    - 冗余存储
  - **系统乌合：取源于“乌合之众”**
    - def：模块之间如无必要，不要交流
    - "乌合之众"是指没有明确目标和领导的人群，他们的行动和思想容易受到外界影响。系统乌合设计方法的缺点可能包括以下几个方面：
      1. 缺乏统一的规划和控制：在系统乌合设计方法中，各个组件之间并没有明确的接口和依赖关系，因此缺乏统一的规划和控制。这可能会导致系统的稳定性和可靠性受到一定程度的影响。
      2. 不兼容和冲突问题：由于系统乌合的组件通常来自不同的开发者和组织，因此在组件之间可能存在不兼容和冲突等问题。这可能会导致系统的稳定性和可靠性受到影响。
      3. 维护成本高：系统乌合的组件来自不同的开发者和组织，因此在维护过程中需要处理不同组件之间的兼容性和冲突等问题，这可能会增加系统的维护成本。
      4. 可扩展性受限：在系统乌合设计方法中，各个组件之间缺乏明确的接口和依赖关系，因此可能会限制系统的可扩展性。
- **【关于如何设计架构的补充内容】**：

        - 需求分析：在设计软件架构之前，需要进行充分的需求分析，明确软件系统需要实现哪些功能和特性，以及用户的需求和期望。这将有助于确定软件系统的架构和组件之间的关系。需求分析可以通过与客户或业务代表的交流来实现，也可以通过文档、用户故事和用例来获取。
        - 模块化设计：将软件系统分解为多个模块，每个模块负责完成特定的功能。这种模块化设计有助于减少组件之间的复杂性，并且使得系统更易于维护和扩展。模块化设计需要考虑以下几个方面：
            - 模块之间的接口和依赖关系
            - 模块的职责和功能
            - 模块的接口和实现
        - 设计模式：使用设计模式可以提高软件系统的可重用性和可扩展性，同时还能够减少代码重复和错误。设计模式是针对特定问题的通用解决方案，可以帮助架构师在设计中遵循已有的最佳实践，从而提高软件系统的质量和性能。常用的设计模式包括单例模式、工厂模式、观察者模式等。
        - 技术选择: 选择合适的技术和工具对于软件架构的设计非常重要。需要考虑的因素包括开发语言、框架、数据库、消息队列等。在选择技术和工具时，需要考虑以下几个因素：
            - 技术和工具的成熟度和可靠性
            - 技术和工具的性能和扩展性
            - 技术和工具的成本和可用性
            - 在选择技术和工具时，需要根据具体的需求和应用场景来进行权衡和选择。
- **平台+服务会成为主流**
- 平台就是一个软件产品
- **从产品为中心 -> 服务为中心**
    - 提高开发效率
        - 每个服务负责完成特定的功能
        - 每个服务可以由不同的开发团队独立开发、测试和部署
    - 可扩展性和可维护性
    - 降低系统的复杂性，使得系统更易于维护
    - 提高系统的可用性和稳定性、
        - 每个服务都可以部署在不同的服务器上
    - 灵活
        - 可以在不同的服务之间进行组合和调用，从而实现更多样化的功能
        - 热插拔
- **未来：平台会被垄断；服务会被分散；云服务一定是主流**
    - 对服务端/云端：最上游被垄断
        - 资金
        - 时间
        - 布局
        - **功能相同，适者生存，仅留胜者**
        - eg. 微软；gpt
    - 对开发者和中小公司
        - 快速开发，省时、省钱、省力
        - 培训成本降低
        - 不同的服务可以使用相同的平台（大模型）
    - 对非cs领域
        - 省钱
    - 对程序员而言会被淘汰，我们需要：
        1. 大局观
        2. 对行业的见解和规划
        3. 领导能力

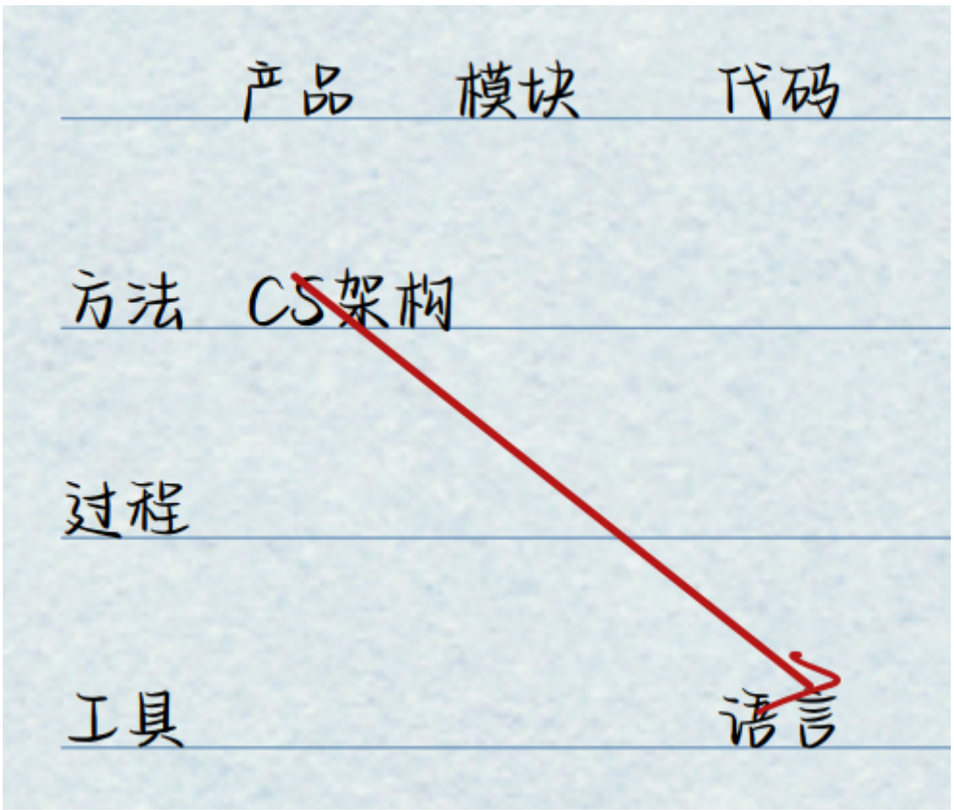
4. 冲破性的技术革新

- 分久必合，合久必分：微服务（now）和serverless（future）。
  - 微服务架构是将一个大型的应用系统分解为多个小型的服务，每个服务都可以独立部署和扩展。而Serverless架构则更进一步，将服务拆分为更小粒度的函数，不需要自己管理服务器，只需要编写和部署函数即可。
- **服务 = 设计+开发+测试+托管**
  - **平台面向领域**
    - 实现领域共性需求
    - 实现领域特定架构
    - **一个领域只需要一个平台**
  - **高度集成一体化** Devops：高度集成一体化的DevOps则是指在服务的开发、测试、部署和运营过程中，采用高度集成的工具和流程，实现快速迭代和持续交付。这种DevOps方式可以提高服务的效率和质量，同时也可以降低开发和运维的成本和复杂性。
- **（如何设计平台）开发大规模平台**
  - 知识
    - 计算机科学、软件工程、数据库管理、网络安全、云计算、人工智能等方面的知识
    - 不同技术之间的相互作用和影响，以及如何进行技术选型和架构设计等方面的知识
  - 经验：架构设计、开发流程、测试流程、部署流程、运维流程等方面的经验
  - 领域需求：在金融领域开发大规模平台需要了解金融行业的法律法规、金融产品、风险控制等方面的需求，同时也需要了解金融技术的最新趋势和发展方向

- **设计审查-设计质量（贯穿整个软件生命周期）-架构质量**
- **架构**
  - 架构分为**抽象架构和具象架构**
  - **跨模块型缺陷MCD**
    - 跨模块型缺陷（Modular Cross Defect，MCD）是指由于不同模块之间的交互和依赖关系导致的缺陷。当一个软件系统由多个模块组成时，这些模块之间存在着复杂的交互和依赖关系，这些关系可能会导致跨模块型缺陷的出现。

      跨模块型缺陷通常表现为模块之间的错误交互、接口不兼容、数据传递错误等问题。这些问题可能会导致软件系统的错误行为、性能下降、安全漏洞等问题，影响软件系统的可靠性和可维护性。
    - 解决：代码审查；测试验证；接口设计和模块化设计
  - **架构质量：**
    - 经验视角
    - 主观视角
    - 客观视角（结构；缺陷）
  - **架构质量评估：**
    - 从**经验视角、主观视角和客观视角**来评估软件系统的架构质量
    - 编码之前
      - 设计原理：模块化、高内聚低耦合、单一职责
      - 设计共识：设计模式、架构模式、UML建模
      - 在这个阶段，可以通过评审、审查等方式来评估系统的架构质量，确保软件系统的设计符合最佳实践和标准。
    - 演化之中
      - 结构视角：结构偏差；评估软件系统的结构是否符合设计原则和设计共识，以及模块之间的耦合和内聚关系是否合理
      - 缺陷视角：评估软件系统中存在的缺陷和问题，例如跨模块型缺陷（MCD）
  - 软件开发过程：设计->架构
  - **架构中心开发方法：**

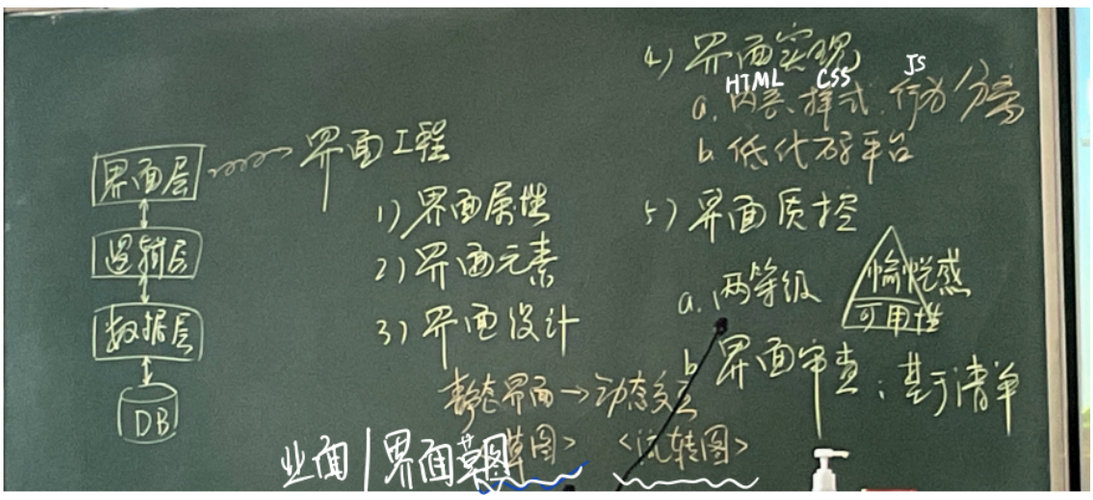
        - 业务驱动
        - 复用优先
- **设计**
    - **软件设计质量：K、R、E**
        - K、E、R是软件设计中的三个重要方面，分别代表着Knowledge、Efficiency、Reliability。
        - Knowledge（知识）：软件设计人员需要具备足够的技术知识，包括编程语言、开发工具、算法、数据结构等方面的知识。只有掌握了这些知识，才能够设计出高质量、高效率的软件系统。
        - Efficiency（效率）：软件设计人员需要在满足软件系统需求的前提下，尽可能地提高软件系统的性能和效率。在软件设计过程中，需要考虑如何优化系统架构、算法、数据结构等方面，以提高软件系统的性能和效率。
        - Reliability（可靠性）：软件设计人员需要确保软件系统的可靠性，即软件系统能够在预期的情况下稳定运行，并且能够处理各种异常情况。在软件设计过程中，需要充分考虑软件的安全性、容错性、可维护性等方面，以保证软件系统的质量。
    - 设计于维护
        - 维护的概念
            - 维护是指在软件、硬件或其他系统运行期间，对其进行修复、更新和改进的过程。维护可以分为预防性维护和修复性维护两种类型。预防性维护是指在问题发生之前采取的措施，以防止问题的出现。而修复性维护则是指在问题已经出现之后，对其进行修复和改进。
        - 维护实践（独立维护）
        - **设计从何而来：逆向工程**
            - 逆向工程是指通过分析已有的产品或系统，来了解其设计和实现的过程。通过逆向工程，维护人员可以了解一个产品或系统的内部结构和工作原理，从而可以二次开发或进行维护。
        - **设计如何使用：CIA**
            - **保密性（Confidentiality）、完整性（Integrity）和可用性（Availability）。**保密性是指确保信息只能被授权的人员访问，完整性是指确保信息在传输、存储和处理过程中不被篡改，可用性是指确保信息在需要时能够被及时访问。设计人员应该考虑这些因素，并在设计过程中采取相应的措施来确保系统的安全性和稳定性。
    - 设计于演化
        - **演化概念**
            - 演化是指事物或系统随着时间的推移而不断变化和发展的过程。在软件开发中，演化通常指软件系统的变化和发展，包括添加新功能、修复错误、优化性能等。
        - **演化实践；演化开发**
            - 演化实践是指在软件开发过程中采用一系列技术和方法，以应对软件系统的不断变化和发展。演化开发是一种敏捷开发的方法，强调快速响应变化和需求，并通过小规模、频繁的迭代来逐步构建和改进系统。演化开发的核心理念是“简单即优雅”，即通过保持代码的简单和清晰，来使系统更容易理解、维护和扩展。
        - **架构恶化**
            - 软件系统的架构随着时间的推移而变得越来越复杂、混乱和难以维护的过程。架构恶化通常发生在软件系统经历了多次演化和变更后，由于缺乏规划和设计，导致系统架构逐渐失去清晰性和稳定性
            - 为了避免架构恶化，设计人员需要在软件系统的演化过程中，时刻保持对系统的规划和设计，并采用合适的架构模式和技术来确保系统的稳定性和可维护性。
- **设计审查需要考虑的点：**
    - **设计审查清单**
        - **Y轴：方法、过程、工具**
        - **X轴：产品、模式、代码**
    - 交互设计审查清单

- 

- 设计审查是基于清单的自查自纠
  - 方法+产品
    是否采用了用户故事或用例分析方法来明确产品需求?
    是否有明确的产品设计文档和标准?
    是否考虑了用户体验和可用性等方面的要求?
    过程+模式
    是否有明确的设计过程和阶段，如需求分析、设计、实现和测试等?
    是否采用了合适的设计模式和架构模式，如MVC、MVVM、RESTful等?
    是否有明确的设计文档和标准，如设计规范、接口文档等?
    工具+代码
    是否使用了合适的编程工具和集成开发环境?
    是否有有效的代码管理和版本控制机制，如Git、SVN等?
    是否符合编码规范和标准，如命名规范、代码注释等?
    是否具有清晰的代码结构和逻辑，如模块化、抽象化等?
    过程+产品
    是否有明确的设计过程和阶段，如需求分析、设计、实现和测试等?
    是否有明确的产品设计文档和标准?
    是否有有效的团队协作和沟通机制?
    是否考虑了用户体验和可用性等方面的要求?
    工具+模式
    是否使用了合适的编程工具和集成开发环境?
    是否有有效的代码管理和版本控制机制，如Git、SVN等?
    是否采用了合适的设计模式和架构模式，如MVC、MVVM、RESTful等?
    是否有合适的自动化构建和测试工具，如Jenkins、Travis CI等?
    方法+代码
    是否采用了合适的设计方法和流程，如敏捷开发、迭代开发等?
    是否符合编码规范和标准，如命名规范、代码注释等?
    是否具有清晰的代码结构和逻辑，如模块化、抽象化等?
    是否易于维护、调试和测试?
- 软件系统的质量控制：审查+测试
- > 设计审查（Design Review）是指对软件系统的设计进行全面评估和审查的过程，旨在发现和纠正设计中存在的问题，提高软件系统的设计质量和架构质量。设计审查可以采用不同的方法、过程和工具，包括以下几个方面：

> 1. 人工审查：由专业的设计人员或团队进行人工审查，通过讨论、分析和评估等方式发现和纠正设计中可能存在的问题和缺陷。
> 2. 自动化工具：利用自动化工具进行代码分析、模型检查等方式，发现和纠正设计中可能存在的问题和缺陷。
> 3. 设计评估模式：采用不同的设计评估模式，如面向对象设计原则、设计模式、UML建模等，评估设计的质量和架构的合理性。
>
> 设计审查的对象可以是产品、模式和代码等，具体包括以下几个方面：
>
> 1. 产品审查：对软件产品的设计进行审查，评估产品的功能、性能、可扩展性、可维护性和可重用性等特点。
> 2. 模式审查：对软件系统采用的设计模式进行审查，评估模式的合理性、适用性和可维护性等特点。
> 3. 代码审查：对软件系统的代码进行审查，评估代码的质量、可读性、可维护性和可重用性等特点。

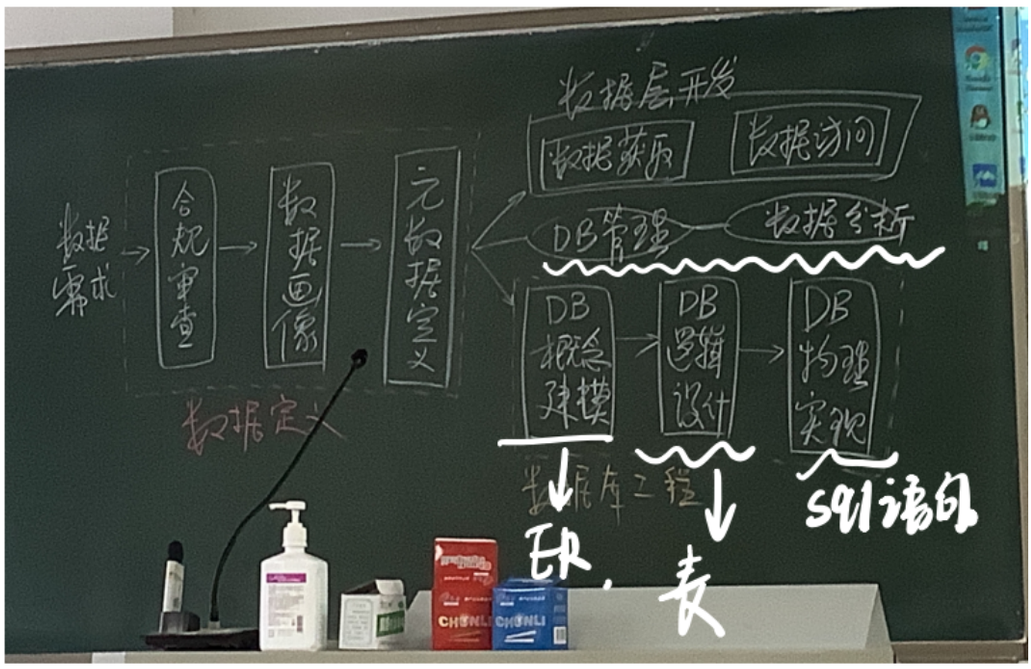
- **软件架构模式：**界面层、逻辑层、数据层和DB
-



1. 界面层（Presentation Layer）：负责与用户交互，展示数据和处理用户的输入。在Web应用中，通常是Web页面或Web应用程序。
2. 逻辑层（Application Layer）：负责处理业务逻辑，包括数据处理、计算、校验等。通常是一个独立的应用程序或服务。
3. 数据层（Data Layer）：负责处理数据的存储和访问，包括数据库和数据访问层。在传统的软件架构中，数据层通常是一个独立的数据库，而在现代的云架构中，数据层常常被拆分成多个分布式存储和处理服务。
4. 数据库（DB）：负责存储应用程序的数据。可以是关系型数据库（如MySQL、Oracle等）或非关系型数据库（如MongoDB、Redis等）。

- **界面工程：**
  - 界面属性：颜色、字体、排版
  - 界面元素：按钮、文本框、下拉框
  - 界面设计
    - **静态 （草图）、 动态（流程图）**
  - 界面实现
    - **内容、样式、行为分离（MVC或MVVM等设计模式）**
    - **低代码平台**
  - 界面质控
    - 两个等级：**愉悦感和可用性**
      - 是否美观、舒适、符合用户期望
      - 是否易于使用、操作是否流畅、是否容易出错
    - **界面审查：基于清单**

- 开发和维护前端用户界面，包括设计、开发、测试和部署等方面
- 界面工程师需要与其他开发人员、产品经理和设计师等密切协作，以确保用户界面能够满足业务需求和用户需求

- **数据工程：**
  - 关系型数据库、NoSQL数据库、数据仓库、数据湖
  - 数据需求：**使全部功能得以执行的所有数据**
  - 

  - **数据定义**
    - 合规审查（**前提**）：合法性和安全性
      - 数据合规官（Data Compliance Officer，**DCO**）是一个组织中负责数据合规的专业人员
    - 数据画像：对数据进行深度分析和可视化，以便更好地理解数据。数据画像可以帮助人们快速了解数据的特征、质量和价值，从而更好地利用数据。
    - 定义元数据：对数据进行描述和分类
  - **数据层开发**
    - 数据获取
    - 数据访问
  - **DB管理、数据分析**：备份、恢复、性能优化
  - **数据库工程**
    - DB概念模型：ER图
    - DB逻辑模型：表
    - DB物理实现：sql语言和文件系统
  - **生产数据**：实际生产中产生的数据，包括企业、机构或个人在业务过程中产生的所有数据。生产数据是实际业务活动的结果，包括交易数据、客户数据、产品数据、供应链数据等，是企业和机构的重要资产之一。
  - **元数据**：**先导型Proactive Metadata，后生型Reactive Metadata**
  - **数据本体**（Data Ontology）是指描述数据和数据之间关系的概念模型，是知识表示和知识管理的一种方式。数据本体通常包括一组定义良好的概念和关系，用于描述领域中的实体、属性和关系，并提供一种共享和重用这些概念和关系的方式。
    - 实体：描述领域中的实体，如人、物、事等。
    - 属性：描述实体的各种属性，如名称、年龄、性别等。
    - 关系：描述实体之间的各种关系，如父子关系、雇佣关系等。
  - Q：获得数据前需要做什么？
    1. 需要什么数据？
    2. 能收集么？能商用吗？

3. 有能力保护数据吗?
4. 基于什么条件来收集?
5. 数据怎么使用?（合规审查，数据合规官DCO）

- **管理的三重境界：**
  - 人：都需要经理调控
    - 工程师
    - 客户、投资人
    - 用户
  - 事
    - 变更未知，不想1+1算1w次那样简单
    - 风险预测未知；时间、成本管理困难
    - CTO不管理？ 不行
    - 胶冻团队：临时组建的跨部门或跨职能团队，在短时间内迅速响应某项任务或项目，快速协作完成任务并在完成后解散的一种团队形式。
      - 胶冻团队通常由各种技术和职能背景的成员组成，包括开发人员、测试人员、设计师、产品经理等。他们通常会在一段短时间内，例如一周或两周，全身心地致力于某项任务或项目，快速迭代、快速实现，以快速响应市场变化和客户需求。
      - 被管理、听话
      - 懂得为人之道，帮老板买coffee
  - 物
    - 产品
    - 文档（制品）
    - 议素
      - 待实现和决议的内容
      - 带移除修复的缺陷

> 物是指软件开发过程中的各种实物和文档，包括产品、文档和议素等。
>
> 产品是指软件开发过程中的各种产品、功能和特性。产品是软件开发的核心，开发人员需要根据产品需求和设计要求进行开发和测试，并且需要进行代码的维护和优化，以确保软件的功能和特性能够满足业务需求和用户需求。
>
> 文档是指软件开发过程中的各种文档和制品，包括需求文档、设计文档、测试文档、用户手册等。这些文档可以帮助开发人员更好地理解需求和设计要求，同时也可以为其他人员提供更好的理解和使用软件的支持。
>
> 议素是指软件开发过程中的待实现和决议的内容，包括需求变更、技术问题、设计问题等。议素通常需要进行讨论和决策，以确定最终的方案和做法。议素的管理需要确保开发人员能够及时了解最新的决策和方案，并且能够快速响应和实现。
>
> 待移除修复的缺陷是指在软件开发过程中发现的各种缺陷和问题，包括代码缺陷、逻辑错误、性能问题等。待移除修复的缺陷需要进行记录和跟踪，并且需要进行优先级排序和处理，以确保软件的质量和稳定性。

- **软件管理三部分**
  - 产品管理
  - 项目管理
  - 过程管理

> 产品管理是指对软件产品的全生命周期进行管理，包括产品规划、产品设计、产品开发、产品测试、产品发布和产品维护等。产品管理需要确保软件产品能够按照需求和设计要求进行开发、测试和部署，同时也需要关注用户需求和市场变化，以调整产品规划和设计，以满足不断变化的需求。

> 项目管理是指对软件开发项目进行管理，包括项目规划、项目执行、项目监控和项目收尾等。项目管理需要确保项目能够按照计划进行，控制项目的进度、成本和质量，同时也需要关注风险管理和变更管理，以确保项目能够顺利完成。
>
> 过程管理是指对软件开发过程进行管理，包括过程改进、过程规范、过程培训和过程评估等。过程管理需要确保软件开发过程能够按照最佳实践进行，同时也需要关注持续改进和团队成长，以提高软件开发过程的效率和质量。

- **软件管理的三种项目**：
    - **软件开发项目**
        - 项目规划
            - **进度规划**
                - **项目估算**
                    - 项目规模：项目规模越大，工作量和工期就越长，成本就越高。
                    - 技术难度：技术难度越高，开发周期就越长，成本就越高。
                    - 人力资源：人力资源充足，开发周期就越短，成本就越低。
                - **里程碑**定义：将整个项目分解为若干个可管理的阶段或目标，以便于项目管理人员对进度和成果进行跟踪和控制。里程碑可以是时间节点、关键事件或阶段性成果等。里程碑通常涉及到项目中的一些关键任务或目标的完成，如软件原型的完成、测试用例的编写、代码的实现、软件集成测试的完成等。
                    - 好处
                        - 确定项目的关键节点，以便于控制项目进度和质量。
                        - 促进项目团队合作，明确每个人员的工作目标和责任。
                        - 为项目管理人员提供有效的进度跟踪和控制工具。
                    - 具体的里程碑设定应该符合以下几个原则：
                        1. 明确：里程碑的目标和任务必须是清晰明确的，能够让团队成员和项目管理者清楚地了解其完成的标志和成果。
                        2. 可量化：里程碑的目标和任务必须是可量化的，能够通过具体的指标或标准进行衡量和评估。
                        3. 可行性：里程碑的目标和任务必须是可行的，能够在预定的时间和资源限制内完成。
                        4. 重要性：里程碑的目标和任务必须是重要的，能够对项目的进展和成功产生重要的影响。
                        5. 时间性：里程碑的目标和任务必须有明确的时间限制，能够在预定的时间内完成。
            - **质量规划**
                - **质量保障（QA）**
                    - 质量标准和规范：制定质量标准和规范，确保软件产品符合质量要求。
                    - 过程管理：建立过程管理体系，确保软件开发过程规范、可控、可持续，以提高软件产品的质量和效率。
                    - 质量培训：针对开发人员、测试人员等，提供相关的质量培训，以提高他们的质量意识和质量水平。
                - **质量控制（QC）【审查+测试】**
                    - 质量检查和测试：对软件产品进行质量检查和测试，确保软件产品的质量符合规定的标准和要求。
                    - 缺陷管理：对软件产品中发现的缺陷进行管理和跟踪，及时解决缺陷，提高软件产品的质量和可靠性。
                    - 质量评估审查：对软件产品的质量进行评估和分析，发现和解决质量问题，提高软件产品的质量和可靠性。
            - **项目监管**
                - 项目风险：对项目风险进行监控和控制，及时采取措施降低风险。
                - 项目团队：对项目团队进行管理和协调，确保项目成员之间的合作和沟通。
            - **项目改进**

- **软件运营项目**
- **软件维护项目**
  - 逆向开发，开发者小队+维护二次开发者小队
- **软件演化项目**
  - 常规
  - 非常规

> 1. 软件开发项目：这是一个全新的项目，旨在从头开始构建一个新的软件系统。在软件开发项目中，需要进行项目规划、进度规划和质量规划，以确保项目按时、高质量地完成。还需要进行项目监管和改进，以确保项目能够达到预期的目标。
> 2. 软件运营项目：这是一个旨在维护和管理现有软件系统的项目。在软件运营项目中，需要确保软件系统的稳定性和可靠性，并提供支持和维护服务。此外，还需要关注用户反馈和需求，以不断改进和优化软件系统。
> 3. 软件维护项目：这是一个旨在修复和改进现有软件系统的项目。在软件维护项目中，需要识别和修复软件系统中的缺陷和问题，并进行软件更新和改进，以确保软件系统的高质量和可靠性。
> 4. 软件演化项目：这是一个旨在对现有软件系统进行扩展和改进的项目。在软件演化项目中，需要根据用户需求和市场变化，对软件系统进行扩展和改进，以满足不断变化的需求和要求。
>   - 常规软件演化项目：这种类型的项目通常是在现有软件系统的基础上进行小规模的扩展和改进，以满足用户的新需求和要求。常规软件演化项目通常具有以下特点：
>   - 需求明确：常规软件演化项目通常是基于用户的具体需求和要求进行的，因此需求比较明确。
>   - 可预测性高：常规软件演化项目的风险相对较低，因为它们通常是在现有软件系统的基础上进行小规模的扩展和改进，所以可以比较准确地预测开发进度和成本。
>   - 开发流程相对简单：常规软件演化项目通常不需要进行大规模的架构设计和技术选型，因此开发流程相对简单。
>   - 非常规软件演化项目：这种类型的项目通常是在现有软件系统的基础上进行大规模的扩展和改进，或者涉及到核心业务的调整和变更。非常规软件演化项目通常具有以下特点：
>   - 需求不确定：非常规软件演化项目通常涉及到核心业务的调整和变更，或者需要进行大规模的架构设计和技术选型，因此需求相对不确定。
>   - 风险高：非常规软件演化项目的风险相对较高，因为它们通常涉及到核心业务的调整和变更，或者需要进行大规模的架构设计和技术选型，因此开发进度和成本难以准确预测。
>   - 开发流程相对复杂：非常规软件演化项目通常需要进行大规模的架构设计和技术选型，因此开发流程相对复杂。

- **开发过程需要考虑的：规模 -> 工作量 -> 工期、用人 -> 成本**
- 规模增长，而软件具有**复杂性和频变性**，**软件有三高：价值、成本、风险**
- 这些因素都会对软件开发的工作量、工期和成本产生重要的影响，做决策规划之前需要确认和考虑：
  - 立项谈判
  - 目标产品
  - 合同价格
  - **交付日期**
  - **完工标准**

人月神话

- Def：增加开发人员的数量并不能保证完成软件开发任务的时间缩短，因为新加入的人员需要时间来适应工作环境，并且他们的加入可能会引起更多的沟通和协调问题，从而导致开发时间延长。
- **主要原因：串并行共同进行，很复杂**
- **软件开发是一项高度复杂的任务，需要许多不同领域的专业人才协同工作。**在软件开发过程中，每个人员都要承担特定的任务，并在完成任务的过程中与其他人员进行协调和沟通。**当需要增加开发人员的数量时，这些新加入的人员需要时间来了解和适应项目的环境**，包括软件的架构、代码规范、开发工具等等。此外，新加入的人员可能会引入新的沟通和协调问题，这可能会导致开发进度的延误
- 布鲁克斯认为，**软件开发的进度不应该只考虑开发人员的数量，而应该考虑开发过程的复杂性和团队的协作能力。**布鲁克斯提出了一种“向左转”（left-shift）的方法，即在软件开发周期的早期阶段，尽可能地投入更多的资源和人力，以确保软件的需求分析和设计阶段尽可能完善，从而避免在开发和测试阶段出现问题。同时，布鲁克斯也强调了软件开发中的团队合作和沟通的重要性，这可以通过采用合适的开发方法、工具和流程来实现。

软件设计原则

1. 单一职责原则 (SRP)
   单一职责原则要求一个类或模块只应该负责一项职责。这个原则的主要目的是将职责分离开来，使得每个类或模块都具有清晰的目的和职责，从而提高代码的可读性、可维护性和可测试性。如果一个类或模块负责多个职责，那么在修改其中一个职责时可能会影响到其他职责，从而导致软件系统变得更加复杂和难以维护。
2. 开闭原则 (OCP)
   开闭原则要求一个软件实体应该对扩展开放，对修改关闭。这个原则的主要目的是使得软件系统更加灵活和可扩展。开闭原则要求将变化的部分与不变的部分分离开来，使得软件系统的不变部分可以稳定地运行，而变化的部分可以通过扩展来满足新的需求。这样可以避免在修改软件系统时对原有代码的影响，从而提高软件系统的可维护性和可扩展性。
3. 里氏替换原则 (LSP)
   里氏替换原则要求子类可以完全替换父类，而不会影响软件系统的正确性和稳定性。这个原则的主要目的是使得软件系统更加灵活和可复用。当一个子类完全替换父类时，它可以在不影响软件系统的正确性和稳定性的情况下修改或扩展父类的行为。
4. 依赖倒置原则 (DIP)
   依赖倒置原则要求高层模块不应该依赖底层模块，而应该依赖于抽象。这个原则的主要目的是将依赖关系抽象出来，使得软件系统更加灵活和可测试。依赖倒置原则要求使用抽象接口来定义模块之间的依赖关系，从而使得模块之间的依赖关系更加松散和灵活。
5. 接口隔离原则 (ISP)
   接口隔离原则要求客户端不应该依赖它不需要的接口。这个原则的主要目的是使得软件系统更加灵活和可扩展。接口隔离原则要求将接口细分成独立的部分，使得客户端只依赖于它需要的接口，而不需要依赖于不需要的接口。这样可以使得软件系统更加灵活和可扩展。
6. 最少知识原则 (LKP)
   最少知识原则要求一个软件实体应当尽可能少地与其他实体发生相互作用。这个原则的主要目的是将对象之间的耦合性降到最低，从而提高软件系统的可维护性和可扩展性。最少知识原则要求尽可能减少对象之间的通信，使得每个对象只需要与它的近邻对象进行交互。
7. 面向对象设计原则 (OOD)
   面向对象设计原则包括封装、抽象、继承和多态，它们可以帮助开发人员设计出符合面向对象设计原则的软件系统。具体来说，封装可以隐藏对象的实现细节，使得对象的状态更加安全；抽象可以将对象的共性抽象出来，使得对象更加灵活；继承可以使得子类继承父类的属性和方法，从而提高代码的复用性；多态可以使得不同对象对同一个消息响应不同的行为，从而提高代码的灵活性。这些原则一起构成了面向对象设计的基础，可以帮助开发人员设计出具有高内聚性、低耦合性、可维护性、可扩展性和可测试性的软件系统。

这些软件设计原则是开发人员在软件设计过程中需要遵循的基本原则，它们可以帮助开发人员设计出具有高内聚性、低耦合性、高可维护性、高可扩展性和高可测试性的软件系统。在具体应用时，开发人员应该结合具体的需求和场景来选择合适的软件设计原则，从而设计出更加优秀的软件系统。

1. 提高代码的可读性：这些原则可以使代码更加清晰、易于理解和维护，从而提高代码的可读性。

2. 提高代码的可维护性：这些原则可以使代码更加灵活、易于修改和扩展，从而提高代码的可维护性。
3. 提高代码的可测试性：这些原则可以使代码更加模块化、易于测试和调试，从而提高代码的可测试性。
4. 提高代码的复用性：这些原则可以使代码更加通用、易于复用，从而提高代码的复用性。
5. 降低代码的耦合度：这些原则可以使代码之间的依赖关系更加松散、耦合度更低，从而降低代码的耦合度。
6. 提高代码的灵活性：这些原则可以使代码更加灵活、易于适应变化，从而提高代码的灵活性。

架构模式

1. **分层架构模式**
   分层架构模式（Layered Architecture Pattern）是一种基于分层的软件架构模式，它将软件系统分解为多个层次，每个层次都有其特定的功能和职责。分层架构模式可以使得软件系统的各个层次之间的依赖性更加清晰，从而提高软件系统的可维护性和可扩展性。
2. 客户端-服务器模式
   客户端-服务器模式（Client-Server Pattern）是一种基于分布式的软件架构模式，它将软件系统分解为两个独立的部分：客户端和服务器。客户端负责向用户提供界面和处理用户输入，而服务器负责处理客户端的请求并返回相应的结果。客户端-服务器模式可以使得软件系统更加灵活和可扩展，同时也可以提高软件系统的性能和安全性。
3. MVC架构模式
   MVC架构模式（Model-View-Controller Pattern）是一种基于模型-视图-控制器的软件架构模式，它将软件系统分解为三个独立的部分：模型、视图和控制器。模型负责处理数据，视图负责显示数据，而控制器负责处理用户输入和控制数据流。MVC架构模式可以使得软件系统更加灵活和可扩展，同时也可以提高软件系统的可维护性和可测试性。
4. 事件驱动架构模式
   事件驱动架构模式（Event-Driven Architecture Pattern）是一种基于事件的软件架构模式，它将软件系统分解为多个独立的组件，每个组件都可以处理来自其他组件的事件。事件驱动架构模式可以使得软件系统更加灵活和可扩展，同时也可以提高软件系统的性能和响应能力。
5. **微服务架构模式**
   微服务架构模式（Microservices Architecture Pattern）是一种基于微服务的软件架构模式，它将软件系统分解为多个独立的微服务，每个微服务都可以独立部署和升级。微服务架构模式可以使得软件系统更加灵活和可扩展，同时也可以提高软件系统的可维护性和可测试性。
6. **领域驱动设计模式**
   领域驱动设计模式（Domain-Driven Design Pattern）是一种基于领域模型的软件架构模式，**它将软件系统的设计焦点放在业务领域上**，通过**建立领域模型来描述业务领域中的概念和关系**。领域驱动设计模式可以使得软件系统更加贴近业务需求，同时也可以提高软件系统的可维护性和可扩展性。

软件质量属性

1. 功能性
   功能性是指软件产品是否满足用户的需求和期望，包括软件产品是否具有正确的功能、是否能够满足用户的需求、是否易于使用等方面。
2. 可靠性
   可靠性是指软件产品在长时间运行和多种环境下的稳定性和可靠性，包括软件产品是否具有足够的可靠性、是否能够正确处理异常情况、是否能够保证数据的完整性等方面。
3. 可用性
   可用性是指软件产品是否易于使用和操作，包括软件产品是否具有良好的界面设计、是否易于学习、是否具有良好的用户体验等方面。
4. 可维护性
   可维护性是指软件产品在修改和维护时的易于性和可扩展性，包括软件产品是否具有良好的代码结构、是否易于维护和修改、是否能够方便地进行扩展等方面。

5. 可移植性
   可移植性是指软件产品在不同环境下的可移植性和兼容性，包括软件产品是否能够在不同的操作系统和硬件平台上运行、是否能够方便地进行移植等方面。
6. 安全性
   安全性是指软件产品在使用和操作时的安全性，包括软件产品是否能够保护用户的隐私、是否能够防止未经授权的访问等方面。

设计模式简述

为什么要有设计？设计的必要性是什么？

- 软件设计是指在满足软件需求的前提下，通过规划、组织、编排、调配和优化软件各个组成部分之间的相互关系，使软件系统具有良好的可用性、可维护性、可扩展性、可移植性等特性的过程。软件设计的目的是使得软件系统在实现其功能的同时，具备更好的质量和可靠性，能够满足用户的需求。
- 设计的必要性在于，一个好的软件设计可以降低软件实现的难度和风险，提高软件的可维护性、可扩展性、可重用性和可移植性，使得软件开发过程更加高效、可控和可预测。【例如我们想要构建一个线性复杂度增长的软件系统，就需要合理设计】

需求与设计（架构）的关系是什么？

- 需求和设计密切相关、相互促进。需求是软件设计的前提和基础，设计则是满足需求的手段和途径；设计应该根据需求进行，设计的结果也需要验证是否满足需求；如果需求变化，设计也需要相应地进行调整。因此，需求和设计是一个不断迭代的过程。
- 需求和设计是反复交替进行的，是一个双峰模型，有什么好处？【三点核心：减少偏差；增加设计质量；获得用户的认可】
  - **更好的风险控制**：通过反复迭代，可以**及时发现和解决问题，减少软件开发过程中的风险和错误。**这有助于提高软件的质量和可靠性，降低开发成本和风险。
  - **更高效的软件开发**：需求和设计之间的反复迭代可以使得软件开发更加高效。通过在需求和设计之间进行反复迭代，可以**更好地理解用户需求，优化设计方案，减少后期修改和调整的工作量**，从而提高软件开发的效率和质量。
  - **更好地满足用户需求，增强用户体验。**
  - **获得用户的认可**：实时与客户交接设计内容并根据其需求再次修改，可以赢得客户的信任和认可，进而带来长期的商业价值和竞争优势
  - **增加设计质量**：反复迭代可以促进设计方案的优化和改进，从而提高设计质量和可靠性。
- 需求和设计之间的反复迭代是一个双向的过程，需要在需求和设计之间建立良好的沟通和协作机制。在这个过程中，**需求的变化会对设计产生影响，设计的变化也会反过来影响需求。**因此，在需求和设计之间进行反复迭代的过程中，需要**保持灵活和敏捷，及时响应变化，确保需求和设计的一致性和完整性。**
- 需求和设计之间的反复迭代也**需要考虑时间和资源的限制。**在开发过程中，需要在需求和设计之间进行适当的平衡，**避免过度设计和需求膨胀，从而导致开发时间和成本的增加。**

分类

【创建型模式 (Creational Patterns)】
工厂方法模式 (Factory Method Pattern)
抽象工厂模式 (Abstract Factory Pattern)
单例模式 (Singleton Pattern)
建造者模式 (Builder Pattern)
原型模式 (Prototype Pattern)

【结构型模式 (Structural Patterns)】
适配器模式 (Adapter Pattern)
桥接模式 (Bridge Pattern)
组合模式 (Composite Pattern)
装饰器模式 (Decorator Pattern)

外观模式 (Facade Pattern)
享元模式 (Flyweight Pattern)
代理模式 (Proxy Pattern)

【行为型模式 (Behavioral Patterns)】
1.责任链模式 (Chain of Responsibility Pattern)
2.命令模式 (Command Pattern)
3.解释器模式 (Interpreter Pattern)
4.迭代器模式 (Iterator Pattern)
5.中介者模式 (Mediator Pattern)
6.备忘录模式 (Memento Pattern)
7.观察者模式 (Observer Pattern)
8.状态模式 (State Pattern)
9.策略模式 (Strategy Pattern)
10.模板方法模式 (Template Method Pattern)
11.访问者模式 (Visitor Pattern)

简述

工厂方法模式

工厂方法模式是一种创建型设计模式，它提供了一种将对象的创建委托给子类的方式，从而实现了对象的创建和使用的解耦。在工厂方法模式中，定义一个创建对象的接口，但让子类决定实例化哪个类。工厂方法允许类把实例化延迟到子类中进行，因此可以避免在代码中使用具体类。

使用场景：

- 当需要创建的对象具有共同的接口时
- 当需要动态指定创建对象的具体类时
- 当需要隐藏具体类的实现细节时

优缺点：

- 优点：
  - 工厂方法模式将对象的创建与使用分离，降低了系统的耦合度，增强了系统的扩展性。
  - 工厂方法模式可以动态地指定创建对象的具体类，增强了系统的灵活性。
  - 工厂方法模式通过使用抽象层，可以隐藏具体类的实现细节，使得代码更易于维护和升级。
- 缺点：
  - 工厂方法模式增加了系统的复杂度，需要额外定义抽象层和具体实现类。

抽象工厂模式

抽象工厂模式是一种创建型设计模式，它提供了一种将一组相关或相互依赖的对象集合进行创建的方式，而无需指定它们具体的类。在抽象工厂模式中，定义了一个抽象工厂接口，其中包含了一组创建抽象产品的方法，每个方法对应一种具体产品类型，而具体的工厂类则实现了这些抽象方法，可以创建具体的产品对象。

使用场景：

- 当需要创建一组相关或相互依赖的对象时
- 当希望将产品的创建与使用解耦，避免客户端直接依赖于具体产品类时
- 当需要动态切换产品族时

优缺点：

- 优点：
  - 抽象工厂模式将一组相关或相互依赖的对象集合进行创建，保证了产品的统一性，增强了系统的稳定性。
  - 抽象工厂模式可以动态切换产品族，增强了系统的灵活性。

    - 抽象工厂模式将产品的创建与使用解耦，避免了客户端直接依赖于具体产品类，增强了系统的扩展性。
- 缺点：
    - 抽象工厂模式增加了系统的复杂度，需要定义抽象工厂类和具体工厂类，同时还需要定义一组抽象产品类和具体产品类。

单例模式

单例模式是一种创建型设计模式，它保证一个类只有一个实例，并提供了一个全局访问点。在单例模式中，类自身负责保证只有一个实例被创建，并提供了一个静态方法来访问该实例。

使用场景：

- 当需要确保一个类只有一个实例时
- 当需要提供一个全局访问点时

优缺点：

- 优点：
    - 单例模式保证了一个类只有一个实例，避免了多个实例造成的资源浪费。
    - 单例模式提供了一个全局访问点，方便了对单例对象的访问。
- 缺点：
    - 单例模式可能会引起全局状态的共享，增加了系统的耦合度。
    - 单例模式对扩展不太友好，因为它不支持子类化

建造者模式

建造者模式是一种创建型设计模式，它将一个复杂对象的构建过程与其表示相分离，使得同样的构建过程可以创建不同的表示。在建造者模式中，定义一个建造者接口，其中包含了一组构建产品的方法，每个方法对应一种产品部件，而具体的建造者类则实现了这些方法，可以按照一定的顺序来构建产品对象。

使用场景：

- 当需要构建一个复杂对象时，该对象由多个部件组成且部件的构建顺序不确定时
- 当需要将构建过程与表示相分离，使得同样的构建过程可以创建不同的表示时

优缺点：

- 优点：
    - 建造者模式将一个复杂对象的构建过程与其表示相分离，使得同样的构建过程可以创建不同的表示，增强了系统的灵活性。
    - 建造者模式可以通过控制建造过程来控制产品的细节，提高了产品的精度和质量。
- 缺点：
    - 建造者模式增加了系统的复杂度，需要额外定义建造者类和具体建造者类。

原型模式

原型模式是一种创建型设计模式，它通过复制已有对象来创建新的对象实例，而不是通过实例化类来创建。在原型模式中，定义一个原型接口，其中包含了一个克隆方法，具体的原型类则实现了这个方法，可以通过复制自身来创建新的对象。

使用场景：

- 当需要创建的对象类型可以通过复制已有对象来获得时
- 当需要避免使用类的实例化来创建新的对象时

优缺点：

- 优点：

    - 原型模式通过复制已有对象来创建新的对象实例，避免了使用类的实例化来创建新对象的过程，提高了对象的创建效率。
    - 原型模式可以通过修改已有对象的属性来创建新的对象，增强了系统的灵活性。
- 缺点：
    - 原型模式需要额外定义原型接口和具体原型类，增加了系统的复杂度。同时，需要注意深浅拷贝的问题，避免对原型对象的修改影响到新创建的对象。

适配器模式

适配器模式是一种结构型设计模式，它将一个接口转换成客户端所期望的另一个接口，从而使得原本不兼容的类可以一起工作。在适配器模式中，定义一个适配器类，该类实现了客户端所期望的接口，同时持有一个需要适配的对象，通过调用该对象的方法来实现客户端的请求。

使用场景：

- 当需要使用一个已经存在的类，但其接口与需要的接口不兼容时
- 当需要使用多个不兼容的类协同工作时
- 当需要重用一个类，但是其接口与系统其他部分不兼容时

优缺点：

- 优点：
    - 适配器模式可以让原本不兼容的类一起工作，增强了系统的灵活性。
    - 适配器模式可以重用已有的类，避免了重复造轮子。
    - 适配器模式可以将不同的类进行解耦，增强了系统的可维护性和可扩展性。
- 缺点：
    - 适配器模式增加了系统的复杂度，需要额外定义适配器类和适配器接口。

桥接模式

桥接模式是一种结构型设计模式，它将抽象部分和实现部分分离开来，使得它们可以独立地进行变化。在桥接模式中，定义一个抽象部分的接口，其中包含了一组操作，而具体的实现则在实现部分中完成，通过将抽象部分和实现部分分别抽象出来，使得它们可以独立地进行变化。

使用场景：

- 当需要将一个系统分成多个层次结构时
- 当需要在多个层次结构中独立地变化抽象部分和实现部分时
- 当需要在抽象部分和实现部分之间建立一个稳定的接口时

优缺点：

- 优点：
    - 桥接模式将抽象部分和实现部分分离开来，使得它们可以独立地进行变化，增强了系统的灵活性。
    - 桥接模式可以在多个层次结构中独立地变化抽象部分和实现部分，增强了系统的可维护性和可扩展性。
    - 桥接模式可以在抽象部分和实现部分之间建立一个稳定的接口，增强了系统的稳定性。
- 缺点：
    - 桥接模式需要额外定义抽象部分和实现部分，增加了系统的复杂度。

组合模式

组合模式是一种结构型设计模式，它将对象组合成树形结构以表示“部分-整体”的层次结构，使得客户端可以统一地处理单个对象和组合对象。在组合模式中，定义一个抽象的组件类，其中包含了一组操作，具体的组件类可以是叶子节点或者是容器节点，容器节点可以包含其他组件对象。

使用场景：

- 当需要表示部分-整体的层次结构时

- 当需要统一地处理单个对象和组合对象时
- 当需要忽略组合对象与单个对象之间的差异时

优缺点：

- 优点：
  - 组合模式可以将对象组织成树形结构，方便对整体和部分进行管理和操作。
  - 组合模式可以统一地处理单个对象和组合对象，增强了系统的灵活性- 组合模式可以方便地增加新的组件类，扩展了系统的可扩展性。
- 缺点：
  - 组合模式可能会使设计变得复杂，需要考虑如何定义抽象的组件类和具体的组件类。

装饰器模式

装饰器模式是一种结构型设计模式，它允许在不改变对象结构的情况下，动态地添加一些额外的职责。在装饰器模式中，定义一个装饰器类，该类实现了与被装饰对象相同的接口，同时持有一个被装饰对象的引用，通过在调用被装饰对象的方法之前或之后添加额外的逻辑来实现装饰功能。

使用场景：

- 当需要动态地为对象添加一些额外的职责时
- 当不能使用继承来扩展对象的行为时
- 当需要透明地扩展对象的功能时

优缺点：

- 优点：
  - 装饰器模式可以动态地为对象添加一些额外的职责，增强了系统的灵活性。
  - 装饰器模式可以避免使用继承来扩展对象的行为，避免了类爆炸的问题。
  - 装饰器模式可以透明地扩展对象的功能，不会影响到其他对象。
- 缺点：
  - 装饰器模式会增加许多小对象，增加了系统的复杂度。

外观模式

外观模式是一种结构型设计模式，它为一组复杂的子系统提供一个统一的接口，从而使得客户端可以更容易地使用这些子系统。在外观模式中，定义一个外观类，该类封装了子系统的复杂逻辑，客户端只需要通过外观类的接口来访问子系统，而不需要直接与子系统进行交互。

使用场景：

- 当需要简化一组复杂的子系统接口时
- 当需要将系统与客户端解耦时
- 当需要隐藏子系统的实现细节时

优缺点：

- 优点：
  - 外观模式可以简化一组复杂的子系统接口，增强了系统的易用性。
  - 外观模式可以将系统与客户端解耦，增强了系统的可维护性和可扩展性。
  - 外观模式可以隐藏子系统的实现细节，增强了系统的安全性。
- 缺点：
  - 外观模式可能会导致系统变得更加复杂，需要考虑如何定义外观类和子系统类的关系。

享元模式

享元模式是一种结构型设计模式，它通过共享对象来减少系统中的对象数量，从而提高系统的性能和内存利用率。在享元模式中，定义一个享元工厂类，该类维护了一个享元池，用于存储共享对象，客户端需要使用共享对象时，可以通过享元工厂类来获取对象。

使用场景：

- 当需要创建大量相似对象时
- 当需要减少系统中对象数量时
- 当对象的状态可以被共享时

优缺点：

- 优点：
  - 享元模式可以减少系统中的对象数量，提高了系统的性能和内存利用率。
  - 享元模式可以共享对象，减少了系统中的重复对象。
- 缺点：
  - 享元模式可能会导致系统变得更加复杂，需要考虑如何定义享元工

代理模式

代理模式是一种结构型设计模式，它为其他对象提供一个代理，从而控制对原始对象的访问。在代理模式中，定义一个代理类，该类持有一个被代理对象的引用，客户端通过代理类来访问被代理对象，代理类可以在访问被代理对象之前或之后添加额外的逻辑，从而实现控制对被代理对象的访问。

使用场景：

- 当需要控制对原始对象的访问时
- 当需要对原始对象的访问进行扩展或修改时
- 当需要在访问原始对象之前或之后添加额外的逻辑时

优缺点：

- 优点：
  - 代理模式可以控制对原始对象的访问，增强了系统的安全性。
  - 代理模式可以在访问原始对象之前或之后添加额外的逻辑，增强了系统的灵活性。
  - 代理模式可以对原始对象的访问进行扩展或修改，增强了系统的可扩展性。
- 缺点：
  - 代理模式可能会增加系统的复杂度，需要定义代理类和被代理类的关系。
  - 代理模式可能会降低系统的性能，因为代理类需要对访问进行控制和处理。

责任链模式

内容概述：
责任链模式是一种行为模式，它允许将请求沿着处理链传递，直到有一个处理器能够处理该请求。该模式的核心思想是将请求与处理解耦，将请求传递到一个处理链中，每个处理器都有机会检查并处理请求。如果一个处理器不能处理请求，它就将请求传递给下一个处理器，直到有一个处理器能够处理它。

使用场景：
责任链模式通常用于需要动态处理请求的场景，例如在一个系统中有多个对象可以处理请求，但是具体由哪个对象处理请求是在运行时决定的。责任链模式还可以用于系统中的日志记录、异常处理等场景。

优缺点：
责任链模式的优点是将请求处理的逻辑解耦，使得系统更加灵活。由于请求可以沿着处理链传递，因此可以动态地添加或删除处理器，以满足不同的需求。责任链模式的缺点是可能会导致请求在链中被重复处理或者无法被处理。如果处理链太长或者处理器之间的关系太复杂，可能会影响系统的性能和可维护性。

命令模式

内容概述：
命令模式是一种行为模式，它将一个请求封装为一个对象，从而使你可以将请求的操作、参数和执行操作的对象解耦。该模式的核心思想是将请求封装为一个命令对象，然后将命令对象传递给调用者。调用者可以在需要时执行命令，或者将命令存储在队列中，以便稍后执行。

使用场景：
命令模式通常用于需要将请求和执行操作的对象解耦的场景。例如在一个系统中，需要将操作记录下来以便撤销或重做，可以使用命令模式来实现。命令模式还可以用于实现菜单、工具栏等用户界面元素。

优缺点：
命令模式的优点是能够在不修改现有代码的情况下增加新的命令。由于命令对象封装了请求的操作、参数和执行操作的对象，因此可以方便地添加新的命令，而无需修改现有的代码。命令模式的缺点是可能会导致系统中的命令对象过多。如果系统中有大量的命令对象，可能会影响系统的性能和可维护性。

解释器模式

内容概述：
解释器模式（Interpreter Pattern）提供了评估语言的语法或表达式的方式，它属于行为型模式。这种模式实现了一个表达式接口，该接口解释一个特定的上下文。这种模式被用在 SQL 解析、符号处理引擎等。

意图：给定一个语言，定义它的文法表示，并定义一个解释器，这个解释器使用该标识来解释语言中的句子。

主要解决：对于一些固定文法构建一个解释句子的解释器。

何时使用：如果一种特定类型的问题发生的频率足够高，那么可能就值得将该问题的各个实例表述为一个简单语言中的句子。这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题。

如何解决：构建语法树，定义终结符与非终结符。

关键代码：构建环境类，包含解释器之外的一些全局信息，一般是 HashMap。

应用实例：编译器、运算表达式计算。

优点： 1、可扩展性比较好，灵活。 2、增加了新的解释表达式的方式。 3、易于实现简单文法。

缺点： 1、可利用场景比较少。 2、对于复杂的文法比较难维护。 3、解释器模式会引起类膨胀。 4、解释器模式采用递归调用方法。

使用场景： 1、可以将一个需要解释执行的语言中的句子表示为一个抽象语法树。 2、一些重复出现的问题可以用一种简单的语言来进行表达。 3、一个简单语法需要解释的场景。

注意事项：可利用场景比较少，JAVA 中如果碰到可以用 expression4J 代替。

迭代器模式

迭代器模式是一种行为模式，它提供了一种方法来顺序访问集合中的元素，而无需暴露集合的内部表示。该模式的核心思想是将集合和迭代器分离，从而使得集合可以独立地改变其内部表示方式，而不影响迭代器的使用。

迭代器模式是一种行为模式，它定义了一种访问聚合对象中各个元素的方法，而不需要暴露聚合对象的内部表示。该模式的核心思想是将访问聚合对象的方法封装成一个迭代器对象，从而使得客户端可以通过迭代器对象来访问聚合对象中的元素。

意图：提供一种方法顺序访问一个聚合对象中各个元素, 而又无须暴露该对象的内部表示。

主要解决：不同的方式来遍历整个整合对象。

何时使用：遍历一个聚合对象。

如何解决：把在元素之间游走的责任交给迭代器，而不是聚合对象。

关键代码：定义接口：hasNext, next。

应用实例：JAVA 中的 iterator。

优点： 1、它支持以不同的方式遍历一个聚合对象。 2、迭代器简化了聚合类。 3、在同一个聚合上可以有多个遍历。 4、在迭代器模式中，增加新的聚合类和迭代器类都很方便，无须修改原有代码。

缺点：由于迭代器模式将存储数据和遍历数据的职责分离，增加新的聚合类需要对应增加新的迭代器类，类的个数成对增加，这在一定程度上增加了系统的复杂性。

使用场景： 1、访问一个聚合对象的内容而无须暴露它的内部表示。 2、需要为聚合对象提供多种遍历方式。 3、为遍历不同的聚合结构提供一个统一的接口。

注意事项：迭代器模式就是分离了集合对象的遍历行为，抽象出一个迭代器类来负责，这样既可以做到不暴露集合的内部结构，又可让外部代码透明地访问集合内部的数据。

中介者模式

中介者模式是一种行为模式，它定义了一个中介对象来封装一系列对象之间的交互。该模式的核心思想是将对象之间的交互从彼此解耦，而将其集中在中介者对象中进行处理。

意图：用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

主要解决：对象与对象之间存在大量的关联关系，这样势必会导致系统的结构变得很复杂，同时若一个对象发生改变，我们也需要跟踪与之相关联的对象，同时做出相应的处理。

何时使用：多个类相互耦合，形成了网状结构。

如何解决：将上述网状结构分离为星型结构。

关键代码：对象 Colleague 之间的通信封装到一个类中单独处理。

应用实例： 1、中国加入 WTO 之前是各个国家相互贸易，结构复杂，现在是各个国家通过 WTO 来互相贸易。 2、机场调度系统。 3、MVC 框架，其中C（控制器）就是 M（模型）和 V（视图）的中介者。

优点： 1、降低了类的复杂度，将一对多转化成了一对一。 2、各个类之间的解耦。 3、符合迪米特原则。

缺点：中介者会庞大，变得复杂难以维护。

使用场景： 1、系统中对象之间存在比较复杂的引用关系，导致它们之间的依赖关系结构混乱而且难以复用该对象。 2、想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类。

注意事项：不应当在职责混乱的时候使用。

备忘录模式

备忘录模式是一种行为模式，它允许在不破坏封装性的前提下捕获和恢复对象的内部状态。该模式的核心思想是将对象的状态保存到备忘录中，然后可以将备忘录存储在外部，以便稍后恢复对象的状态。

意图：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。

主要解决：所谓备忘录模式就是在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样可以在以后将对象恢复到原先保存的状态。

何时使用：很多时候我们总是需要记录一个对象的内部状态，这样做的目的就是为了允许用户取消不确定或者错误的操作，能够恢复到他原先的状态，使得他有"后悔药"可吃。

如何解决：通过一个备忘录类专门存储对象状态。

关键代码：客户不与备忘录类耦合，与备忘录管理类耦合。

应用实例： 1、后悔药。 2、打游戏时的存档。 3、Windows 里的 ctrl + z。 4、IE 中的后退。 5、数据库的事务管理。

优点： 1、给用户提供了一种可以恢复状态的机制，可以使用户能够比较方便地回到某个历史的状态。 2、实现了信息的封装，使得用户不需要关心状态的保存细节。

缺点：消耗资源。如果类的成员变量过多，势必会占用比较大的资源，而且每一次保存都会消耗一定的内存。

使用场景： 1、需要保存/恢复数据的相关状态场景。 2、提供一个可回滚的操作。

注意事项： 1、为了符合迪米特原则，还要增加一个管理备忘录的类。 2、为了节约内存，可使用原型模式+备忘录模式。

观察者模式

观察者模式是一种行为模式，它定义了一种对象之间的一对多依赖关系，以便在一个对象状态发生变化时，所有依赖于它的对象都得到通知并自动更新。该模式的核心思想是将对象之间的依赖关系解耦，从而使得对象可以独立地改变状态，而不会影响其他对象。

意图：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

主要解决：一个对象状态改变给其他对象通知的问题，而且要考虑到易用和低耦合，保证高度的协作。

何时使用：一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知，进行广播通知。

如何解决：使用面向对象技术，可以将这种依赖关系弱化。

关键代码：在抽象类里有一个 ArrayList 存放观察者们。

应用实例： 1、拍卖的时候，拍卖师观察最高标价，然后通知给其他竞价者竞价。 2、西游记里面悟空请求菩萨降服红孩儿，菩萨洒了一地水招来一个老乌龟，这个乌龟就是观察者，他观察菩萨洒水这个动作。

优点： 1、观察者和被观察者是抽象耦合的。 2、建立一套触发机制。

缺点： 1、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。 2、如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。 3、观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

使用场景：

- 一个抽象模型有两个方面，其中一个方面依赖于另一个方面。将这些方面封装在独立的对象中使它们可以各自独立地改变和复用。
- 一个对象的改变将导致其他一个或多个对象也发生改变，而不知道具体有多少对象将发生改变，可以降低对象之间的耦合度。
- 一个对象必须通知其他对象，而并不知道这些对象是谁。
- 需要在系统中创建一个触发链，A对象的行为将影响B对象，B对象的行为将影响C对象……，可以使用观察者模式创建一种链式触发机制。

注意事项： 1、JAVA 中已经有了对观察者模式的支持类。 2、避免循环引用。 3、如果顺序执行，某一观察者错误会导致系统卡壳，一般采用异步方式。

状态模式

状态模式是一种行为模式，它允许对象在内部状态发生变化时改变它们的行为。该模式的核心思想是将对象的状态和行为分离，从而使得对象可以独立地改变它们的状态，而不会影响其他对象。

在状态模式（State Pattern）中，类的行为是基于它的状态改变的。这种类型的设计模式属于行为型模式。

在状态模式中，我们创建表示各种状态的对象和一个行为随着状态对象改变而改变的 context 对象。

意图：允许对象在内部状态发生改变时改变它的行为，对象看起来好像修改了它的类。

主要解决：对象的行为依赖于它的状态（属性），并且可以根据它的状态改变而改变它的相关行为。

何时使用：代码中包含大量与对象状态有关的条件语句。

如何解决：将各种具体的状态类抽象出来。

关键代码：通常命令模式的接口中只有一个方法。而状态模式的接口中有一个或者多个方法。而且，状态模式的实现类的方法，一般返回值，或者是改变实例变量的值。也就是说，状态模式一般和对象的状态有关。实现类的方法有不同的功能，覆盖接口中的方法。状态模式和命令模式一样，也可以用于消除 if...else 等条件选择语句。

应用实例： 1、打篮球的时候运动员可以有正常状态、不正常状态和超常状态。 2、曾侯乙编钟中，'钟是抽象接口','钟A'等是具体状态，'曾侯乙编钟'是具体环境（Context）。

优点： 1、封装了转换规则。 2、枚举可能的状态，在枚举状态之前需要确定状态种类。 3、将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为。 4、允许状态转换逻辑与状态对象合成一体，而不是某一个巨大的条件语句块。 5、可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数。

缺点： 1、状态模式的使用必然会增加系统类和对象的个数。 2、状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱。 3、状态模式对"开闭原则"的支持并不太好，对于可以切换状态的状态模式，增加新的状态类需要修改那些负责状态转换的源代码，否则无法切换到新增状态，而且修改某个状态类的行为也需修改对应类的源代码。

使用场景： 1、行为随状态改变而改变的场景。 2、条件、分支语句的代替者。

注意事项：在行为受状态约束的时候使用状态模式，而且状态不超过 5 个。

策略模式

策略模式是一种行为模式，它定义了一组算法，并将每个算法都封装起来，使得它们可以互换。该模式的核心思想是将算法的实现从它们的使用中解耦，从而使得算法可以独立地变化。

在策略模式（Strategy Pattern）中，一个类的行为或其算法可以在运行时更改。这种类型的设计模式属于行为型模式。

在策略模式中，我们创建表示各种策略的对象和一个行为随着策略对象改变而改变的 context 对象。策略对象改变 context 对象的执行算法。

意图：定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。

主要解决：在有多种算法相似的情况下，使用 if...else 所带来的复杂和难以维护。

何时使用：一个系统有许多许多类，而区分它们的只是他们直接的行为。

如何解决：将这些算法封装成一个一个的类，任意地替换。

关键代码：实现同一个接口。

应用实例： 1、诸葛亮的锦囊妙计，每一个锦囊就是一个策略。 2、旅行的出游方式，选择骑自行车、坐汽车，每一种旅行方式都是一个策略。 3、JAVA AWT 中的 LayoutManager。

优点： 1、算法可以自由切换。 2、避免使用多重条件判断。 3、扩展性良好。

缺点： 1、策略类会增多。 2、所有策略类都需要对外暴露。

使用场景： 1、如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为。 2、一个系统需要动态地在几种算法中选择一种。 3、如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。

注意事项：如果一个系统的策略多于四个，就需要考虑使用混合模式，解决策略类膨胀的问题。

模板方法模式

在模板模式（Template Pattern）中，一个抽象类公开定义了执行它的方法的方式/模板。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行。这种类型的设计模式属于行为型模式。

意图：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

主要解决：一些方法通用，却在每一个子类都重新写了这一方法。

何时使用：有一些通用的方法。

如何解决：将这些通用算法抽象出来。

关键代码：在抽象类实现，其他步骤在子类实现。

应用实例： 1、在造房子的时候，地基、走线、水管都一样，只有在建筑的后期才有加壁橱加栅栏等差异。 2、西游记里面菩萨定好的 81 难，这就是一个顶层的逻辑骨架。 3、spring 中对 Hibernate 的支持，将一些已经定好的方法封装起来，比如开启事务、获取 Session、关闭 Session 等，程序员不重复写那些已经规范好的代码，直接丢一个实体就可以保存。

优点： 1、封装不变部分，扩展可变部分。 2、提取公共代码，便于维护。 3、行为由父类控制，子类实现。

缺点：每一个不同的实现都需要一个子类来实现，导致类的个数增加，使得系统更加庞大。

使用场景： 1、有多个子类共有的方法，且逻辑相同。 2、重要的、复杂的方法，可以考虑作为模板方法。

注意事项：为防止恶意操作，一般模板方法都加上 final 关键词。

访问者模式

访问者模式是一种行为模式，它定义了一种在不改变对象结构的前提下，对对象中的元素进行操作的方法。该模式的核心思想是将操作从元素中分离出来，并将其封装成一个访问者对象，从而使得元素可以独立地变化

意图：主要将数据结构与数据操作分离。

主要解决：稳定的数据结构和易变的操作耦合问题。

何时使用：需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，使用访问者模式将这些封装到类中。

如何解决：在被访问的类里面加一个对外提供接待访问者的接口。

关键代码：在数据基础类里面有一个方法接受访问者，将自身引用传入访问者。

应用实例：您在朋友家做客，您是访问者，朋友接受您的访问，您通过朋友的描述，然后对朋友的描述做出一个判断，这就是访问者模式。

优点： 1、符合单一职责原则。 2、优秀的扩展性。 3、灵活性。

缺点： 1、具体元素对访问者公布细节，违反了迪米特原则。 2、具体元素变更比较困难。 3、违反了依赖倒置原则，依赖了具体类，没有依赖抽象。

使用场景： 1、对象结构中对象对应的类很少改变，但经常需要在此对象结构上定义新的操作。 2、需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，也不希望在增加新操作时修改这些类。

注意事项：访问者可以对功能进行统一，可以做报表、UI、拦截器与过滤器。

技术选型

- **技术选型就是项目决策**
- 受制于时间、范围、成本的约束
    - 项目金三角（时间、范围、成本），在决策时不能超出这三者的边界。
    - 比如说项目时间紧时，决策上就要偏向于能提升开发速度的技术；
    - 在成本吃紧的情况下，要多用成熟的免费的框架、工具；避免用贵的商业软件或者自己造轮子提升成本；
    - 在范围大、需求多的情况下，架构就要考虑如何能用简单的方法快速完成需求。
    - 还要注意一个问题就是随着项目的推进，制约项目的三个因素一直在动态变化（比如人员流动），需要及时根据情况调整技术决策。
- 要分析可行性和风险
    - 如果在项目决策时，不考虑可行性，不预估风险，就极有可能导致决策失败

- 要考虑利益相关人
    - 在做项目的决策时，如果决策时没有人代表利益相关的人，就可能会做出不考虑他们利益的决策。
    - 比如说光顾着选用新酷的技术，而没有考虑客户的利益，导致成本增加，进度延迟；
    - 比如在选择 UI 组件时，只想着哪个调用方便，而不考虑产品经理的利益，导致产品体验不好。
- **决策中常见的坑**
    - 把听到的观点当事实
    - 先入为主，有结论后再找证据：在做技术选型或者项目决策时，还有一个问题就是可能心中已经有了答案，后面所谓的决策，不过是寻找有利于自己答案的证据。

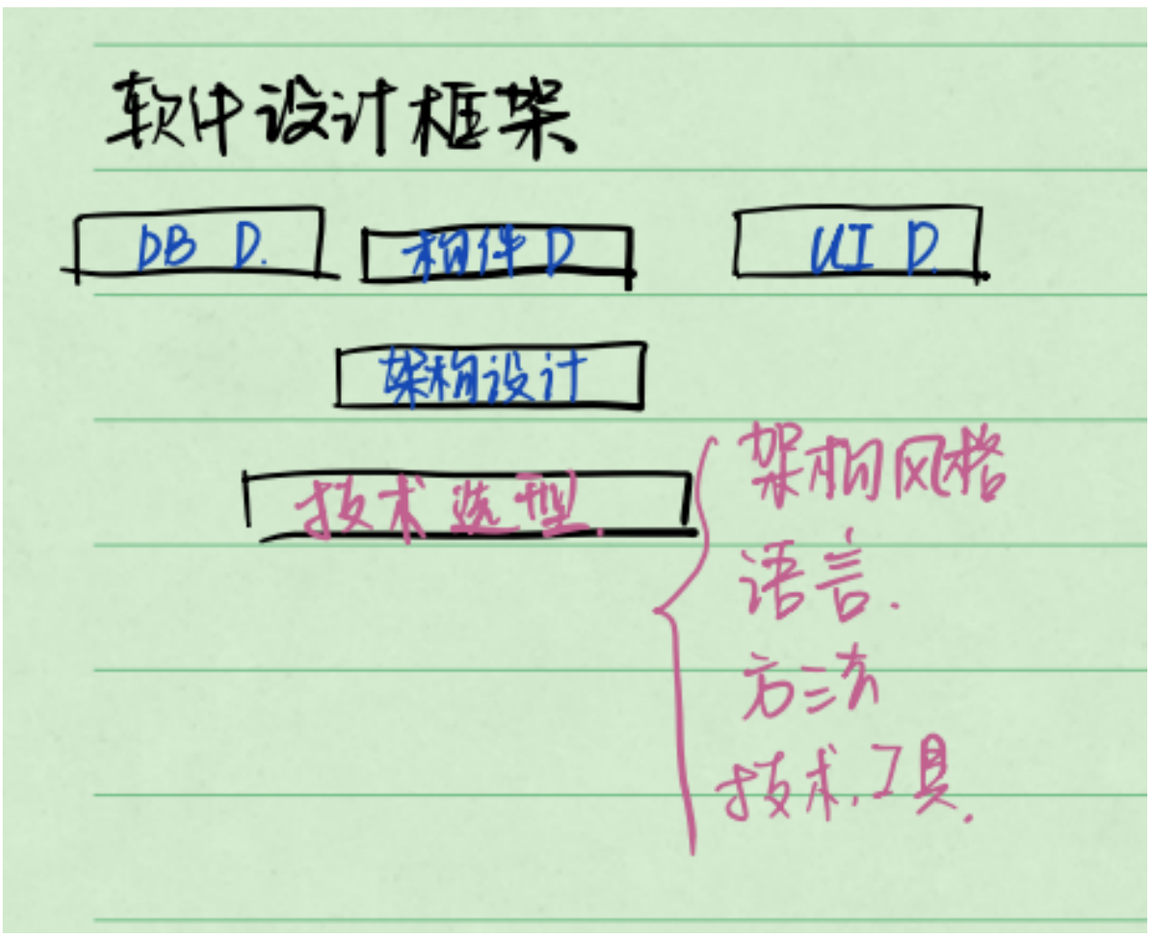
选型要求

1. 项目需求：**选择的技术方案必须能够满足项目需求**，包括性能、安全性、可靠性、可扩展性、易用性等方面。
2. 技术成熟度：选择的**技术方案必须是成熟稳定的，具有广泛的应用和良好的社区支持**，以便于开发和维护。
3. 开发人员技能：选择的**技术方案必须与开发人员的技能和经验相匹配**，以保证项目的开发质量和进度。
4. 项目预算：选择的**技术方案必须能够在项目预算范围内实现**，同时也要**考虑到项目的长期维护和升级成本**
5. 要考虑利益相关人
6. 要分析可行性和风险
7. **考虑项目金三角（时间、范围、成本），在决策时不能超出这三者的边界**

详细选型方案

软件设计框架：

【架构风格；语言；方法；技术；工具】



需求分析、设计、编码、测试、部署和维护

软件开发生命周期（Software Development Life Cycle，SDLC）：指软件从需求分析、设计、编码、测试到部署和维护的整个过程。

- 需求分析：它涉及到对用户需求的了解以及需求的规范化。UML（统一建模语言）是一个非常受欢迎的技术。
- 设计：考虑系统的性能、稳定性以及开发的难易程度。目前常见的软件架构有MVC、分布式架构、微服务架构、SOA（面向服务架构）等。技术选型可以根据系统的具体要求进行选择。**【架构风格】**
- 编码：编程语言、框架、数据库和存储方案，以及前端和后端技术等；适合该系统的编码规范和代码库，开发工具和集成开发环境**【语言、技术、工具】**
  - 语言的特性、性能、可维护性和社区支持等因素
  - 框架考虑其适用性、易用性
- 测试：考虑其适用性、易用性、测试覆盖率和报告生成**【方法】**
  - 考虑如何进行性能测试和安全测试，以及如何进行开发者和用户反馈的收集和处理
- 部署：
  - 使用Docker来部署我们的应用程序，使用Kubernetes来进行应用程序管理和协调
  - 选择适合该系统的云计算和容器技术，以及选择适合该系统的监控和日志分析工具等
- 维护：
  - 选择版本控制工具、错误监控工具、日志分析工具和性能分析工具等；
  - 选择适合该系统的代码重构和性能优化方法
- **此外还需要设计师有如下技能**
  - 硬技能：【高智商：使用；选择；设计】
    1. 了解各种编程语言的特点和优劣，熟悉各种开发框架、库和工具的**使用方法和限制。**
    2. 熟悉各种常用的数据库、网络和安全技术，能够**根据项目需求选择适合的技术方案。**
    3. 具备一定的系统架构设计和软件工程实践经验，能够**根据项目需求进行系统设计和模块划分。**
  - 软技能：【高情商：沟通、说服、均衡】
    1. 能够与团队成员和客户进行**良好的沟通和协作**，理解他们的需求和意见，以便于做出合理的技术选型决策。
    2. 能够根据实际情况进行技术**决策的权衡和取舍**，做出合理的技术选型决策。
    3. 能够**向团队成员和客户清晰地解释**和说明技术选型的理由和优劣，以便于获得他们的认可和支持。

如何选型

问题定义、调研、验证、决策

- 在问题定义阶段，需要搞清楚两个问题：为什么需要技术选型？技术选型的目标是什么？只有明确了技术选型的目标，才能有一个标准可以来评估应该选择哪一种方案
- 在明确技术选型的目标后，就可以去调研，看看哪些技术选型可以满足目标，包括开源的方案和商业的方案。

  在调研时，可以参考前面"项目决策的特点"中的内容，从几个方面去分析：

  - 满足技术选型目标吗？
  - 满足范围、时间和成本的约束吗？
  - 是不是可行？
  - 有什么样的风险？风险是不是可控？
  - 优缺点是什么？

  在调研结束后，可以筛选掉明显不合适的，最终保留 2-3 种方案留待验证。必要的话，可以一起讨论，最终确认。

- **【先验证后使用】**一个技术是不是合适，如果不够了解，没有应用过的话，实际用一下是很有必要的。可以通过一个小型的快速原型项目，用候选的技术方案快速做出一个原型来，做的过程才能知道，你选择的技术选型是不是真的能满足技术选型的目标。
- 在调研和验证完成后，就可以召集所有利益相关人一起，就选择的方案有一个调研结果评审的会议，让大家提出自己的意见，做出最终的决策。

  必须要承认，对于技术选型来说，是有不确定性的。即使通过上面的流程，也一样可能会做出错误的决策。但有一个科学的流程，至少可以保证提升做出正确决策的概率。

  如果遇到很纠结的情况，就需要负责决策的人来拍板了，这时候其实并不一定有对错，重点的就是做出一个选择，然后按照选择去执行。有时候迟迟不选择、不拍板才是最坏的结果。

  在项目结束后，也要对之前技术选型和项目决策做总结，不断的完善技术选型和项目决策的机制，帮助未来更好的进行决策。

尝试

1. 软件是一种计算机程序，用于实现特定的功能或解决一定的问题。
2. 没有万能的模型
3. 应用时需要调整/裁剪
4. **没有完美的软件**
   - **软件始终变化**
   - **软件复杂度高，不可能0缺陷**
5. 软件价值在于满足用户需求
6. 追求性价比
7. 软件三高：**价值，成本，风险高**
8. 工程师决定了三高
9. 找相对简单的方式来进行
10. 阶段化开发**【里程碑】**
11. 成文（文档）

理念

1. 需求制导（用户为中心）
2. 均衡：用户、投资人、老板、自己的利益**【四方均衡-软工实践】**
   - 解决：满足核心诉求，例如用户的使用需求
3. 分治思想：
   - 软件很复杂；
   - 增量迭代；
   - **敏结方法是一类增量迭代方法**【必考！！！】
     - 一种以迭代、增量和自组织为基础的软件开发方法，强调快速响应需求变化、持续交付和团队协作。
4. 复用：保证质量高；保证工作量少
5. 度的适中适量
6. 拥抱变更，接受现实，面向变更的设计（**现在没未实现**）
7. 习本学：学习文档和demo

矛盾

1. 大与小

    - 大小有别
    - 规模改变一切，其背后是复杂度
2. **人与月**
    - 人力 vs 时间
    - **如何处理?**
        - 在项目开始前，进行充分的需求分析和技术评估，确定项目的重点和难点，并合理评估项目所需时间和人力资源。
        - 根据项目的需求和时间安排，规划好人员的分配和安排，确保团队成员的技能匹配和协作效率。
        - 在项目进行过程中，及时调整人员的分配和安排，确保人员的利用率达到最大。
        - 增加人手的时间尽量前移，即在项目开始阶段就增加人手，通过并行开发等方式提高开发效率，减少项目开发时间。
        - 建立有效的沟通机制，包括开发团队内部的沟通和与客户的沟通，及时解决问题和调整开发计划。
        - 让新来的成员写文档，而不参与核心项目构建，避免污染
        - 确保团队成员的技能匹配：为确保团队成员的技能匹配，可以在项目开始前对团队进行评估和培训，提高团队的整体技能水平。
        - *控制需求变更：在软件开发过程中，需求变更是不可避免的，但是需要合理管理需求变更，并尽可能避免影响项目进度。
        - *优化开发流程：通过采用敏捷开发方法、持续集成等方式，优化软件开发流程，从而提高开发效率，减少沟通和协调成本。
3. 多与少
    - 功能少 vs 多
    - 我们开发者期望少，用户期望多
    - 解决：去除非核心功能
4. 新与旧
    - 不可忽略新技术的风险
    - 合适的入手点
5. 硬与软
    - 区别对待约束
6. 独与群
    - 个体 vs 团队
    - 都要选
7. 点与面
    - 点面结合，做好所有环节
    - 需求审查不跳过
    - 个人能力要匹配，不能一颗老鼠屎坏了一锅粥
8. 胖与瘦
    - 团队规模小：容错性差，一个人生病就完了
    - 团队规模大：容错性高，但是人均效率低，产品成本高
    - 我会选择大团队，锻炼能力

论软件复杂性*

软件复杂性是指由软件系统本身及其开发、维护和演化所带来的各种复杂性因素所构成的综合体。软件复杂性是软件工程领域中的一个重要问题，因为它对软件开发、测试、部署和维护等方面都有很大的影响。

软件复杂性的主要原因包括以下几个方面：

1. 功能复杂性：软件系统的功能复杂性是指系统需要实现的各种功能之间的关系和交互，以及系统的功能规模和复杂度。

2. 数据复杂性：软件系统的数据复杂性是指系统需要处理的各种数据类型和数据结构的复杂度，以及数据之间的关系和交互。
3. 接口复杂性：软件系统的接口复杂性是指系统与外部系统或组件之间的接口和协议的复杂度，包括数据格式、通信协议、错误处理和容错机制等。
4. 体系结构复杂性：软件系统的体系结构复杂性是指系统的组件和模块之间的关系和交互，以及系统的层次结构、分层结构和模块化程度等。
5. 开发过程复杂性：软件系统的开发过程复杂性是指开发团队需要进行的各种活动和决策，包括需求分析、设计、编码、测试、部署和维护等。
6. 维护复杂性：软件系统的维护复杂性是指在系统上线后，开发团队需要进行的各种维护活动和修复决策，包括版本控制、缺陷管理、代码重构和性能优化等。

软件复杂性是一个非常重要的问题，因为它会对软件系统的开发、测试、部署和维护等方面产生负面影响。为了降低软件复杂性，开发团队需要采取一系列措施，例如使用合适的设计模式、架构风格和编码规范，以及使用自动化测试和质量保证工具等。同时，开发团队还需要积极学习和应用新的技术和方法，以便更好地管理和降低软件复杂性，提高软件开发效率和质量。

需求分析*

一.需求分析的过程

需求过程包括需求开发和需求管理2个部分：

(1)需求开发就是对开发前期的管理，与客房的沟通过程，可以分为4个阶段：需求获取、需求分析、编写需求和需求验证。

(2)需求管理：就是在软件项目开发过程中控制和维持需求约定的活动。包括：变更控制、版本控制、需求跟踪、需求状态跟踪。

二.需求的层次

需求的层次包括：业务需求、用户需求、功能需求、非功能需求等4个方面。

三.需求开发阶段的重点

(1)提取业务对象

业务对象是指系统使用的真实对象，例如一个供应链管理业务对象主要包括：生产批发商、零售商、送货商、顾客多个层次。

(2)提取业务流程

在了解业务逻辑的过程中，应该列举出所开发软件模块的各自职能，并细化每个工作流程，深入分析业务逻辑。

(3)性能需求

在分析的前期应该注意客户对所开发软件的技术性能指标，如存储容量限制、运行时间限制、安全保密性等。

(4)环境需求

环境需求是指软件平台运行时所处环境的要求，如硬件方面：机型、外部设备、数据通信接口;软件方面：系统软件，包括操作系统、网络软件、数据库管理系统方面;使用方面：使用部门在制度上，操作人员的技术水平上应具备怎样的条件。

(5)可靠性需求

对所开发软件在投入运行后发生故障的概率，应该按实际的运行环境提出要求。对于重要的软件，或是运行失效会造成严重后果的软件，应提出较高的可靠性要求。

(6)安全保密要求

在需求分析时应当在这方面恰当地做出规定，对所开发的软件给予特殊的设计，使其在运行中，其安全保密方面的性能得到必要的保证。

(7)用户界面需求

为用户界面细致地规定到达的要求。

(8)资源使用需求

开发的软件在运行时和开发时所需要的各种资源。

(9)软件成本消耗与开发进度需求

在软件项目立项后，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。

(10)开发目标需求

预先估计以后系统可能达到的目标，这样可以比较容易对系统进行必要的补充和修改。

四.需求分析的任务

需求分析的主要任务是借助于当前系统的逻辑模型导出目标系统的逻辑模型，其流程如下：

(1)确定对系统的综合需求（功能、性能、运行、扩充需求）

(2)制作产品需求文档(PRD)

(3)分析系统的数据需求（概念模型、数据字典、规范化）

(4)导出目标系统的详细的逻辑模型（数据流图、数据字典、主要功能描述）

(5)开发原形系统

(6)从PRD提取编制软件需求规格说明书（SRS）