CS5351 Software Engineering

1.process

动机 Motivation

项目成员之间并不清楚互相交付的工作内容功能和漏洞以及需求，并且常常会出现成员交替等情况

Project members are not aware of the functionality and gaps and requirements of each other's deliverables, and there is often turnover of members, etc.

需求 Needs

针对软件项目开发中交付过程的困难，可通过以下策略来优化项目任务顺序及信息流动：

- 减小生产负担
- 改进信息传递效率
- 提供从用户期待到交付的可处理性
- 预测及跟踪任务的完成情况和未解决问题
- 提高代码质量
- 跟踪公认可不进行设计或实施的内容
- 尽可能提高新成员的生产效率

The following strategies can be used to optimize the project task sequence and information flow in response to the difficulties of the delivery process in software project development:

- Reducing the production burden
- Improving the efficiency of information transfer
- Providing manageability from user expectations to delivery
- Predicting and tracking task completion and unresolved issues
- Improving code quality
- Track what is recognized as not being designed or implemented
- Maximize productivity of new members

然而BUG仍然是不可避免的

However bugs are still inevitable

软件工程流程 Software Engineering Process

软件工程流程模型是指一系列指导软件工程师和项目经理的流程框架，包含

1. 活动(Activities)
2. 技术(Techniques)
3. 交付物(Deliverables)
4. 工具(Tools)
5. 最佳实践(Best Practices)：使用1-4进行软件系统管理

A software engineering process model is a series of process frameworks that guide software engineers and project managers, containing：

1.Activities (Activities)

2. Technology (Techniques)

3. Deliverables (Deliverables)

4. 4. Tools (Tools)

5. 5. Best Practices (Best Practices): using 1-4 for software systems management

Activities

1. 收集用户需求
2. 设计软件
3. 编程/实施
4. 测试
5. 用户端部属
6. 维护软件
7. 结构管理
8. 项目管理

9. gathering user requirements

2. designing the software

3. 3. programming/implementation

4. 4. testing

5. 5. client-side deployment

6. 6. maintaining the software

7. 7. architecture management

8. 8. project management

Techniques

活动对应技术

1. 收集用户需求--->使用案例、用户案例、使用案例图表、会议

   Gathering user requirements ---> use cases, user cases, use case diagrams, meetings

2. 设计软件--->UML、模式、原则、策略

   Designing software ---> UML, patterns, principles, strategies

3. 编程/实施--->Java、C++、框架、平台

   Programming/Implementation--->Java, C++, frameworks, platforms

4. 测试--->单元测试框架、debugger

   Testing ---> unit testing framework, debugger

5. 用户端部属--->独立软件、插件、app、web服务

   Client-side deployment ---> standalone software, plugins, apps, web services

6. 维护软件--->Bug报告、软件仓库

   Maintaining software ---> Bug reports, software repositories

7. 结构管理--->版本控制

   Structure Management --> Version Control

8. 项目管理--->工作分解结构、任务调度算法

   Project Management ---> Work Breakdown Structure, Task Scheduling Algorithm

Deliverables

技术对应交付物

1. 使用案例、用户案例、使用案例图表、会议--->需求文档

   Use Cases, User Cases, Use Case Diagrams, Meetings ---> Requirements Documentation

2. UML、模式、原则、策略--->设计文档

   UML, Patterns, Principles, Strategies ---> Design Documentation

3. Java、C++--->代码清单及测试脚本

   Java, C++ ---> code list and test scripts

4. 单元测试框架、debugger--->质量保证报告

   Unit testing framework, debugger ---> quality assurance reports

5. 独立软件、插件--->客户端软件

   Standalone software, plug-ins ---> client software

6. Bug报告、软件仓库--->Bug报告、仓库release

   Bug report, repository ---> Bug report, repository release

7. 版本控制--->代码改变、路径、改变历史报告

   Version control ---> code changes, paths, change history reports

8. 工作分解结构、任务调度算法--->项目调度和状态追踪

   Work Breakdown Structure, Task Scheduling Algorithm ---> Project Scheduling and Status Tracking

Tools

技术对应工具

1. 使用案例、用户案例、使用案例图表、会议--->UML工具、需求建模工具、Word

   Use cases, user cases, use case diagrams, meetings ---> UML tools, requirements modelling tools, Word

2. UML、模式、原则、策略--->UML工具、SonarQube

   UML, Patterns, Principles, Strategies ---> UML Tools, SonarQube

3. Java、C++--->Visual Studio/Eclipse

4. 单元测试框架、debugger--->JUnit/ debugger /fuzzer

   Unit testing framework, debugger ---> JUnit/ debugger /fuzzer

5. 独立软件、插件--->安装器、插件安装框架

   Standalone software, plug-ins ---> installer, plug-in installation framework

6. Bug报告、软件仓库--->Bug报告系统

   Bug Reporting, Software Repository --> Bug Reporting System

7. 版本控制--->Apache subversion、git

Version control ---> Apache subversion, git

8. 工作分解结构、任务调度算法--->MS Project/Scrum工具

   Work Breakdown Structure, Task Scheduling Algorithm ---> MS Project/Scrum Tools

瀑布流程模型

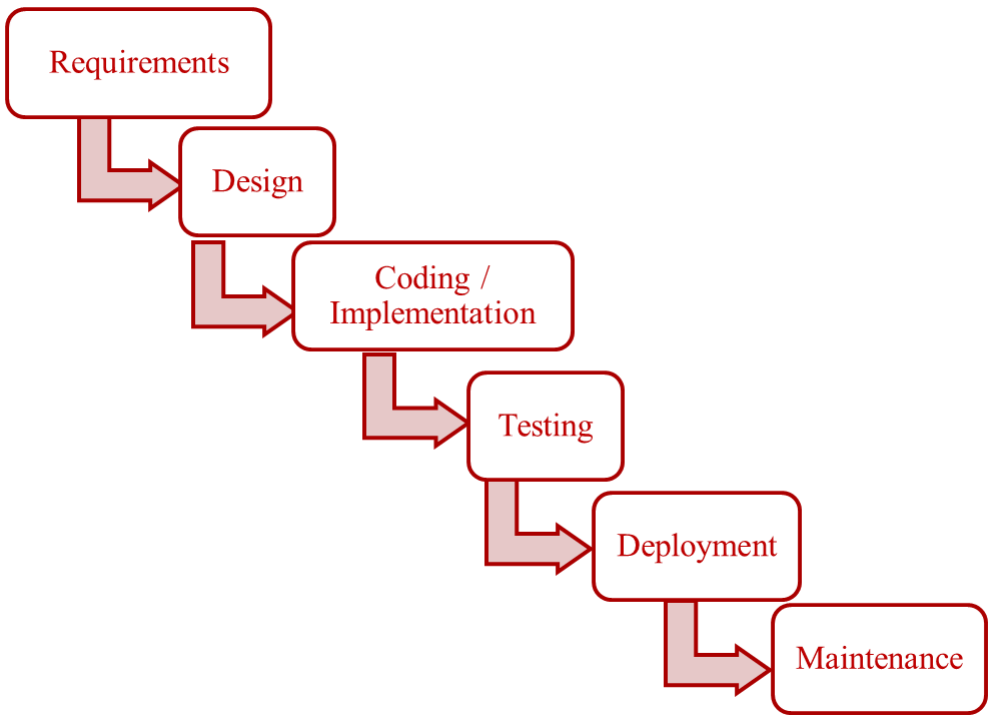
瀑布模型(Waterfall Model)

一种线性的软件开发方法，它的特点是完全完成当前阶段的所有可交付成果后才开始下一个阶段

A linear approach to software development that is characterised by the complete completion of all deliverables of the current phase before starting the next phase

优点在于工作流程简单，缺点在于测试任务繁重，需针对需求、代码及设计统一测试

Advantage is the simple workflow, disadvantage is the heavy testing task, need for unified testing for requirements, code and design.



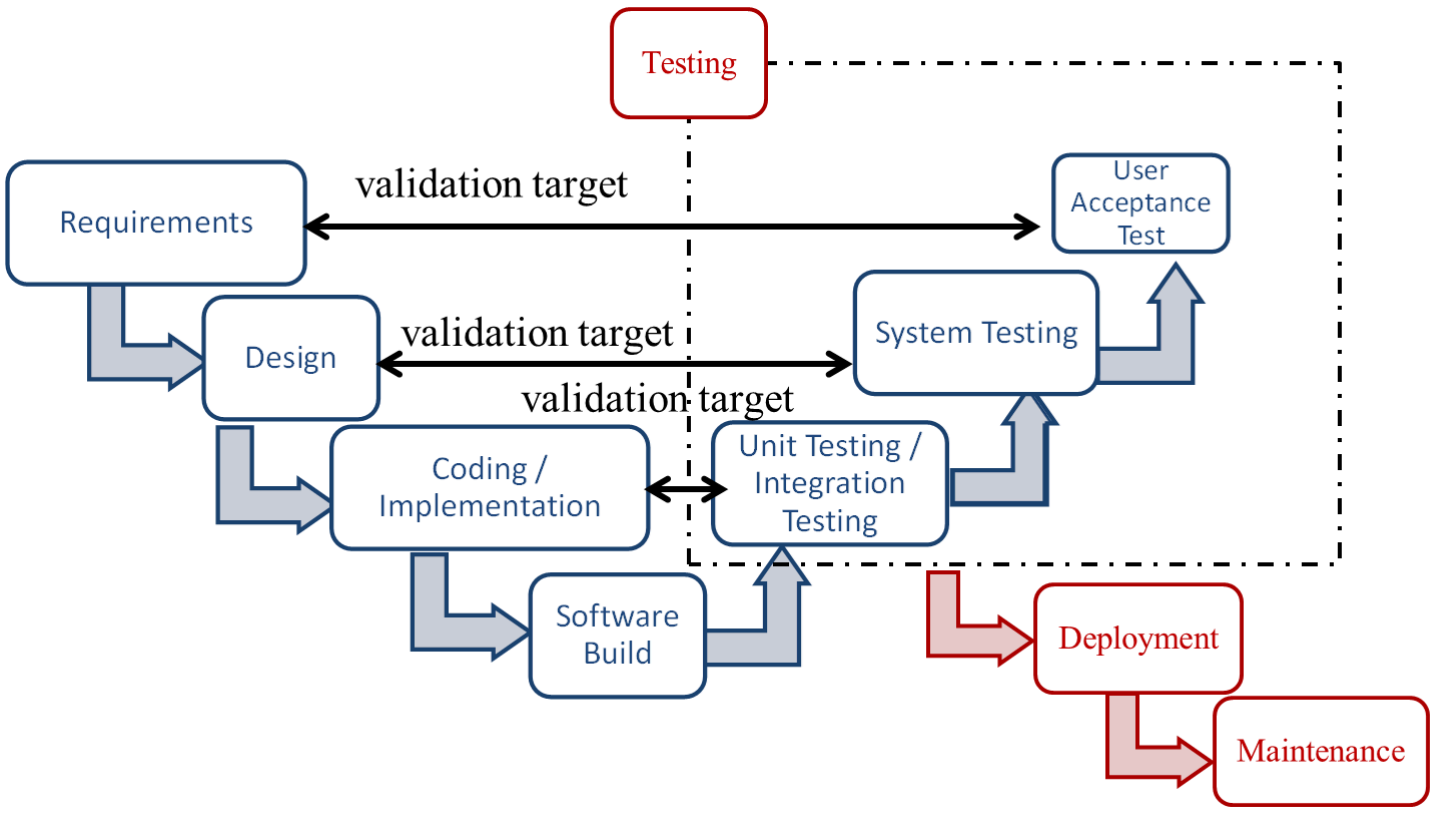
特点(specificities)：

- 线性：完成前阶段后才可进入后阶段
- Linear: complete the first stage before moving on to the second stage
- 详细文档：保证所有人都完全理解各阶段
- Detailed documentation: to ensure that everyone fully understands the stages
- 可预测时间及成本
- Predictable time and cost
- 刚性及不易更改
- Rigid and not easy to change
- 问题发现较晚
- Late detection of problems
- 不适合未定义项目
- Unsuitable for undefined items

V型瀑布模型(V-Shape Waterfall Model)

软件构建阶段每一步都有对应的验证目标

Each step of the software build phase has a corresponding validation objective



两个模型都有共同的缺点：

Both models have common drawbacks:

1. 阶段划分太过清晰 Stages are too clearly delineated
2. 事务完全结束后才开始下一步 The next step is not started until the transaction is completely finished

两个模型提供了经验：早期事务的修缮更便宜 Two models provide lessons learned: repairs are cheaper for early affairs

趋势

许多阶段活动可以同步进行

Many phases of activities can be synchronised

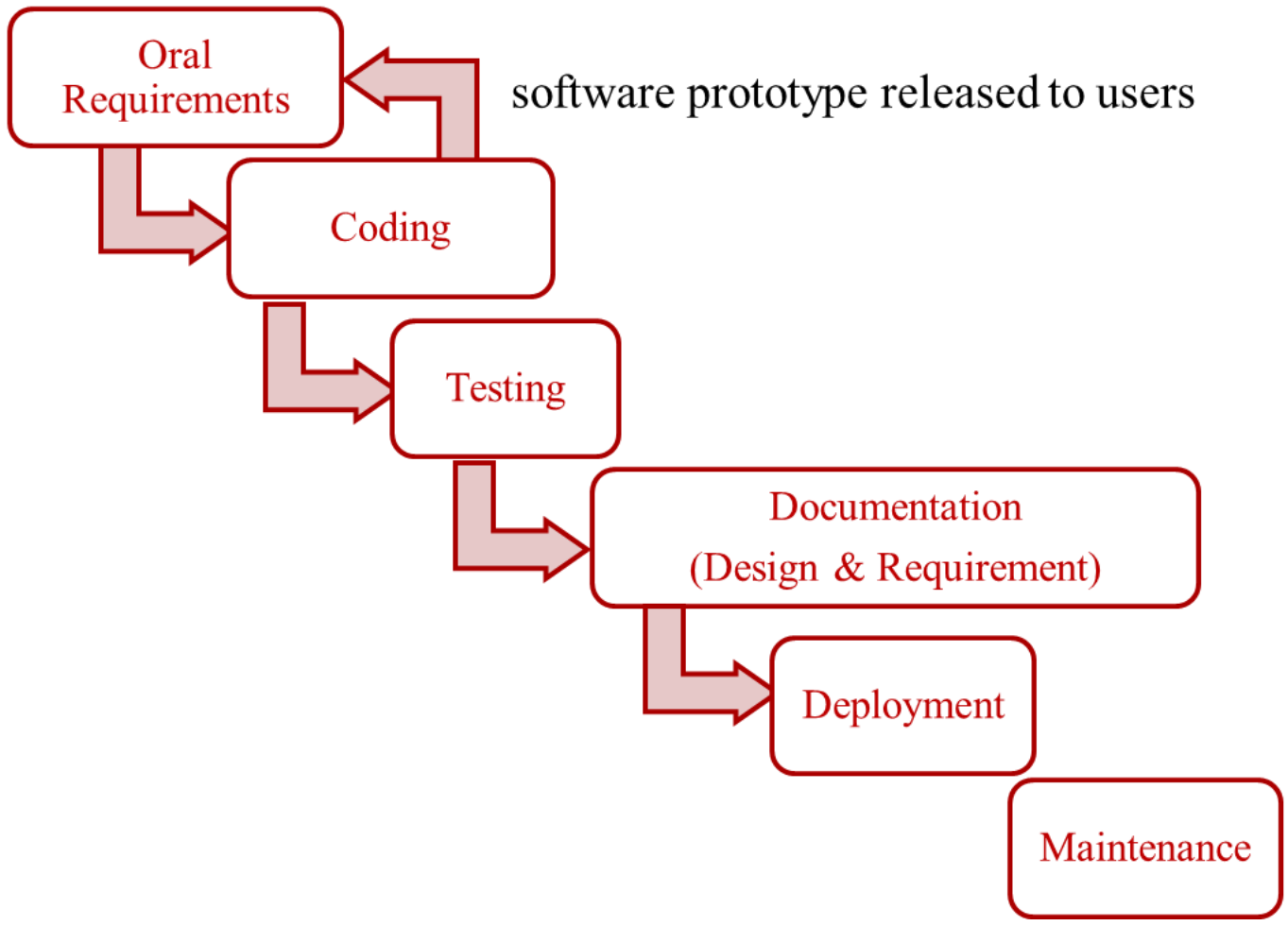
阶段间可以向后反馈

Feedback can be given backwards between stages

原型设计

在进行详细设置之前构建最终软件的原始版本反馈给用户，根据用户意见进行进一步正式设计

Build the original version of the final software to be fed back to the user prior to detailed setup, and further formalise the design based on user comments



敏捷方法(Agile Methods)

敏捷方法强调通过小步快跑的方式向用户提供增量式的工作成果，并且注重与用户的持续沟通和反馈，以便快速适应变化的需求

Agile methods emphasise delivering incremental work products to users in small steps and focus on continuous communication and feedback with users in order to adapt quickly to changing requirements

- 迭代开发 Iterative development
- 客户协作 Customer collaboration
- 对变化响应 Response to changes
- 文档不足 Inadequate documentation
- 不可预测成本及交付时间 Unpredictable delivery time and costs
- 用户参与依赖 Dependence on client involvement

敏捷方法案例——极限编程（Extreme Programming）

五个阶段：

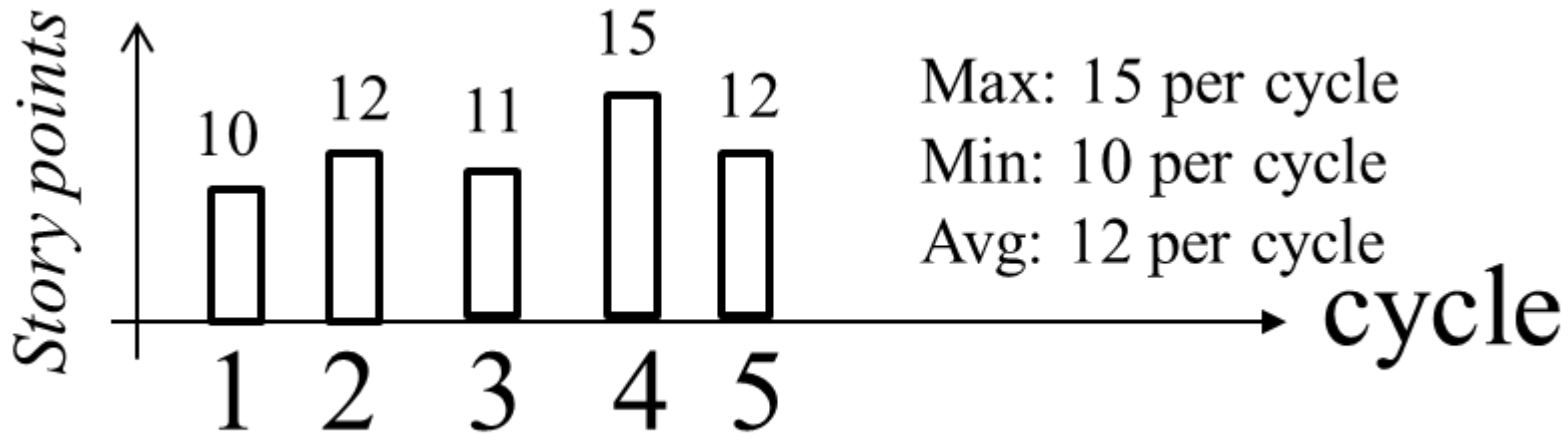
1. 规划：要与用户见面，定义用户故事并估计故事点，计划通过N次迭代来交付用户故事 Planning: to meet users, define user stories and estimate story points, plan to deliver user stories through N iterations
2. 设计：采用简单一致的草图 Design: using simple and consistent sketches
3. 编码：应用XP实践 Coding: Applied XP Practice
4. 测试：自动化单元测试和验收测试 Testing: automated unit testing and acceptance testing
5. 反馈：项目经理和用户评估实施所交付的用户故事的价值 Feedback: project managers and users assess the value of user stories delivered by the implementation

XP实践的重点在于产生反馈、拥抱变化、保持客户参与、短迭代和早期修复bug

XP practices focus on generating feedback, embracing change, keeping customers engaged, short iterations and early bug fixes

项目速度

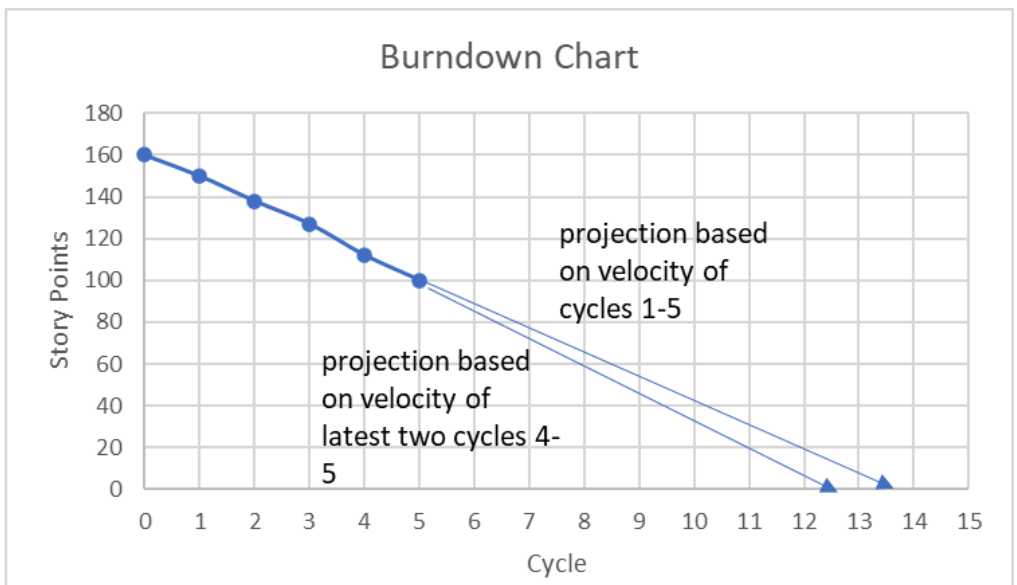
- 用户故事在通过周期审查后尽可能在一个周期内实现 User stories are realised in one cycle as far as possible after reviewing through the cycle
- 敏捷项目是按一系列周期（例如极限编程的迭代或Scrum冲刺）组织的 Agile projects are organised in a series of cycles (e.g. Extreme Programming iterations or Scrum sprints)
- 每个周期实现一组分配了故事点的用户故事 Each cycle realises a set of user stories with assigned story points
- 如果用户故事太大而无法在一个周期内完成，则应将其分解为多个用户故事 If a user story is too large to be completed in one cycle, it should be broken down into multiple user stories
- 在图表中绘制了每个周期提供的总故事点 The total number of story points provided in each cycle is plotted in the graphs



燃尽图(Burndown Chart)

用于预估任务完成时间，其斜率即为项目速度 Used to predict task completion time, the slope of which is the project speed

Cycle	Story Points
0	160
1	150
2	138
3	127
4	112
5	100



在项目实施过程中若出现延迟情况，项目经理应采取相应措施进行处理，例如削减用户故事或要求开发团队提高每周期内完成任务点数量

If there are delays in project implementation, the project manager should take appropriate measures to deal with them, such as cutting back on user stories or asking the development team to increase the number of task points to be completed in a weekly period

Scrum

Scrum是一种敏捷开发过程，目前是最常用的开发过程之一，主要通过每日和定期的一小时会议驱动。在会议上，团队会讨论当前Sprint的目标，优先级功能以及详细的待办事项，并且根据已完成和未完成的任务进行规划和评审会议。Scrum允许客户随时改变需求，强调快速响应和最大化团队的交付能力

Scrum is an agile development process, currently one of the most commonly used development processes, driven primarily through daily and regular one-hour meetings. In the meeting, the team discusses the current Sprint goals, prioritised features and detailed to-do lists, and plans and reviews the meeting based on completed and unfinished tasks.Scrum allows the client to change requirements at any time, with an emphasis on rapid response and maximising

the team's ability to deliver!

- 产品待办列表（包括用户故事和因某些用户故事产生的开发任务） Product to-do list (including user stories and development tasks arising from certain user stories)
- 优先级排序 prioritisation
- 周期长度（如两周）Cycle length (e.g., two weeks)
- 高优先级放入目前Sprint待办 High priority put on current Sprint to-do list
- 成员选择待办事务 Member selection of to-do list
- Sprint结束后进行回顾与评估 Review and evaluate after Sprint

角色 roles：

- 团队教练(Scrum Master)：帮助团队提高性能，不提供日常指导也不分配任务给个人，这个角色通常由项目经理担任 Helps the team improve performance without providing day-to-day guidance or assigning tasks to individuals; this role is usually filled by the project manager
- 产品负责人：负责在Scrum开发过程中对待办事项进行优先级排序 Product Owner: responsible for prioritising to-do's in the Scrum development process
- 开发团队：作为整体 Development team: as a whole

2. Modern Code View

What is Code Review

由代码作者以外的开发者对源代码进行人工检查，是一种成熟的软件工程实践

Manual inspection of source code by developers other than the code author is a well-established software engineering practice

旨在维护和提高源代码质量，并通过其他人的设计和实施解决方案的知识转移来维持开发社区

Aims to maintain and improve the quality of the source code and sustain the development community through knowledge transfer from others who design and implement solutions

代码审查有许多其他目的，包括代码改进、替代解决方案、知识转移、团队建设等等

Code reviews serve many other purposes, including code improvement, alternative solutions, knowledge transfer, team building, and more!

Different Levels of Formality

- **Code walkthrough**（代码走查）：这是一个非正式会议，程序员带领评审团队浏览他的/她的代码，评审员试图找出错误。如果程序员和评审员不是同一个人，则效果更好。有时会结合配对编程。 This is an informal meeting where the programmer leads the review team through his/her code and the reviewer tries to find the bugs. It works better if the programmer and the reviewer are not the same person. Sometimes it is combined with pair programming.
- **Code Inspection**（代码检查）：这是一种正式会议，用于有效地查找代码中的错误。 This is a formal meeting used to efficiently find errors in the code.
- **Modern Code Review**（现代代码审查）：通常指的是利用在线工具和其他技术进行的更加结构化和系统化的代码审查过程。 This usually refers to a more structured and systematic code review process using online tools and other technologies.

Code walkthrough

- Result from an old paper :根据一篇旧论文的结果，代码走查在找寻bug方面的效果与测试类似，但是需要更多人力投入，因此效率较低 According to the results of an old paper, code walk-throughs are similar to tests in finding bugs, but require more human input and are therefore less efficient
- 近年来，开发者开始意识到技术债务问题（非bug问题），而这些问题使用测试往往难以解决，例如可读性、可重用性和可维护性等问题。 In recent years, developers have become aware of technical debt issues (non-bug issues) that are often difficult to resolve using tests, such as readability, reusability and maintainability issues.
- 由 “居家办公 ”需求引发的远程协作使视障开发人员可以远程参与代码演练。 Remote collaboration, triggered by the need to "work from home," allows visually impaired developers to participate in code walkthroughs remotely.

Code Inspection

- 代码检查是一个高度结构化和正式的审查过程。在某些公司里，它也被称为静态测试。这种审查方式对于发现bug和改进流程非常有效。尽管如此，它的受欢迎程度正在下降。 Code checking is a highly structured and formal review process. In some companies it is also known as static testing. This type of review is very effective in finding bugs and improving processes. Nevertheless, its popularity is declining.
- 在检查过程中需要注意的事项包括：算法的效率、代码的逻辑和正确性以及注释的一致性。 Things to look for during the checking process include: efficiency of the algorithm, logic and correctness of the code, and consistency of the comments.

主要思想 main ideas：

- 会前：作者教育评审员，评审员从测试角度阅读/提问代码 Pre-meeting: authors educate reviewers, reviewers read/question code from a testing perspective
- 会中：由代码作者带领评审员一起走查代码，并对照检查表进行检查。回答问题并消除歧义。每个评审员在预会期就被分配了特定的角色（例如，负责检查某个特定领域）In-meeting: the code author leads the reviewer in a walk-through of the code and checks it against a checklist. Questions are answered and ambiguities are removed. Each reviewer is assigned a specific role during the pre-session (e.g., responsible for checking a specific area)
- 会后：作者跟进并纠正代码。主持人核实所要求的纠正措施是否正确 Post-session: authors follow up and correct the code. Facilitator verifies correctness of requested corrective action

Modern Code Review (MCR)

MCR是一种轻量级、基于工具的代码变动内容审查，而不是对整个代码进行审查，已经成为许多软件开发项目的规范。

MCR is a lightweight, tool-based review of the content of code changes, rather than a review of the entire code, and has become the norm for many software development projects.

MCR的特点包括非正式、基于工具的物流管理、异步审查、专注于审查代码变更，比代码检查更加轻便。

MCR features include informal, tool-based logistics management, asynchronous review, and a focus on reviewing code changes that is lighter than code inspection.

主要思想 main ideas：

- 作者**A**制作了一个补丁**P**用于修复代码块**C**中的问题，并通过电子渠道（如电子邮件、WhatsApp、Wechat或GitHub工具）发送给一组评审员R。 The author **A** produced a patch **P** for fixing the problem in code block **C** and sent it to a group of reviewers R via electronic channels (e.g., email, WhatsApp, Wechat, or GitHub tools).
  - 每位评审员**R**评估**P**，认为**P**良好或者拒绝**P**。 Each assessor **R** assesses **P** and finds **P** favourable or rejects **P**.

    - 作者**A**提交**P**，其他评审员可能在提交后继续审查。 Submitted by author **A P**, other reviewers may continue to review after submission
- 作者和评审员交流想法，查找错误，并讨论更好的设计方案。 Authors and reviewers share ideas, look for errors, and discuss better design solutions.

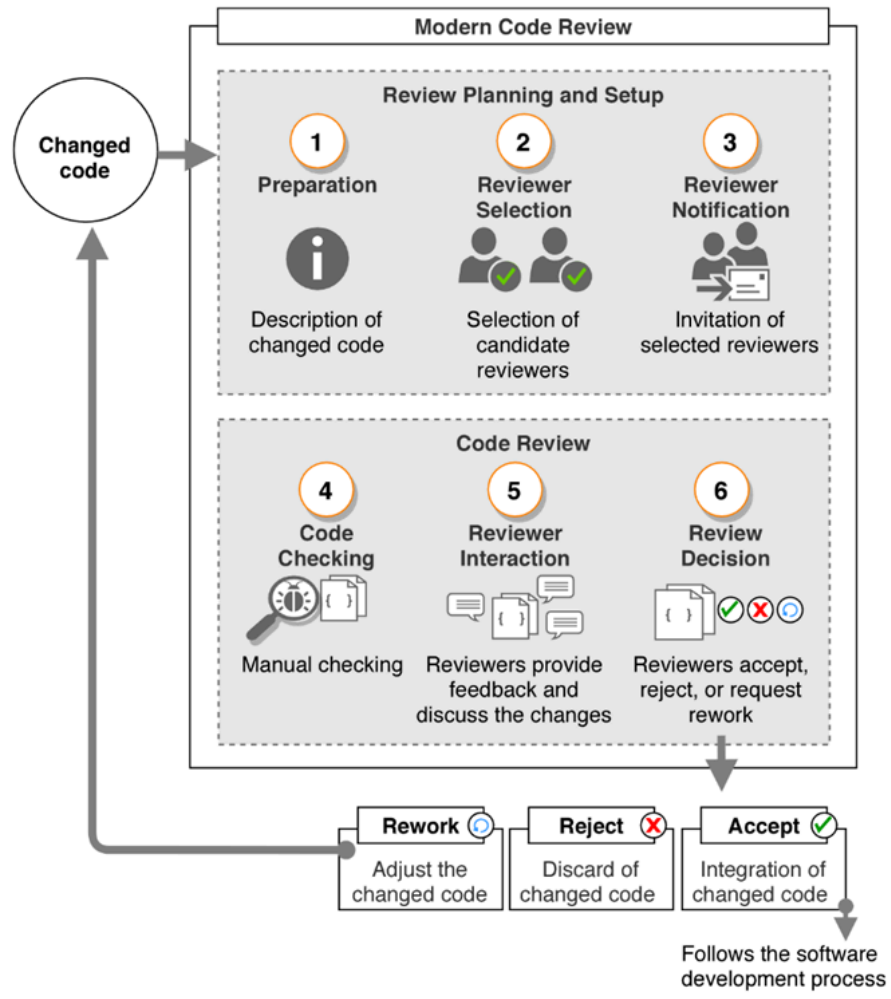


Fig. 1. An overview of the tasks of modern code review.

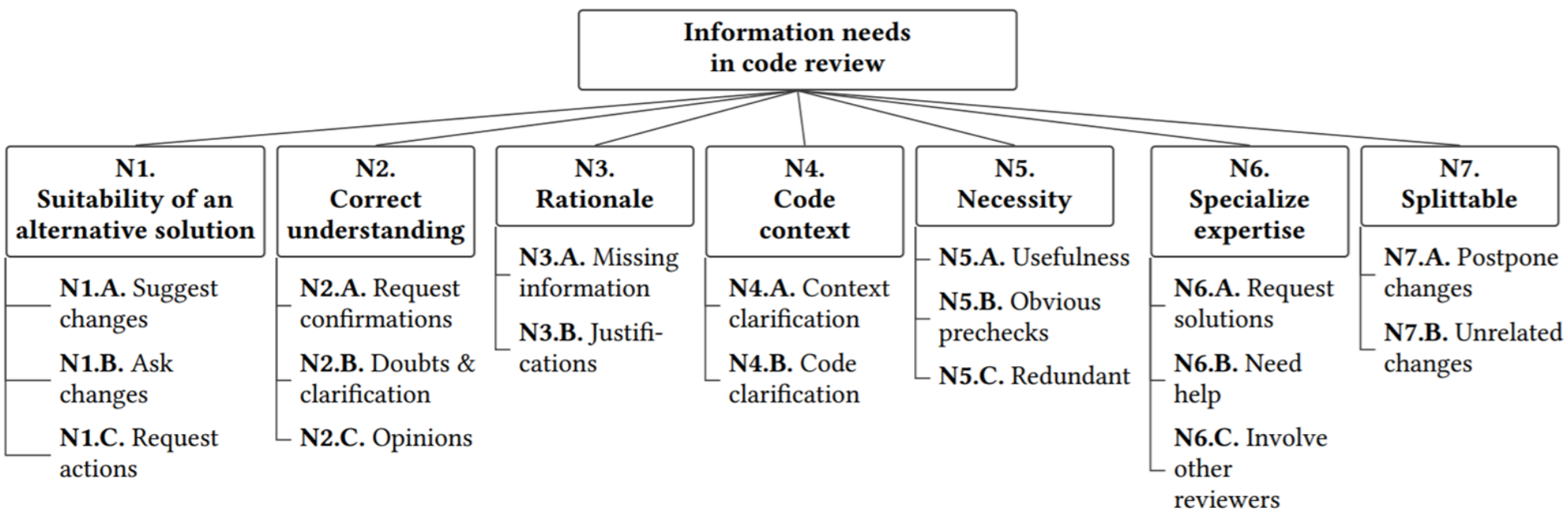
特点：

非正式(informal)：没有预先定义的角色，审查过程不遵循正式程序

基于工具的物流(Tool-based logistics)：电子邮件、WhatsApp和WeChat是常用的工具。开源工具包括GitHub、Gerrit（用于Git）、ReviewBoard和Phabricator。

异步(Asynchronous)：作者和评审员不需要同时处理任务（但不应过度延迟） Authors and reviewers do not need to work on tasks at the same time (but should not be unduly delayed)

专注于补丁(patch)，即代码 Focus on patches, i.e., code



然而并不是进行越多轮次的MCR就越能提高代码质量 However, it is not the case that the more rounds of MCR you do, the more you improve the quality of your code.

MCR最佳实践：

- 审查遵循轻量级、灵活的过程

  The review followed a lightweight, flexible process

- 在变更提交之前尽早进行审查，并且频繁地进行

  Review changes early before they are submitted and frequently

- 审阅者事先了解上下文和代码，能更快地完成审阅，并向作者提供更有价值的反馈意见

  Reviewers with prior knowledge of the context and code can complete reviews faster and provide more valuable feedback to authors

- 减小变更规模

  Reducing the scale of change

- 审核员人数少（1 到 5 人）

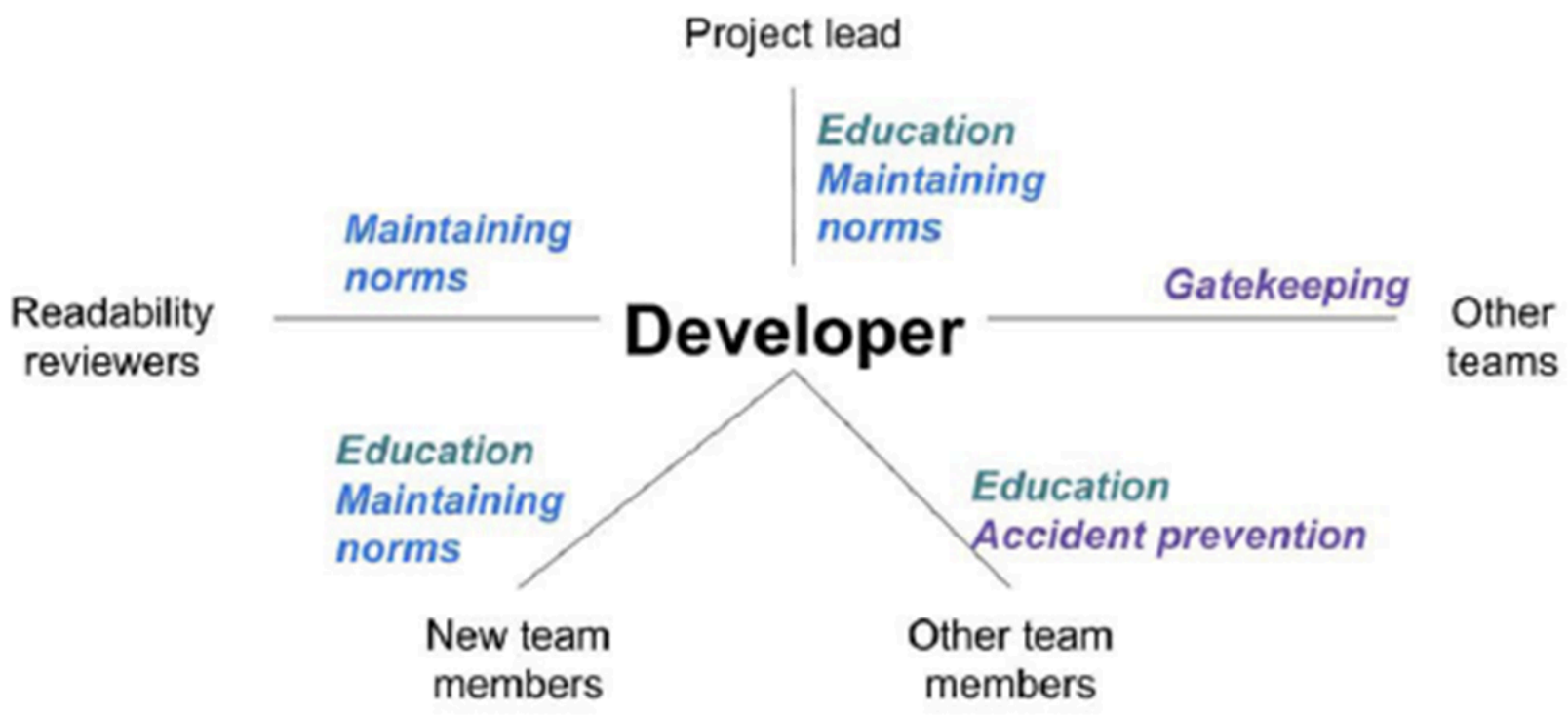
  Small number of reviewers (1 to 5)

- 阅读关于代码和补丁的非正式帖子和评论

  Read unofficial posts and comments about codes and patches

- 审阅不仅仅是为了在小组层面发现错误：解决问题（解决方案开发）、代码可读性和可维护性、遵循规范（所属公司的规范）、预防事故、把关（还记得主持人在代码检查中的作用吗？）

  Reviewing is not just about catching bugs at the group level: problem solving (solution development), code readability and maintainability, adherence to specifications (those of the company to which it belongs), prevention of accidents, gatekeeping (remember the facilitator's role in code checking?).



审查者可以由以下成员担任：

The reviewer may be one of the following members:

- 代码所有者

  Code Owner

- 对代码做出过先前更改的开发者

  Developers who have made previous changes to the code

- 以前开发此代码的开发者（例如，通过配对编程产生代码）

  Previous developers who developed this code (e.g., generated code through pair programming)

- 经验丰富的审查者

  Experienced reviewers

- 开发其他代码中类似功能的开发者

  Developers who develop similar functionality in other codes
- 等等

可能存在的问题： Possible problems:

- 审查质量和代码大小

  Review quality and code size
  - 复杂度较高的代码获得的讨论和反馈较少

    Higher complexity code gets less discussion and feedback
  - 反馈较少的代码在发布后可能会遇到更多bug

    Code with less feedback may encounter more bugs after release
- 距离 distance
  - 导致审查过程延迟或导致误解

    cause delays in the review process or lead to misunderstandings
    - 方面1：地理问题

      Aspect 1: Geographical issues
      - 作者与审查者之间的物理距离较大

        Greater physical distance between author and reviewer
    - 方面2：组织问题

      Aspect 2: Organisational issues
      - 来自不同团队或承担不同角色的审查者

        Reviewers from different teams or with different roles
- 社交互动

  social interaction
  - 语气：负面语气的评论不太有用

    Tone: comments with a negative tone are less useful
  - 权力：要求他人修改评论。作者不满意

    Power: Ask others to modify their comments. Author dissatisfaction
- 上下文

  (textual) context
  - 因对代码变更的期望不匹配而产生的误解（例如，希望有的特性与紧急修复，完整的解决方案与高层次草图）
    Misunderstandings due to mismatched expectations of code changes (e.g., desired features vs. emergency fixes, complete solutions vs. high-level sketches)

3.Requirements Engineering

Non-functional requirements

仅仅了解功能性需求不足以提供给客户良好的体验，还需要关注非功能性需求例如效率问题和可扩展性问题等

Understanding functional requirements is not enough to provide a good customer experience, but you need to focus on non-functional requirements such as efficiency and scalability issues.

需求工程是一个找出并结构化要构建的软件的功能性和非功能性需求的过程。

Requirements engineering is a process of identifying and structuring the functional and non-functional requirements of the software to be built.

在很多情况下，大部分需求是非功能性需求 In many cases, the majority of the requirements are non-functional requirements

客户可能采用下列词汇来描述非功能性需求： Customers may use the following terms to describe non-functional requirements:

- System properties/characteristics/constraints
- Quality attributes, non-behavioral requirements, concerns, goals, extra-functional requirements, quality requirements, and system attributes

而在不同领域的非功能性需求也有所不同 And non-functional needs vary in different areas

Table 3 - Application Domains and Relevant NFRs

Application Domain	Relevant NFRs
Banking and Finance	accuracy, confidentiality, performance, security, usability
Education	interoperability, performance, reliability, scalability, security, usability
Energy Resources	availability, performance, reliability, safety, usability
Government and Military	accuracy, confidentiality, performance, privacy, provability, reusability, security, standardizability, usability, verifiability, viability
Insurance	accuracy, confidentiality, integrity, interoperability, security, usability
Medical/Health Care	communicativeness, confidentiality, integrity, performance, privacy, reliability, safety, security, traceability, usability
Telecommuni-cation Services	compatibility, conformance, dependability, installability, maintainability, performance, portability, reliability, usability
Transportation	accuracy, availability, compatibility, completeness, confidentiality, dependability, integrity, performance, safety, security, verifiability

Table 2 - The Most Commonly Considered NFRs

NFRs	Definition	Attributes
Performance	requirements that specify the capability of software product to provide appropriate performance relative to the amount of resources needed to perform full functionality under stated conditions	response time, space, capacity, latency, throughput, computation, execution speed, transit delay, workload, resource utilization, memory usage, accuracy, efficiency compliance, modes, delay, miss rates, data loss, concurrent transaction processing
Reliability	requirements that specify the capability of software product to operates without failure and maintains a specified level of performance when used under specified normal conditions during a given time period	completeness, accuracy, consistency, availability, integrity, correctness, maturity, fault tolerance, recoverability, reliability, compliance, failure rate/critical failure
Usability	requirements that specify the end-user-interactions with the system and the effort required to learn, operate, prepare input, and interpret the output of the system	learnability, understandability, operability, attractiveness, usability compliance, ease of use, human engineering, user friendliness, memorability, efficiency, user productivity, usefulness, likeability, user reaction time
Security	requirements that concern about preventing unauthorized access to the system, programs, and data	confidentiality, integrity, availability, access control, authentication
Maintainability	requirements that describe the capability of the software product to be modified that may include correcting a defect or make an improvement or change in the software	testability, understandability, modifiability, analyzability, changeability, stability, maintainability compliance

即使在同一应用领域，不同类型的应用也有不同的非功能性要求

Different types of applications have different non-functional requirements, even within the same application area

- 许多质量属性是未定义或模糊不清的。

  Many quality attributes are undefined or ambiguous.
- 如果一个质量属性无法测量或无法验证，我们就不知道我们的软件在多大程度上满足它。

  If a quality attribute can't be measured or verified, we don't know to what extent our software fulfils it.
- 对于这些质量属性，我们只能依赖用户对软件的评价。

  For these quality attributes, we can only rely on user evaluations of the software.

处理多级关注点需求：

Dealing with multi-level attention span requirements:

- 具体到应用领域

  Specific areas of application
  - 满足行业部门的一般要求

    Meeting the general requirements of the industry sector
- 具体到软件应用程序类型

  Specific to the type of software application
  - 满足此类软件的一般要求

    Fulfilment of general requirements for such software
- 特定的应用程序

  Application-specific
  - 满足当前应用程序的独特要求

    Meet the unique requirements of today's applications

在所有级别中考虑非功能性要求

Consider non-functional requirements at all levels

否则，我们的应用程序将不容易在应用环境中使用

Otherwise, our application will not be easy to use in an application environment

Requirements Engineering (RE)

- 需求工程是解决数据处理问题的第一步。

  Requirements engineering is the first step in solving data processing problems.
- 需求工程的结果是一个需求规范。

  The result of requirements engineering is a requirements specification.
- 需求规范对于客户来说是一份合同，同时也是设计工作的起点。

  The requirements specification is a contract for the client and the starting point for the design work.

简单来说，需求工程就是确定并记录一个系统或产品所需功能的过程，这个过程产生的文档被称为需求规格说明书。它是项目开发的基础，明确了用户的需求以及开发者需要实现的功能。需求规格说明书不仅是与客户达成共识的依据，也是后续设计、编码和测试等阶段的重要参考

Simply put, requirements engineering is the process of identifying and documenting the required functionality of a system or product, and the document that results from this process is known as a requirements specification. It is the foundation of project development and specifies the needs of the users and the functionality that the developers need to implement. The requirements specification is not only the basis for reaching a consensus with the client, but also an important

reference for the subsequent design, coding and testing phases.

在需求工程中的常见错误包括： Common mistakes in requirements engineering include:

- 噪声（noise）：没有添加相关的信息。 No relevant information has been added.
- 沉默（silence）：问题中的特性在说明书中未提到。 The characteristic in question is not mentioned in the specification.
- 过度指定（over-specification）：讨论解决方案而不是问题本身。 Discuss solutions rather than the problem itself.
- 矛盾（contradictions）：描述不一致。 Inconsistent descriptions.
- 模糊不清（ambiguity）：不清楚。 It's not clear.
- 后续引用（forward references）：特别是在长文档中特别繁琐。 Especially cumbersome in long documents.
- 幻想（wishful thinking）：无法实际实现的功能。 Functions that cannot be practically realised.

自然语言规范具有危险性，例如“所有用户都有相同的控制字段”，这句话可能有多种理解方式：

Natural language specifications can be dangerous, e.g. "all users have the same control fields" can be interpreted in many ways:

- 控制字段具有相同值？ Control fields have the same value?
- 控制字段格式相同？ Control fields formatted the same?
- 所有用户共用同一个控制字段？ All users share the same control field?

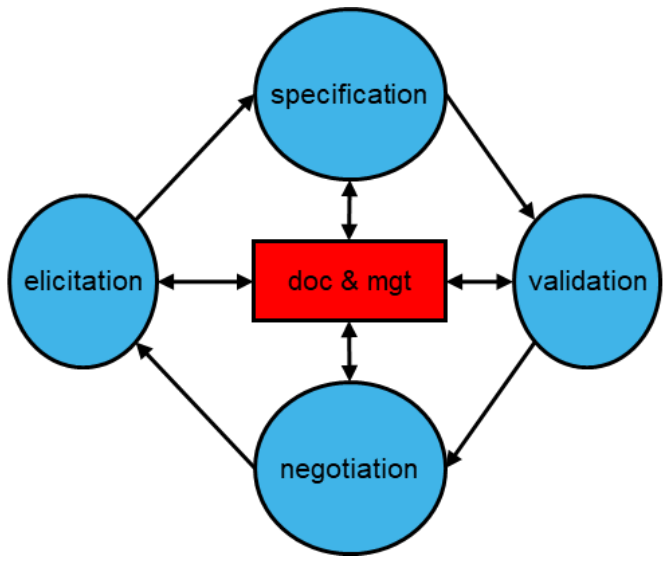
为避免误解，我们需要验证记录下来的要求。这会引导我们进入RE流程。

To avoid misunderstandings, we need to validate the documented requirements. This will lead us to the RE process.

需求工程的主要步骤： The main steps of requirements engineering:

1. 理解问题：收集信息 Understanding the problem: elicitation
2. 描述问题：制定规范 Describing the problem: specification
3. 达成共识：验证 Reaching consensus: validation
4. 达成边界共识：协商 Reaching consensus on the border: consultations

这是一个迭代过程。 It's an iterative process.



The Positions of Analyst in RE

使用概念建模(**Conceptual modeling**)来明确需求范围 Use of Conceptual modelling to define the scope of requirements

- 我们明确地将需求范围建模为领域模型。 We explicitly modelled the scope of requirements as a domain model.
- 明确建模需要解决两个问题： Explicit modelling needs to address two issues:
  - 分析：找到并消除歧义，因为未言明的假设、不同的语言/术语以及对问题领域的不完备编码化。 Analysis: find and remove ambiguities due to unstated assumptions, different language/terminology and incomplete codification of the problem area.
  - 协商：由于利益相关者的目标不同而产生的歧义（例如，工人与管理层之间的分歧）。 Negotiation: Ambiguity due to different objectives of stakeholders (e.g., disagreement between workers and management).
- 作为分析师，我们必须参与塑造领域模型。 As analysts, we must be involved in shaping the domain model.
- 具有明确的模型可以减少后续的意外情况。 Having a clear model reduces subsequent surprises.

分析人员的不同假设： Different assumptions by analysts:

- 主观-客观（关于知识） Subjective-objective (on knowledge)
- 冲突-秩序（关于世界） Conflict-Order (about the world)

导致在需求工程中有四种基本立场 Leading to four basic positions in requirements engineering

1. 功能主义（客观+有序）：分析师是寻求真理的专家，以经验为基础。 Functionalism (objective + ordered): analysts are truth-seeking experts, empirically based.
2. 社会相对主义（主观+有序）：分析师是一个“变革推动者”，需求工程是一个由分析师引导的学习过程。 Social relativism (subjective + ordered): the analyst is a "change agent" and requirements engineering is an analyst-led learning process.
3. 激进结构主义（客观+冲突）：存在阶级斗争；分析师选择支持某一方。 Radical structuralism (objective + conflict): there is class struggle; analysts choose to support one side.
4. 新人本主义（主观+冲突）：分析师像社会治疗师一样，把各方聚集在一起。 Neo-humanism (subjective + conflict): analysts act like social therapists, bringing parties together.

这些立场反映了分析师在处理需求工程任务时的不同态度和方法，涵盖了从技术导向到更关注社会因素和冲突解决等不同角度。

These positions reflect the different attitudes and approaches of analysts when approaching requirements engineering tasks, covering a range of perspectives from technically orientated to a greater focus on social factors and conflict resolution

功能主义（objective+order）：分析师是通过实证方法寻找真相的专家。 Functionalism (objective+order): analysts are experts in finding the truth through empirical methods.

- 冲突由管理解决。 Conflict resolution by management
- 管理决定事情的发展方向。 Management decides where things are going.
- 需求清晰、共享且客观。 Needs are clear, shared and objective.
- 开发是正式的、理性的。 Development is formal and rational.
- 政治无关紧要。 Politics is irrelevant.
- 现实是可以衡量的，对每个人来说都是一样的。 Reality is measurable, for everyone.
- 设计是一个技术问题。 Design is a technical issue.

这是对功能主义立场的一个概述，强调了在这个视角下，需求工程被视为一种基于技术和理性决策的过程，其中冲突由管理解决，现实被认为是可度量且普遍适用的。

This is an overview of the functionalist position, emphasising a perspective in which requirements engineering is seen as a process based on technical and rational decision-making, in which conflicts are resolved by management and reality is considered measurable and universally applicable.

社会相对主义（subjective+order）：分析师是一个“变革推动者”。 Social relativism (subjective+order): the analyst is a "change agent".

- 并不存在唯一的真相，可能存在不同的看法。 There is no single truth and there may be different views.
- 信息系统是不断变化的社会环境的一部分。 Information systems are part of a changing social environment.
- 系统的目标来自组织塑造自己的现实。 The goals of the system come from the organisation shaping its own reality.
- 没有好坏的客观标准。 There are no objective criteria for good or bad.
- 目标是达成共识。The goal is to reach consensus.

社会相对主义认为没有绝对的真理，而是强调了系统目标是由组织自身塑造的，而且在达成共识的过程中，信息系统的目的是适应不断变化的社会环境。

Social relativism argues that there are no absolute truths, but rather emphasises the fact that system goals are shaped by the organisation itself and that in reaching consensus, information systems aim to adapt to the changing social environment.

激进结构主义（objective+conflict）：

- 存在阶级斗争；分析师选择支持某一方。 Class struggle exists; analysts choose to support one side or the other.
- 存在根本的真相，但也存在阶级间的根本斗争。 There is a fundamental truth, but there is also a fundamental struggle between classes.
- 可能选择支持管理层（机器取代人力，机器指导工作，监督，失业，无趣的工作），或者支持工人（提升劳动技能，使工作更具挑战性和趣味性）。 May choose to support management (machines replacing manpower, machines directing work, supervision, unemployment, uninteresting work), or workers (upgrading labour skills, making work more challenging and interesting).

新人文主义（subjective+conflict）：倾向于解放，消除障碍。 The tendency is to liberate and remove obstacles.

Requirements Elicitation

在了解我们在项目的需求工程（RE）过程中所处的位置后，下一步就是收集需求。 After understanding where we are in the Requirements Engineering (RE) process for the project, the next step is to gather requirements.

需求获取过程中的活动可以分为五类： The activities in the demand acquisition process can be divided into five categories:

1. 了解应用领域 Learn about application areas
2. 确定需求来源 Identifying sources of demand
3. 分析利益相关者 Analysing stakeholders
4. 选择使用的技术、方法和工具 Selection of techniques, methods and tools to be used
5. 从利益相关者和其他来源中获取需求 Capturing demand from stakeholders and other sources

因为需求获取是一个迭代过程，所以不能在一个类型完成后再开始另一个类型。 Because requirement acquisition is an iterative process, you can't finish one type and then start another one

steps

1.了解应用领域 Learn about application areas

- 从理解应用领域开始 Start by understanding the application area
- 全面探索当前系统环境，包括与系统相关的政治、组织和社会方面，以及它们可能对系统及其发展施加的任何限制。 A comprehensive exploration of the current system environment, including the political, organisational and social aspects associated with the system, and any constraints they may impose on the system and its development.

这一步需要深入了解系统的背景和环境，以便更好地理解其功能需求。

This step requires an in-depth understanding of the context and environment of the system in order to better understand its functional requirements.

2.识别需求来源 Identify sources of requirements

需求分布在多个来源，并以不同的格式存在。 Requirements are distributed across multiple sources and exist in different formats.

- 人： people
  - 利益相关者：系统的主要需求来源。 Stakeholders: the main source of demand for the system.
  - 用户和主题专家：提供详细信息和需求。 Users and subject matter experts: provide detailed information and requirements.
- 物品： items
  - 如现有系统和操作流程。 such as existing systems and operational processes.
  - 如现有文档（如手册、表格和报告）：提供有关组织、环境和理由的有用信息。 e.g. existing documentation (e.g. manuals, forms and reports): provides useful information about the organisation, the environment and the rationale.

3.分析利益相关者 Analysing stakeholders

- 利益相关者是关注系统的人们。 Stakeholders are the people who care about the system.
- 内部和外部的团体和个人。 Internal and external groups and individuals.
- 客户和直接用户。 Customers and direct users.
- 分析并涉及所有相关利益相关者。 Analyse and involve all relevant stakeholders.
- 分析利益相关者的过程通常还包括确定关键用户代表和支持软件的产品拥护者。 The process of analysing stakeholders usually also includes identifying key user representatives and product champions who support the software.
- 在需求获取期间咨询他们。 Consult them during demand acquisition.

4.选择使用的技术、方法和工具 Selection of techniques, methods and tools to be used

- 使用多种技术进行需求获取效果最佳（例如访谈和表单分析）。 Requirements capture works best using multiple techniques (e.g., interviews and form analysis).
- 许多项目中，在不同软件开发生命周期阶段采用多种需求获取技术。 In many projects, multiple requirements acquisition techniques are used at different software development life cycle stages.

5.从利益相关者和其他来源获取需求 Obtaining demand from stakeholders and other sources

在以上四个步骤后，这一步骤是实际收集需求的步骤 After the four steps above, this step is the actual collection of requirements

目标：targets:

- 确定系统的范围。 Determine the scope of the system.
- 详细调查利益相关者的需求和愿望，尤其是用户的需求。 Investigate in detail the needs and aspirations of stakeholders, especially users.
- 确定系统将执行的未来业务运营过程。 Identify the future business operations processes that the system will perform.
- 检查系统如何支持满足主要目标（并解决业务的关键问题）。 Examine how the system supports meeting the main objectives (and solves the key problems of the business).

这一步骤的目标是深入了解利益相关者的需求，以便为系统设计提供依据。

The goal of this step is to gain a deeper understanding of stakeholder needs in order to inform system design.

Techniques

接下来对几种常用的需求获取技术进行讨论

面谈(interview):

- 常用于需求获取。 Commonly used for requirements acquisition.
- 向利益相关者询问关于当前应用的使用情况以及待建应用的预期用途。 Ask stakeholders about the use of the current application and the intended use of the to-be-built application.
- 能够从利益相关者的角度全面了解他们的需求和现有应用的问题。 Ability to gain a comprehensive understanding of stakeholders' needs and problems with existing applications from their perspective.
- 面试本质上是非正式的，其有效性很大程度上取决于参与者之间的互动质量。Interviews are inherently informal and their effectiveness depends greatly on the quality of the interaction between participants.
- 面试是一种快速收集大量数据的有效方式。 Interviews are an effective way to gather a lot of data quickly.

面试是一种常见且有效的需求获取手段，能够帮助理解利益相关者的需求和观点，并能迅速收集信息。

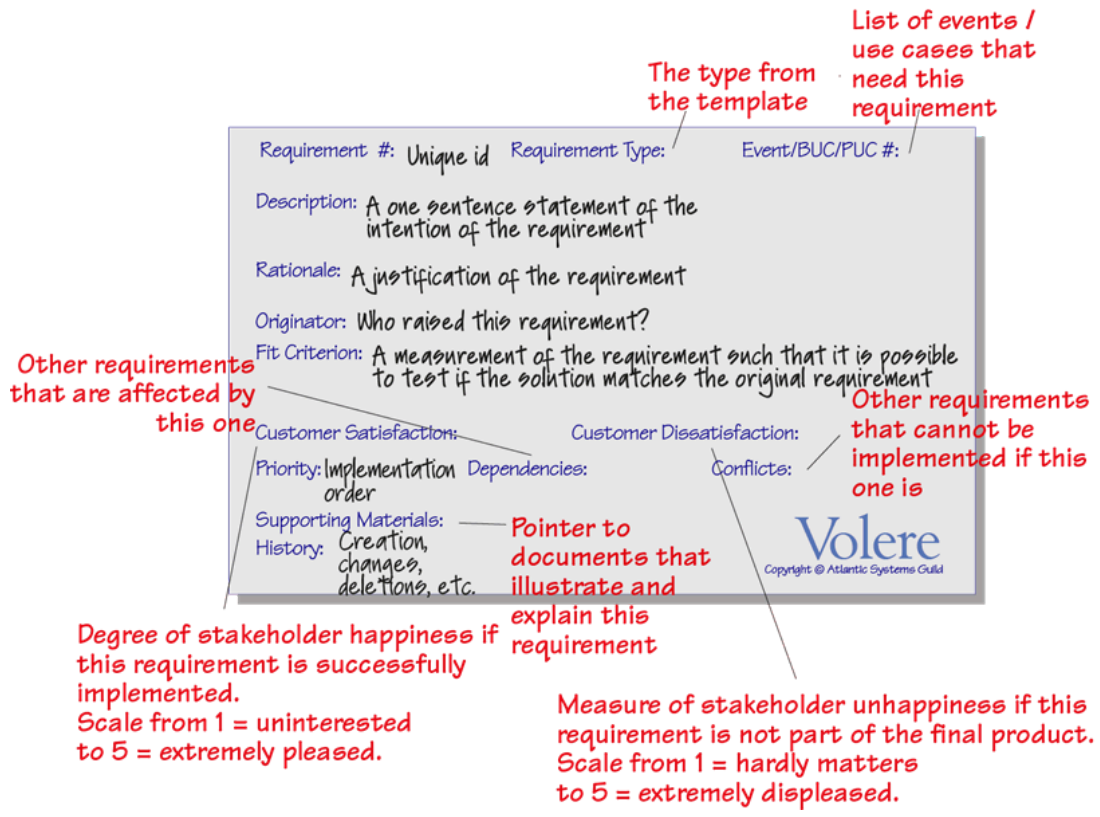
Interviews are a common and effective means of capturing requirements, helping to understand the needs and perspectives of stakeholders and to gather information quickly.

面试的不同类型及其特点： Different types of interviews and their characteristics:

三种类型的面试：非结构化、结构化和半结构化（前两种的混合） Three types of interviews: unstructured, structured and semi-structured (a mixture of the first two)

- 非结构化的面试： Unstructured interviews:
  - 对讨论方向的控制有限。 Limited control over the direction of the discussion.
  - 当对领域知识的理解有限时，最适合探索。 Exploration is best suited when there is limited understanding of domain knowledge.
  - 存在风险：某些方面过于详细，而其他方面则不够详细。 There is a risk that some aspects are too detailed while others are not.
- 结构化面试：Structured interviews:
  - 使用预设的一组问题来收集特定的信息。 Use a pre-defined set of questions to gather specific information.
  - 成功的关键在于知道何时问什么问题，向谁提问。 The key to success is knowing when to ask what questions and to whom.

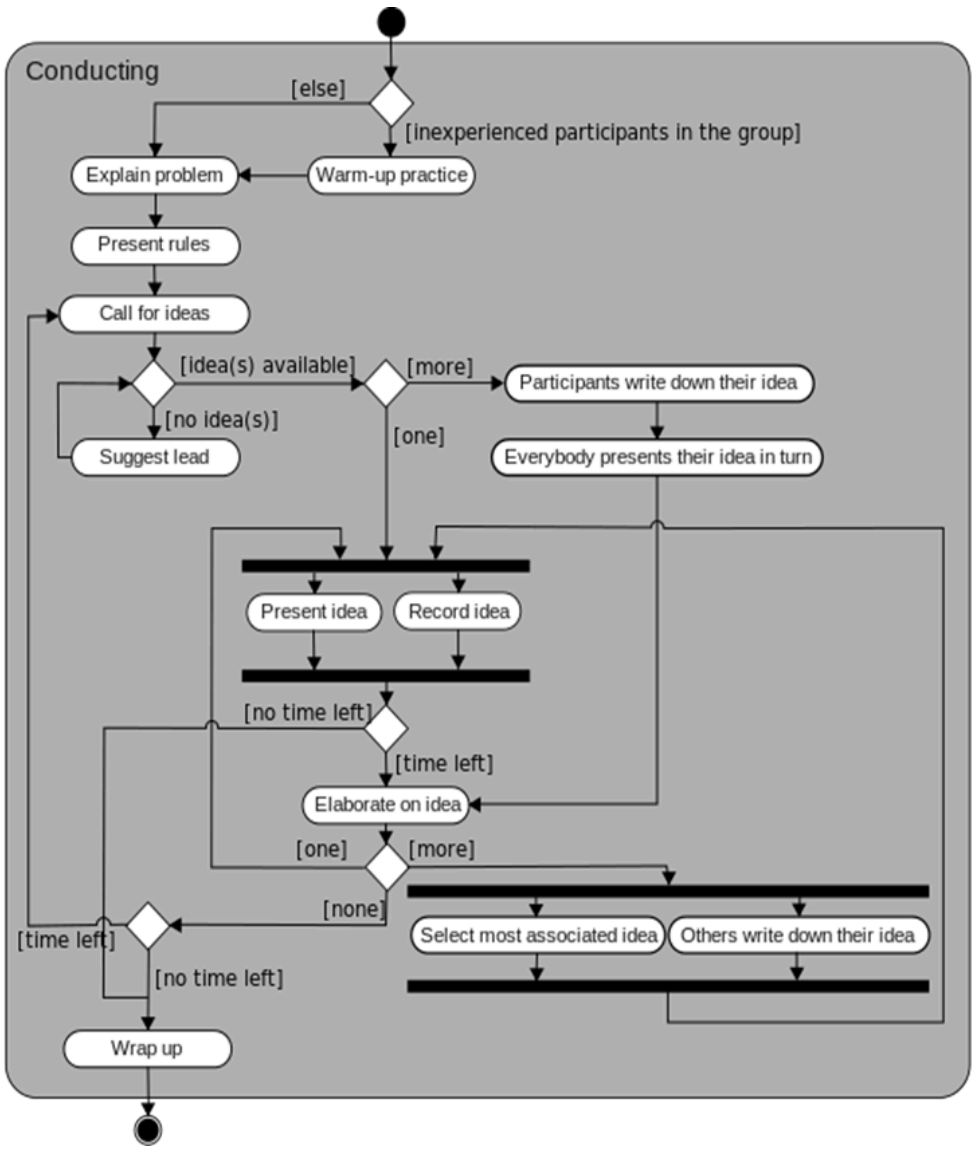
- 可以使用模板来指导结构化面试，例如 Volere 卡片。 Templates can be used to guide structured interviews, such as Volere cards.
- 问题可以先开放后封闭（更具体）。 Questions can be open and then closed (more specific).



头脑风暴会议(Brainstorming session)

头脑风暴是一项以小组为基础的活动，通过反复收集针对特定主题的想法清单

Brainstorming is a group-based activity in which a list of ideas for a specific topic is gathered through repeated



任务分析(Task Analysis)

任务分析研究人们如何完成工作： Task analysis examines how people get work done:

- 他们做的事情、操作的对象以及需要的知识。 What they do, who they operate with, and the knowledge they need.
- 采用自顶向下方法，将高级任务分解为子任务并最终描述详细的序列。 A top-down approach is used to decompose high-level tasks into subtasks and eventually describe detailed sequences.
- 确定执行任务所需的知识。 Identify the knowledge required to perform the task.

- 示例：SGS 工作人员如何处理学生注册任务，包括各种典型和非典型的情况。 EXAMPLE: How SGS staff handle student registration tasks, including a variety of typical and atypical situations.

场景分析(Scenario-Based Analysis)

- 提供了设计和开发交互式系统时更加用户导向的观点。 Provides a more user-orientated view of designing and developing interactive systems.
- 举例说明观察 SGS 工作人员如何处理非本地学生 A 和本地学生 B 的注册，并询问未涵盖的情况。 Give an example of observing how SGS staff handled the enrolment of non-local student A and local student B and ask about non-covered situations.

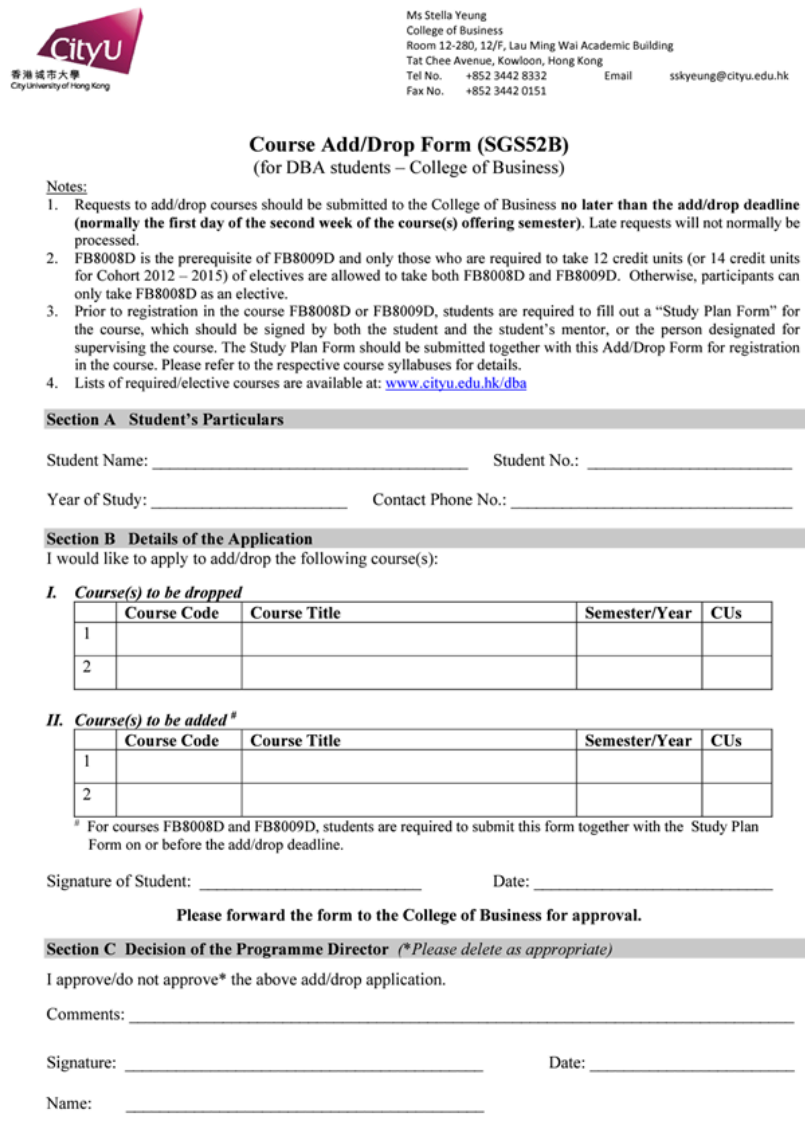
场景分析的一个例子：观察图书馆的书籍归还过程 An example of scenario analysis: observing the process of returning books in a library

- 检查到期日期。 Check the expiry date.
- 如果逾期，则收取罚款。 If it is overdue, a fine is charged.
- 记录书籍再次可用。 Record books are available again.
- 将书籍放回原处。 Put the books back where they belong.
- 接着与图书管理员讨论： Discussion with the librarian ensues:
  - 如果归还书籍的人没有登记为客户？ If the person returning the book is not registered as a customer?
  - 如果书籍损坏？ If the books are damaged?
  - 如何处理客户有其他逾期书籍或预订？ How do you handle customers with other overdue books or bookings?

表格分析(Form analysis)

- 列出确定或具有多样性的项目，以及填写表格时（过去、现在或未来）信息可获得的时间点

  List items that are identified or have diversity and the point in time when the information was available (past, present or future) at the time of completing the form



Ms Stella Yeung
College of Business
Room 12-280, 12/F, Lau Ming Wai Academic Building
Tat Chee Avenue, Kowloon, Hong Kong
Tel No. +852 3442 8332 Email sskyeung@cityu.edu.hk
Fax No. +852 3442 0151

Course Add/Drop Form (SGS52B)
(for DBA students – College of Business)

Notes:
1. Requests to add/drop courses should be submitted to the College of Business **no later than the add/drop deadline (normally the first day of the second week of the course(s) offering semester)**. Late requests will not normally be processed.
2. FB8008D is the prerequisite of FB8009D and only those who are required to take 12 credit units (or 14 credit units for Cohort 2012 – 2015) of electives are allowed to take both FB8008D and FB8009D. Otherwise, participants can only take FB8008D as an elective.
3. Prior to registration in the course FB8008D or FB8009D, students are required to fill out a "Study Plan Form" for the course, which should be signed by both the student and the student's mentor, or the person designated for supervising the course. The Study Plan Form should be submitted together with this Add/Drop Form for registration in the course. Please refer to the respective course syllabuses for details.
4. Lists of required/elective courses are available at: www.cityu.edu.hk/dba

Section A Student's Particulars

Student Name: _____ Student No.: _____

Year of Study: _____ Contact Phone No.: _____

Section B Details of the Application

I would like to apply to add/drop the following course(s):

I. Course(s) to be dropped

	Course Code	Course Title	Semester/Year	CUs
1				
2				

II. Course(s) to be added #

	Course Code	Course Title	Semester/Year	CUs
1				
2				

For courses FB8008D and FB8009D, students are required to submit this form together with the Study Plan Form on or before the add/drop deadline.

Signature of Student: _____ Date: _____

Please forward the form to the College of Business for approval.

Section C Decision of the Programme Director *(*Please delete as appropriate)*

I approve/do not approve* the above add/drop application.

Comments: _____

Signature: _____ Date: _____

Name: _____

焦点小组和引导式研讨会(Focus Group and Facilitated Workshop)

- 焦点小组是一个规模较小但人口构成多样的群体，通过引导或公开讨论研究他们的反应。

  Focus groups are small but demographically diverse groups of people whose responses are studied through guided or open discussion.

- 小组成员会被问及他们的看法、意见、信念和态度。

  The panellists will be asked about their views, opinions, beliefs and attitudes.

- 软件需求分析师记录他或她从小组中获得的要点。

  The Software Requirements Analyst records the key points he or she takes away from the group.

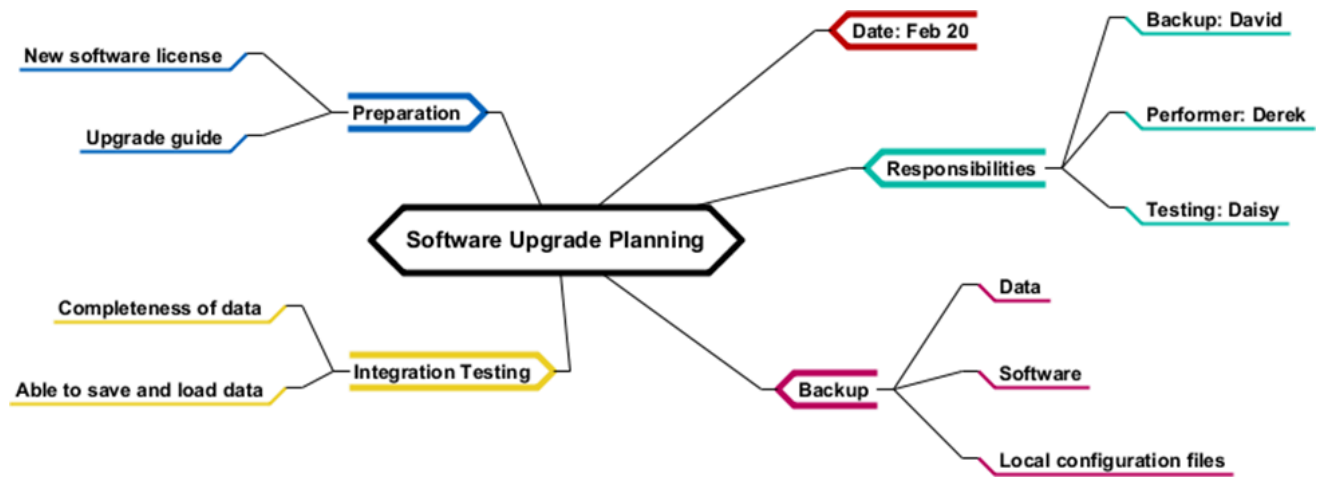
- 促进式研讨会由跨职能团队成员参与，从各个角度研究主题，并根据研讨会的结果做出决定。

  Facilitated workshops involve cross-functional team members who examine the topic from various perspectives and make decisions based on the results of the workshop.

思维导图(Mind map)

一种图形化的组织工具，用于展示概念之间的关系

A graphical organisational tool to show relationships between concepts



集体故事(Group Storytelling)

共同构建一个故事、分享一个故事、完成一个未完成的故事、放大/缩小故事、角色扮演故事或分析故事

Co-constructing a story, sharing a story, completing an unfinished story, zooming in/out of a story, role-playing a story or analysing a story

用户故事(User story)

以“作为一个…，我想要…以便…”的形式描述用户的需求

With the phrase "As a… , I want… to…" Describe the user's needs in the form of

相关建议：

Related recommendations:

- 访谈：特别适用于在新领域的新项目中收集初始背景信息。

  Interviews: Particularly useful for gathering initial background information in new projects in new areas.

- 合作会议（焦点小组、研讨会、头脑风暴）：有效。

  Collaborative sessions (focus groups, workshops, brainstorming): effective.

- 从现有系统收集数据：必须做，但不过度分析。

  Collect data from existing systems: must be done, but not over-analysed.

- 敏捷（思维导图、集体讲故事、用户故事）：如今很流行。

  Agile (mind mapping, collective storytelling, user stories): very popular today.

- 问卷调查：无效。

  Questionnaire: not valid.

Specifying requirements

- 在收集需求时，应该对其进行结构化处理。

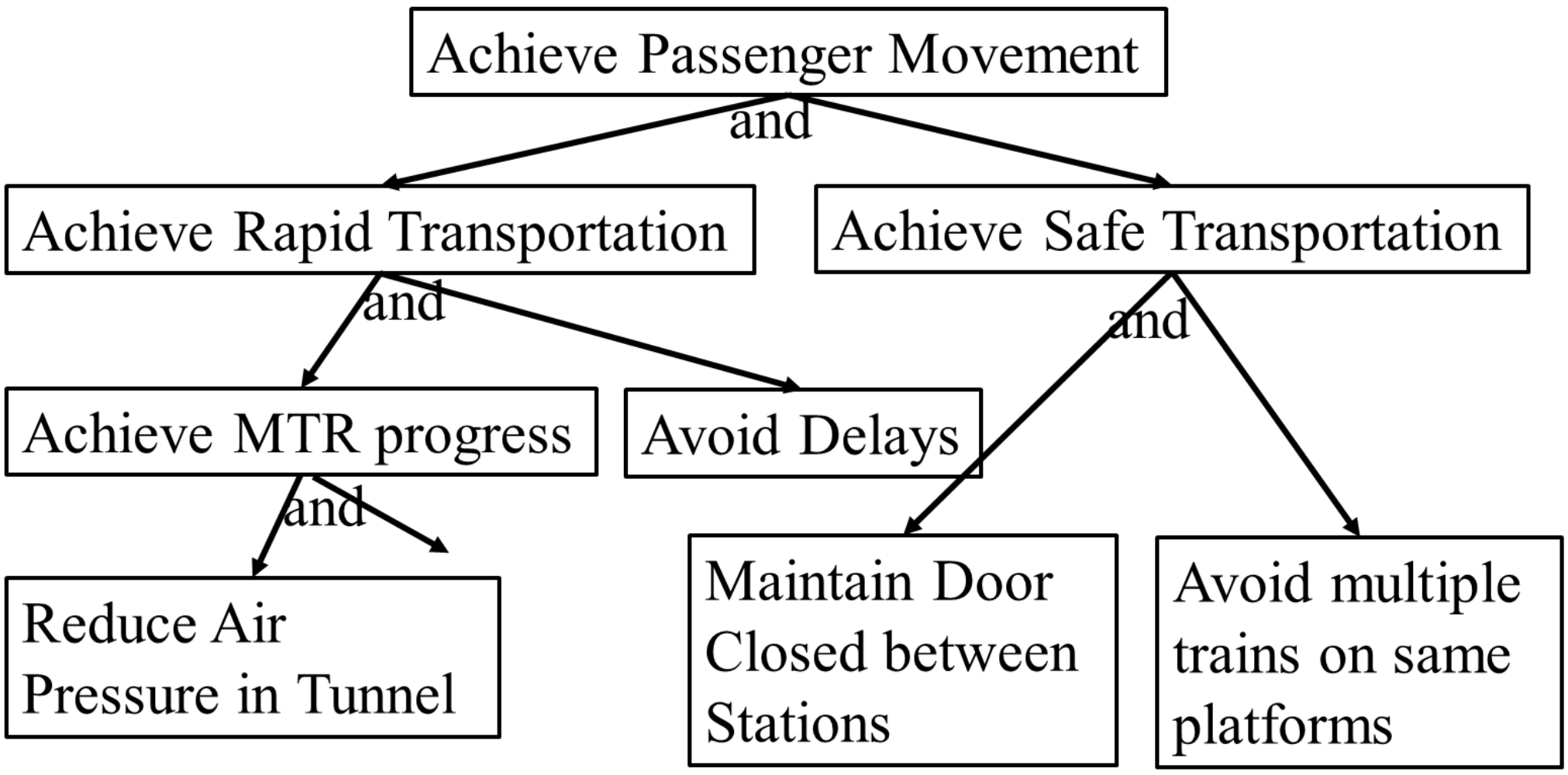
  When collecting requirements, they should be structured.
- 结构化需求的方法包括：

  Methods of structuring requirements include:
  - 层次结构：将高层级需求分解为低层级需求。

    Hierarchy: decomposition of high-level needs into low-level needs.
  - 将需求链接到特定的利益相关者（例如，管理和最终用户都有自己的需求集）。

    Linking requirements to specific stakeholders (e.g., management and end users each have their own set of requirements).
- 在结构化需求的同时，也要分析它们及其相互关系。

  While structuring the needs, it is also important to analyse them and their interrelationships.



Validating Requirement

来自多个来源的相同需求有助于我们判断其有效性。

Identical requirements from multiple sources help us judge their validity.

直接链接是指需求直接来自利益相关者，而间接链接则是通过中间人或代理用户获取的需求

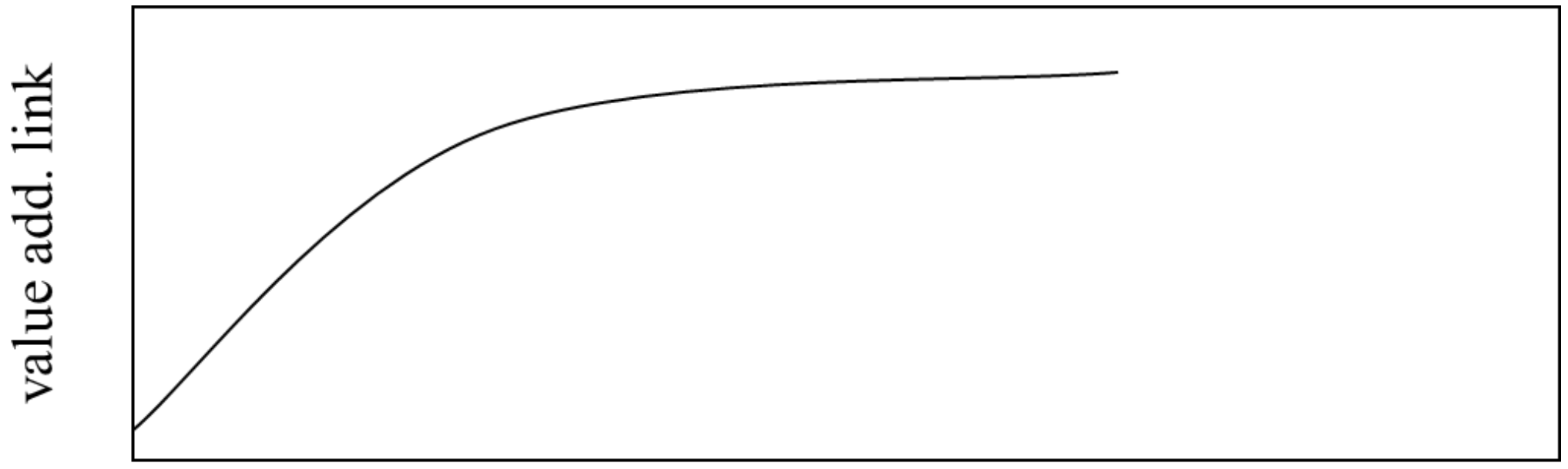
Direct links are where the demand comes directly from the stakeholder, whereas indirect links are where the demand is acquired through an intermediary or proxy user

- 第一课：不要过分依赖间接链接（中间人、代理用户）（中间人或代理用户的理解和传达可能会导致需求失真或者遗漏重要细节。

  Lesson 1: Don't rely too much on indirect links (intermediaries, proxy users) (Intermediaries or proxy users' understanding and communication may result in distorted requirements or omission of important details.

- 第二课：更多的链接更好（但有一定的限度）。（更多的链接通常会提高需求验证的价值，但是也有一个临界点，在这个点之后增加更多的链接可能不会带来显著的好处，甚至可能导致混乱或冗余

  Lesson 2: More links are better (but with limits). (More links usually increase the value of demand validation, but there is a tipping point after which adding more links may not provide significant benefits and may even lead to confusion or redundancy)



需求规格说明书的检查应考虑正确性、完整性和一致性、准确性和可读性以及可测试性。

Requirements specifications should be checked for correctness, completeness and consistency, accuracy and readability, and testability.

- 不同开发阶段的一些辅助工具： Some aids for different development stages:
  - 早期与客户进行结构化的走查； Early structured walk-throughs with clients;
  - 初版原型； Prototype
  - 敏捷开发中的测试计划或单元测试； Test planning or unit testing in agile development;
  - 交付时的用户验收测试。 User acceptance testing at delivery.

Negotiating requirements

除了在需求提取过程中协商需求外，还应在优先级排序时与利益相关者协商需求。

In addition to consulting on requirements during the requirements extraction process, requirements should also be consulted with stakeholders during prioritisation.

例如，可以给需求项目打上“高”、“中”或“低”的标签。

For example, demand items can be labelled as "high", "medium" or "low".

可以使用MoSCoW方法分类需求：

Requirements can be categorised using the MoSCoW methodology:

- 必须具备的要求：强制性的需求； Requirements that must be in place: mandatory requirements;
- 应该具备的要求：重要的但不是强制性的； Requirements that should be in place: important but not mandatory;
- 可能具备的要求：如果时间允许的话； Possible requirements: if time permits;
- 不具备的要求：今天不具备（可能是明天）。 Requirements not available: not available today (could be tomorrow).

Summary

- 需求工程包括需求提取、规范、验证和谈判。 Requirements engineering includes requirements elicitation, specification, validation and negotiation.

- 提取有五个迭代步骤，其中四个是准备步骤，并且使用多种提取技术。

  The elicitation has five iterative steps, four of which are preparatory, and uses a variety of elicitation techniques.
- 我们不仅收集需求，还要塑造它们：

  We don't just collect requirements, we shape them:
  - 通过结构化分析需求；

    Analysing requirements through structured;
  - 在需求提取和优先级排序时与利益相关者进行谈判；

    Negotiate with stakeholders in requirements extraction and prioritisation;
  - 从多个来源验证需求。

    Validate requirements from multiple sources.

4.Design Principles and Framework

In Short

我们需要注重程序结构

We need to focus on programme structure

David Parnas 强调了分解程序的不同方式对软件维护的影响

David Parnas highlights the impact of different ways of decomposing the process on software maintenance

不同的程序分解方法会产生非常不同的软件维护后果

Different approaches to programme decomposition can have very different software maintenance consequences

程序结构的风格：

Style of programme structure:

- 在更高层次上，我们指的是软件架构（或系统架构）、企业模式、架构模式等及软件应用如何组织的大局观。

  At a higher level, we refer to the big picture of software architecture (or system architecture), enterprise models, architectural patterns, etc. and how software applications are organised.
- 在较低层次上，我们指的是编程习惯、编程原则和设计模式

  At a lower level, we refer to programming habits, programming principles and design patterns

软件架构的基础是模块化，即将一个模块分解成一组连接的子模块。例如在计算机网络中的 OSI 模型中的分层样式。

Software architecture is based on modularity, i.e. the decomposition of a module into a set of connected sub-modules. An example is the layering style in the OSI model in computer networks.

我们的目的是利用这些架构样式来满足非功能性需求例如快速启动等。

Our aim is to use these architectural styles to fulfil non-functional requirements such as fast startup.

我们在构建软件架构时通常采用以下三个步骤：

We typically use the following three steps when building software architectures:

1. 将一个模块递归分解为一组相互作用的子模块。为子模块分配功能责任，以满足 “挑选 ”的功能要求。例如，可以将不同模型的子模块和模型完整性作为一个整体考虑。

   Recursively decompose a module into a set of interacting submodules. Assign functional responsibilities to sub-modules to meet "pick and choose" functional requirements. For example, sub-modules of different models and model integrity can be considered as a whole.

2. 为子模块分配非功能质量属性。例如，视图的可用性、子模型的可扩展性、控制器的互操作性。

   Assign non-functional quality attributes to submodules. For example, view availability, submodel extensibility, controller interoperability.

3. 对子模块施加非功能性约束。例如，控制器在不同模型之间的互操作不应影响视图模块。

   Impose non-functional constraints on submodules. For example, controller interoperability between different models should not affect the view module.

此外，观察也是不可缺少的步骤

In addition, observation is an indispensable step

在架构设计过程中，我们不会预先确定或决定以下内容：

During the architectural design process, we do not pre-determine or decide on the following:

- 功能需求的具体实现方式。例如，不需要关心软件初始化算法或者维护模型完整性的方法。

  The specific implementation of the functional requirements. For example, there is no need to be concerned with software initialisation algorithms or methods for maintaining model integrity.

- 非功能性需求的具体实现方式。我们只分配质量和非功能性约束，并附带解释/文档说明为什么它们对于实现非功能性需求是必要的。

  Specific realisation of non-functional requirements. We only assign quality and non-functional constraints with explanations/documentation of why they are necessary for the realisation of non-functional requirements.

软件架构定义： Software Architecture Definition:

架构是一个系统在其环境中的基本概念或性质，体现在其元素、关系以及设计和演进原则中

Architecture is the fundamental concept or nature of a system in its environment, embodied in its elements, relationships, and design and evolutionary principles

- 架构是概念性的。例如，我们为不同的模块赋予意义，使它们的交互能讲述如何处理功能需求及其相关的质量问题。

  The architecture is conceptual. For example, we give meaning to the different modules so that their interactions tell the story of how functional requirements and their associated quality issues are handled.

- 架构关注的是重要的事情。

  Architecture focuses on what matters.

- 架构存在于一定的上下文中。例如，针对“软件初始化”的架构可能并不适用于基于服务器的批处理应用。任何非平凡的软件系统都有架构，而上下文决定了架构是否符合系统的需要

  Architectures exist in certain contexts. For example, an architecture for "software initialisation" may not be applicable to a server-based batch application. Any non-trivial software system has an architecture, and the context determines whether the architecture fits the needs of the system.

在设计一个具有模糊设计要求的排序数据系统的软件架构时，大致流程如下：

In designing a software architecture for a sorted data system with fuzzy design requirements, the general process is as follows:

1. 罗列系统的基本需求。例如包括快速响应用户操作、高效运行以及持久化且高效地保存排序结果

   List the basic needs of the system. These include, for example, fast response to user actions, efficient operation, and persistent and efficient storage of sorting results.

2. 根据需求在不同子模块中指定决策和设计架构。如分离用户界面以提高响应速度、根据数据特性选择高效的排序算法等

   Specify decision-making and design architecture in different sub-modules as required. For example, separating the user interface to improve responsiveness, selecting efficient sorting algorithms based on data characteristics, etc.

3. 在设计过程中会不断发现新的设计要求，比如安全性和可用性，提出相应的解决方案

   New design requirements, such as security and usability, are continually identified during the design process, and solutions are proposed accordingly.

4. 最后系统可能会需要满足其他设计需求，最终进行改进

   Eventually the system may need to meet other design requirements and eventually make improvements

Role of Software Architecture

架构师需要负责所有技术层面的高阶设计决策、理解商业影响、接受和预见变化等。

Architects need to be responsible for high-level design decisions at all technical levels, understanding business impacts, embracing and anticipating change, and more.

架构为什么重要？ Why is architecture important?

架构是利益相关者沟通的载体 Architecture as a vehicle for stakeholder communication

架构体现了最早的一组设计决策 The architecture reflects the earliest set of design decisions

- 对实施的约束 Constraints on implementation
- 决定组织结构 Decide on the organisational structure
- 抑制或促进质量属性 Inhibit or promote quality attributes

Driver of Software Architecture

架构驱动是一个设计要求，会影响软件架构师早期的设计决策

Architecture-driver is a design requirement that influences the software architect's early design decisions

注： notes:

- 功能特性（即功能）是软件需求，不足以作为架构驱动器。

  Functional features (i.e., functionality) are software requirements and are not sufficient as architectural drivers.

- 架构处理非功能性需求（例如，具有不同质量的解决方案以实现相同的功能）

  Architecture to handle non-functional requirements (e.g., solutions with different qualities to achieve the same functionality)

在支付系统中： in the payment system:

传统支付系统： Traditional payment systems:

- 特征：账户间资金转移、结算

  Characteristics: transfer of funds between accounts, settlement

- 好品质：T+2结算、单一平台、可靠、易用性（商家和客户）

Good qualities: T+2 settlement, single platform, reliability, ease of use (merchants and customers)

更快支付系统（FPS）： Faster Payment System (FPS):

- 特征：账户间资金转移、结算、二维码

  Features: inter-account fund transfer, settlement, QR code

- 好品质：实时且持续的资金转移和结算、跨平台、可靠、易用性（商家和客户）、易于添加新功能、单一通用二维码标准

  Good qualities: real-time and continuous funds transfer and settlement, cross-platform, reliability, ease of use (for both merchants and customers), easy to add new features, single universal QR code standard

更快支付系统相对于传统支付系统在质量方面的观察结果：

Observations on the quality of faster payment systems relative to traditional payment systems:

- 保持：可靠、易用（商家和客户）

  Maintain: reliable and easy to use (merchants and customers)

- 改进：T+2结算变为实时和持续

  Improvement: T+2 settlement becomes real-time and continuous

- 普及化：单一平台变为跨平台

  Universalisation: single platform to cross-platform

- 扩展：易于添加新功能、统一的二维码标准

  Extension: easy to add new features, unified QR code standard

支付系统的设计实际上是对软件架构原则的应用案例，它展示了如何通过精心规划的架构来解决实际业务问题并满足用户需求。无论是提高处理速度还是增强安全性，都是通过对系统架构进行深思熟虑的设计而实现的。

The design of a payment system is actually an application case of the principles of software architecture, showing how a carefully planned architecture can be used to solve real business problems and satisfy user needs. Whether it's increasing processing speed or enhancing security, it's all achieved through a thoughtful design of the system architecture.

Quality Attributes and Patterns/Tactics

质量这一概念在软件架构中起着核心作用。

The concept of **quality** plays a central role in software architecture.

- 有些质量可以通过执行观察到，比如性能、安全、可用性和功能。

  Some qualities can be observed through execution, such as performance, security, usability, and functionality.

- 其他一些质量则不能通过执行观察到，如可修改性、可移植性、可重用性、可集成性和可测试性。

  Some other qualities are not observable through execution, such as modifiability, portability, reusability, integrability and testability.

- 质量也影响软件项目的各个方面。

  Quality also affects all aspects of a software project.

质量属性指的是系统的一种可测量或可检验的属性，用于表明系统满足利益相关者需求的程度

A quality attribute is a measurable or testable property of a system that indicates the degree to which the system meets the needs of its stakeholders.

在开发中，可测量性或可检验性非常重要

Measurability or testability is very important in development

如果不知道当前系统离目标有多远，就无法设计出符合目标的系统。

Without knowing how far the current system is from the goal, it is impossible to design a system that meets the goal.

例如： Such as:

- 不可测量的用户故事：“作为顾客，我可以迅速看到所有加密货币的价格”；

  Unmeasurable user story: 'As a customer, I can quickly see the prices of all cryptocurrencies';
- 可测量的用户故事：“作为顾客，我可以在一秒钟内看到我选择的20种加密货币的最新和历史价格

  Measurable user story: "As a customer, I can see the latest and historical prices of 20 cryptocurrencies of my choice in a second!

作为分析师，应该追求可测量的目标

As an analyst, you should pursue measurable goals

许多以“-ity”结尾的词组成了权衡关系

Many words ending in "-ity" form trade-offs.

- 提升隐私会降低可用性。 Enhancing Privacy Reduces Usability
- 更高的性能意味着更低的互操作性。 Higher performance means lower interoperability.
- 更高的模块化会导致更长的上市时间。 Higher modularity leads to longer time to market.
- 更高的可靠性能导致更低的性能。 Higher reliability can lead to lower performance.
- 更快的上市时间和更低的成本会导致更低的稳定性。 Faster time to market and lower costs lead to less stability.

我们应该意识到，在分解系统以应对一个质量属性时，应评估其对其他质量属性的影响

We should be aware that when disaggregating the system to respond to one quality attribute, its impact on other quality attributes should be assessed

系统分解是软件工程和系统设计中的一个重要概念，它指的是将一个复杂系统划分为更小、更易于管理的子系统或组件的过程。这种做法有助于降低系统的复杂性，使每个部分都可以独立地进行设计、实现、测试和维护。通过系统分解，可以提高开发效率，简化问题解决过程，并且更容易理解和控制整个系统的行为

System decomposition is an important concept in software engineering and system design that refers to the process of dividing a complex system into smaller, more manageable subsystems or components. This practice helps to reduce the complexity of a system so that each part can be designed, implemented, tested and maintained independently. Through system decomposition, development can be made more efficient, the problem-solving process can be simplified, and the behaviour of the entire system can be more easily understood and controlled

接下来逐一介绍设计品质

The next step is to introduce the design qualities one by one

- 可行性：指项目的成功概率和资源负担能力。根据上市时间和负担能力等考虑项目的成功率

  Feasibility: refers to the probability of success and resource affordability of the project. Consider the success rate of the project based on time to market, affordability, etc.
- 可复用性：设计是否能够方便地重复使用。考虑复用策略和复用的先决条件

  Reusability: whether the design can be easily reused. Consider reuse strategies and reuse prerequisites
- 一致性：设计元素之间的协调性和统一性。设计在概念上是否清晰且一致，遵循架构风格约束并记录主要的架构决策。

  Coherence: the coordination and unity between design elements. Is the design conceptually clear and consistent, following architectural style constraints and documenting key architectural decisions.
- 简洁性：设计是否简洁明了，易于理解。复杂的问题不一定需要复杂的解决方案，尽可能简洁。

  Simplicity: Whether the design is clear and concise and easy to understand. Complex problems do not necessarily require complex solutions, and are as simple as possible.
- 稳定性：设计是否稳定，不易出错。避免模块之间过多的依赖，最好将环境问题封装到底层平台或框架中。

  Stability: whether the design is stable and not prone to errors. Avoid too many dependencies between modules, and it is best to encapsulate environmental issues into the underlying platform or framework.
- 组合性：设计是否容易与其他组件组合在一起。由辅助组件组成更高级组件是否容易

  Combinability: how easily the design can be combined with other components. Is it easy to compose higher-level components from auxiliary components
- 可部署性：设计是否便于部署到实际环境中。手动或自动，基于云或服务组合?

  Deployability: Whether the design is easy to deploy into a real-world environment. Manual or automated, cloud-based or a combination of services?

在设计品质之外，还有运营品质、故障品质、安全品质、需求或特性转变品质

In addition to design quality, there are operation quality, failure quality, security quality, and demand or Change in requirements or features

运营品质： operation quality

- 性能：延迟（延迟，首次响应，完成时间）和吞吐量（单位时间输出，带宽）

  Performance: latency (delay, first response, completion time) and throughput (output per unit time, bandwidth)
- 可扩展性：随着吞吐量（请求数）、用户数（并发性）、数据量（I/O 和处理）、网络规模（节点数）和系统大小（组件数）的增长情况：动态扩展可能吗?

  Scalability: what happens as throughput (number of requests), number of users (concurrency), amount of data (I/O and processing), network size (number of nodes), and system size (number of components) grow: is dynamic scaling possible?
- 容量：限制、过载和利用率

  Capacity: limit, overload and utilisation rate

- 易用性：易于学习和记忆吗？遵循规范？用户满意度高吗？易于访问吗？

  *E*asy to learn and recall efficiently? Follow the norm? user satisfaction? Easy to access?
- 可维护性：停止操作与连续操作？重启还是不重启？手动错误报告与自动化？

  operation stop vs. continuous operation? Reboot or no reboot? Manual error report vs. automated?
- 可见性：系统行为可审计并记录吗？

  system behavior auditable and logged

故障品质：

- 可靠性：平均故障间隔时间（MTBF），平均故障时间（MTTF）

  Reliability: Mean time between failures (MTBF), Mean time to failure (MTTF)
- 恢复能力：平均恢复时间，平均修复时间，平均响应时间

  Recoverability: Mean time to recovery, mean time to repair, or mean time to respond
- （功能）可用性：根据可靠性和恢复能力计算得出

  (Functional) Availability: Can be calculated based on reliability and recoverability
  - 例如，可用性 = MTTF / (MTTF + MTTR)

    E.g., Availability = MTTF / (MTTF + MTTR)

安全品质：

Security-related Quality

- 认证：确认用户身份

  Authentication: to confirm user identity
- 授权：限制访问

  Authorization: to restrict access
- 机密性：避免未经授权的数据访问

  Confidentiality: to avoid unauthorized data access
- 完整性：保护系统/数据免受篡改

  Integrity: to protect the system/data from tampering
- 可防御性：防止攻击

  Defensibility: to protect against attacks
- 隐私：保护敏感数据

  Privacy: to secure confidential data

需求或特性转变品质：

Change-related Quality

- 可配置性（推迟架构设计更好）：

  Configurability (deferring the architectural design is better)
  - 如何初始化、重新激活、放置和竞标组件?

    How to initialize, (re-)activate, place and bid components?
  - 如何添加或删除系统资源（如节点或重复的组件副本）？

    How to add or remove system resources (e.g., nodes or duplicated copies of the same components)
  - 特性如何切换开或关?

How can a feature be toggled on or off?

    - 示例：实时参数 > 启动时参数 > 硬编码

      E.g., live parameters > parameters at starting > hardcoded

- 可定制性：

  Customizability

    - 适用于所有尺寸的产品线、UI皮肤、JSON以解耦数据

      One-size-fits-all, product line, UI skin, JSON for data to decouple from hardcoding the data

- 持续时间：

  Duration

    - 是否可以临时然后撤销？是否可以永久进化？

      Temporary and then revoked possible? Permanent evolution possible?

- 功能演进：

  Feature Evolution

    - 新功能的可扩展性；现有功能的可修改性

      extensibility for new feature for,; modifiability for existing feature

- 兼容性：

  Compatibility

    - 标准化接口、格式、平台。版本向后/向前兼容

      Standardized interface, format, platform. Version backward/forward compatibility

- 可移植性：

  Portability

    - 平台独立与本机代码？一次编写，到处运行？基于虚拟机？

      Platform independent versus native code? Write once, run anywhere? VM-based?

- 互操作性（与其他系统）：

  Interoperability (with other systems)

    - 标准、中介器的存在、内容协商

      Standard, presence of medicator, content negotiation

- 集成：

  Integration

    - 标准接口和数据格式、持续、实时、同时集成的组件数量

      Standard interface and data format, continuous, real-time, # of integrated components at the same time

项目健康品质：

Project Health Quality

- 数据持久性、检查点和恢复

  Data persistence, checkpoint, and recovery

- 备份和灾难恢复计划

  Backup and disaster recovery plan

- 可维护性

  Maintainability

在进行API设置时，应遵循以上原则

When setting up the API, you should follow these principles

- 易于理解：可用性、简单、小、文档良好、有意义的错误消息

  Easy to understand: Usability, Simplicity, Small, Well-documented, Meaningful error messages
- 高服务质量：可扩展性、可靠、可用、安全、兼容性、标准化输入输出消息类型

  High quality of services: Scalability, Reliable, Available, Security, Compatibility, Standardized I/O message type,
- 命名一致性（从端点、函数、I/O、元素顺序、与其他API的相似性）

  Naming Consistency (from endpoints, functions, I/O, Ordering of elements, similarity with other APIs)
- 从客户端应用的角度进行设计（易用性）

  Design from the client application’s perspective (usability)

Attribute-Driven Design

属性驱动设计（ADD）是一种针对质量属性增量处理的架构设计方法论

ADD is an architectural design methodology to tackle quality attributes incrementally

- 步骤：

  steps:

  1. 选择一个系统模块/连接器分解

     Choose a system module/connector to decompose
  2. 改进这个模块：

     Refine this module:
  - 选择架构驱动力（质量是驱动力）

    choose architectural drivers (quality is the driving force)
  - 选择满足驱动力的模式/策略/风格

    choose patterns/tactics/styles that satisfy the drivers
  - 应用模式/策略/风格并分配责任

    apply patterns/tactics/styles and assign responsibilities
  3. 重复步骤1-2

     Repeat steps 1-2

ADD迭代一个例子：

Example ADD iterations

- 顶层（整个系统）：

  Top-level (whole system):
  - 地址可用性---->战术：分离用户界面----->模式：三层架构

    address usability --->tactic: separate user interface ----> pattern: three tier architecture
- 在表示层模块内的较低层级：

  Lower-level, within the presentation tier module:
  - 地址安全性----->战术：验证用户身份

address security ----->tactic: authenticate users

- 在数据层模块内的较低层级：

  Lower-level, within the data tier module:

  - 地址可用性---->战术：主动冗余

    address availability -----> tactic: active redundancy

架构战术：

Architectural Tactics

- 架构战术是应对质量属性问题的一种迭代应用的方法。

  ADD iteratively applies one or more architectural tactics to address a quality attribute issue.

- 架构战术是概念，通常在解决方案中隐式使用。

  Architectural tactics are concepts. They are usually used implicitly in a solution

- 当今最常提到的战术是安全战术（防御方法）。

  The most often mentioned tactics nowadays are Security tactics (defense approaches).

Taking Decisions

设计决策的作用：

Uses of design decisions

- 识别利益相关者的关键设计决策

  Identify key design decisions for a stakeholder

  - 快速提供关键决策，例如向新成员介绍并使其了解最新情况

    Make the key decisions quickly available. E.g., introducing new people and making them up-to-date.

  - 获取理由，评估/验证决策是否符合需求

    Get a rationale, evaluate/validate decisions against requirements

- 评估影响

  Evaluate impacts

  - 如果要更改某个元素，哪些元素会受到影响（决策、设计、问题等）

    If we want to change an element, what elements are impacted (decisions, design, issues)?

  - 清理架构，识别重要的架构驱动因素

    Clean up the architecture, identify important architectural drivers

- 明确设计决策

  Make the design decision explicit

  - 例如，带有理由的架构描述

    E.g., architectural description with rationale

  - 例如，使用4+1架构视图来表示架构

    E.g., Use the 4+1 architectural view to represent the architecture

在做出设计决定的同时，我们需要将设计概念体现为明确的产品（即演示），以便与利益相关者（如团队成员、客户或用户）沟通，解决他们对非功能性需求的担忧。

While we make our design decision, we need to manifest the design concept as **explicit artifacts (i.e., representation)** to communicate with stakeholders

针对不同人群和不同目的需要不同的演示形式，例如PPT和UML等

Different presentation formats, such as PPT and UML, are required for different people and purposes.

Architectural styles

架构风格是对组件和连接器类型的描述以及它们运行时控制和/或数据传输模式的一种描述。

An architectural style is a description of **component and connector types** and a pattern of their runtime control and/or data transfer.

- 没有正式的概念来定义什么是组件和连接器

  No formal concept for what is a component and what is a connector
- 可通过实例学习

  Learn by examples

黑板架构:

Blackboard

- 黑板 (BB)：包含问题和部分解决方案的数据。

  Blackboard (BB): contains data on the problem and partial solutions.
- 知识源 (KS)：专门化的模块，每个提供对整个问题的一部分的解决方案。

  Knowledge source (KS):Specialized modules, each providing a solution to some part of the entire problem.
  - Condition()检查模块是否能为BB作出贡献

    Condition() checks whether the module can contribute to the BB
  - Acton()更新BB

    Acton() updates BB.
- 控制

  Control
  - 定期监控BB上的变化

    Periodically monitor changes on BB
  - 发现变化后决定选择并执行哪个KS

    On identifying changes, decide KS to be selected and executed.

客户端服务器及其变体的架构

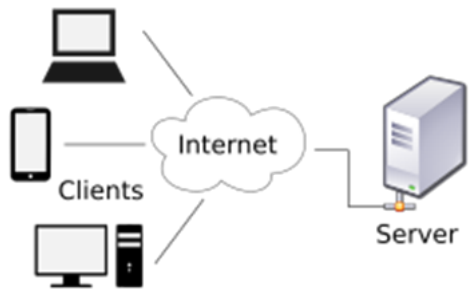
Client-Server and variant

客户端服务器:

Client-server

客户端-服务器架构（Client-Server Architecture）是一种分布式应用架构，其中任务或工作负载被分为服务请求者（客户端）和服务提供者（服务器）

Client-Server Architecture (CSA) is a distributed application architecture in which tasks or workloads are divided into service requesters (clients) and service providers (servers)

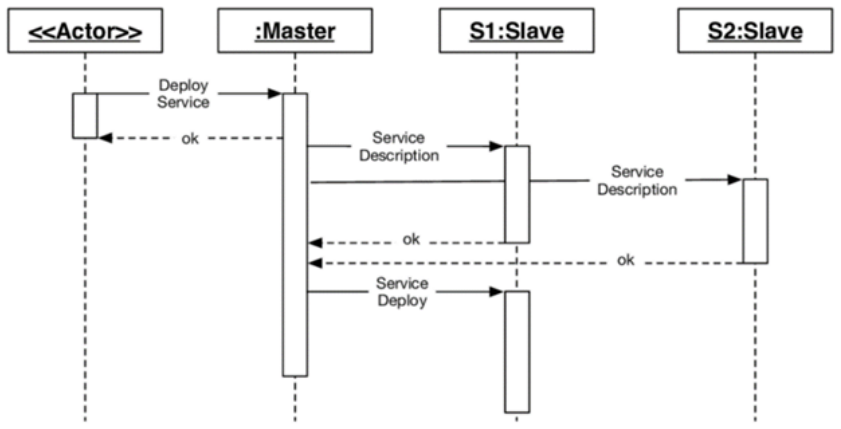


主从：

Master-slave

主从架构（Master-Slave Architecture）是一种常见的分布式系统设计模式，主要用于提高系统的性能、可靠性和可扩展性。在这种架构中，系统被划分为一个主节点（Master）和一个或多个从节点（Slaves）。主节点负责管理全局状态并协调各个从节点的操作，而从节点则执行具体的任务或者存储数据。

Master-Slave Architecture (MSA) is a common distributed system design pattern used to improve the performance, reliability and scalability of a system. In this architecture, the system is divided into a Master and one or more Slaves. The master node is responsible for managing the global state and coordinating the operations of the slave nodes, while the slaves perform specific tasks or store data.



事件总线/发布订阅模式

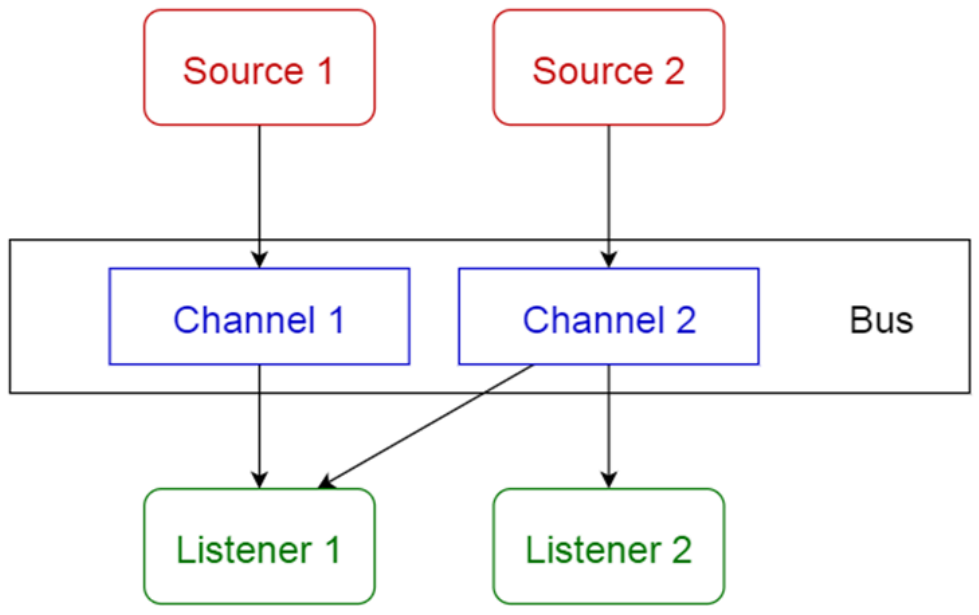
Event-bus/Publish–subscribe pattern

- 源（Source）发布事件到某个频道

  Source publishes an event to a channel
- 频道被分组到总线上

  Channels are grouped into bus
- 订阅者（Subscriber）订阅某个频道

  Subscriber subscribes to a channel



管道过滤器（Pipe-and-Filter）架构模式

- 数据来自源1和源2

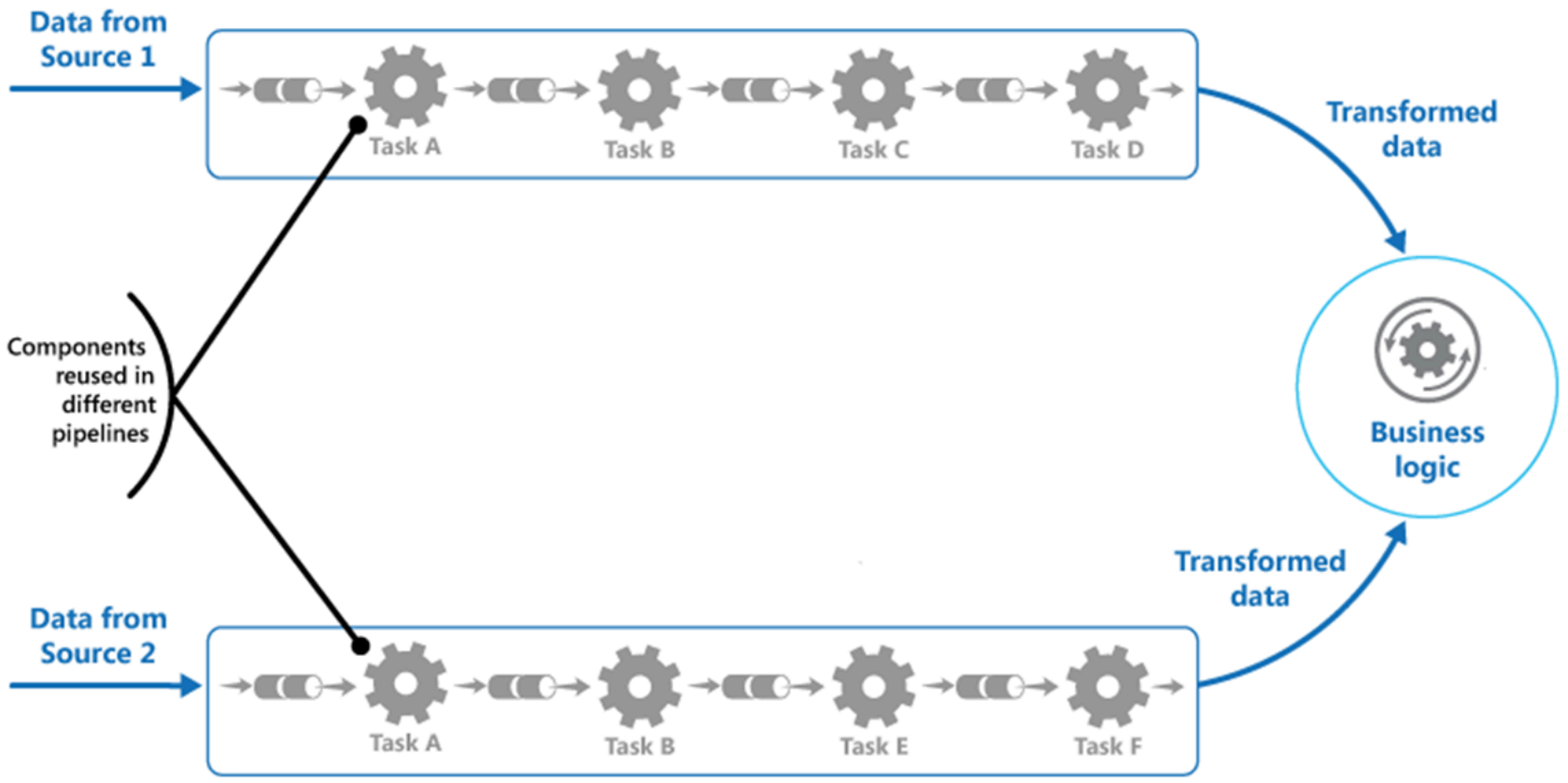
  Data from source 1 and source 2

- 数据经过一系列转换（任务A, B, C, D, E, F）

  Data undergoes a series of transformations (Tasks A, B, C, D, E, F)

- 转换后的数据进入业务逻辑组件

  transformed data goes to the business logic component



这种模式将数据处理分解成一系列可重用的过滤器组件，这些组件按顺序连接在一起形成管道。每个过滤器组件专注于特定的功能，比如数据清洗、转换或验证。这种模式有助于提高代码的可维护性和可扩展性，因为每个过滤器都可以单独开发、测试和替换。

This model decomposes data processing into a series of reusable filter components that are sequentially connected together to form a pipeline. Each filter component focuses on a specific function, such as data cleaning, transformation, or validation. This pattern helps improve code maintainability and extensibility because each filter can be developed, tested, and replaced individually.

分层架构

Layered pattern

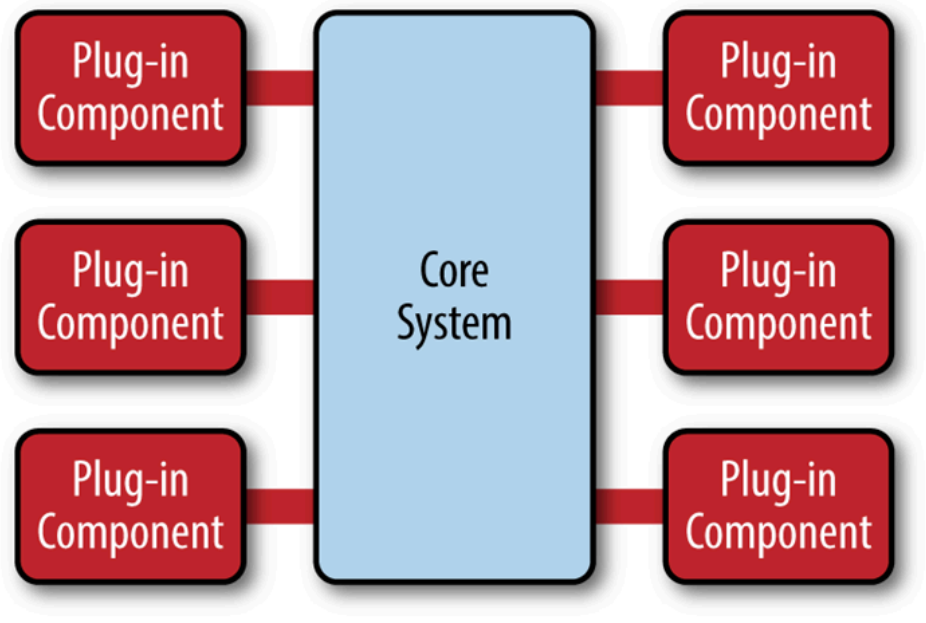
微内核（Microkernel）架构模式

- 核心系统是一个精简版系统

  A core system is a barebone system

- 其他功能以插件形式提供

  All other features are provided as plug-in



- 微内核提供核心功能集。

  Microkernel provides the core set of functionalities

- 内部服务器（InternalServer）为微内核添加额外的重要功能。

  ·*InternalServer* adds additional important functionalities to the microkernel.

- 外部服务器（ExternalServer）实现源自微内核提供的更高层次的功能。

  ·*ExternalServer* implements higher level functionality derived from the functionality offered by the microkernel.

- 适配器（Adapter）在用户（客户端）和系统（微内核和外部服务器）之间架起桥梁，处理异构系统组件。

  ·*Adapter* bridges between the user (client) and the system (microkernels and external servers) to handle heterogeneous system components.

微内核架构将操作系统的核心功能最小化，只保留最基本的服务，如内存管理、进程调度和基本的硬件抽象。其他功能和服务作为独立的模块（插件）运行在用户空间，而不是内核空间。这样做的好处包括更好的安全性和可扩展性，因为任何有问题的模块不会影响到核心系统，而且可以根据需要动态加载或卸载模块。

The microkernel architecture minimises the core functionality of the operating system, retaining only the most basic services such as memory management, process scheduling and basic hardware abstractions. Other functions and services run as separate modules (plug-ins) in user space, not kernel space. The benefits of this include better security and scalability, as any faulty modules do not affect the core system, and modules can be dynamically loaded or unloaded as needed.

模型-视图-控制器（Model-View-Controller，MVC）架构模式

- MVC常用于改善用户体验

  MVC is often used to improve the user experience

- 模型（model）代表核心功能和数据

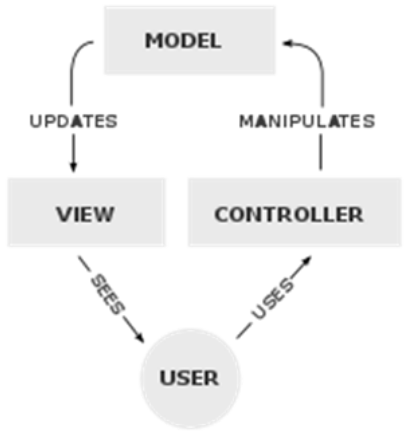
  model : represent the core functionality and data

- 视图（view）向用户显示模型的一部分

  view: displays a part of the model to the user

- 控制器（controller）接受用户输入并更新模型

  controller : accept user input and update models



Summary

- 软件架构存在于一个特定的上下文中（即在部署时满足质量属性的需求）

  Software architecture exists within a context (the quality attributes when delivering the functional requirements in the deployment time)

- 我们通过风格来划分系统模块及其连接，以解决质量属性的问题（例如，使用ADD方法）

  We divide the system into modules and their connections by styles to address quality attributes (e.g., through ADD)

- 存在多种架构风格，其中成功的会被总结成模式

  There are many architectural styles,The successful ones are summarized as patterns

- 架构描述应包含利益相关者的关注点以及选择某种架构的理由，这些通常以视图和观点的形式呈现

  An architectural description includes the stakeholders' concerns and the rationales of the architecture chosen. They are presented in the form of views and viewpoints.

5.Technical Debt

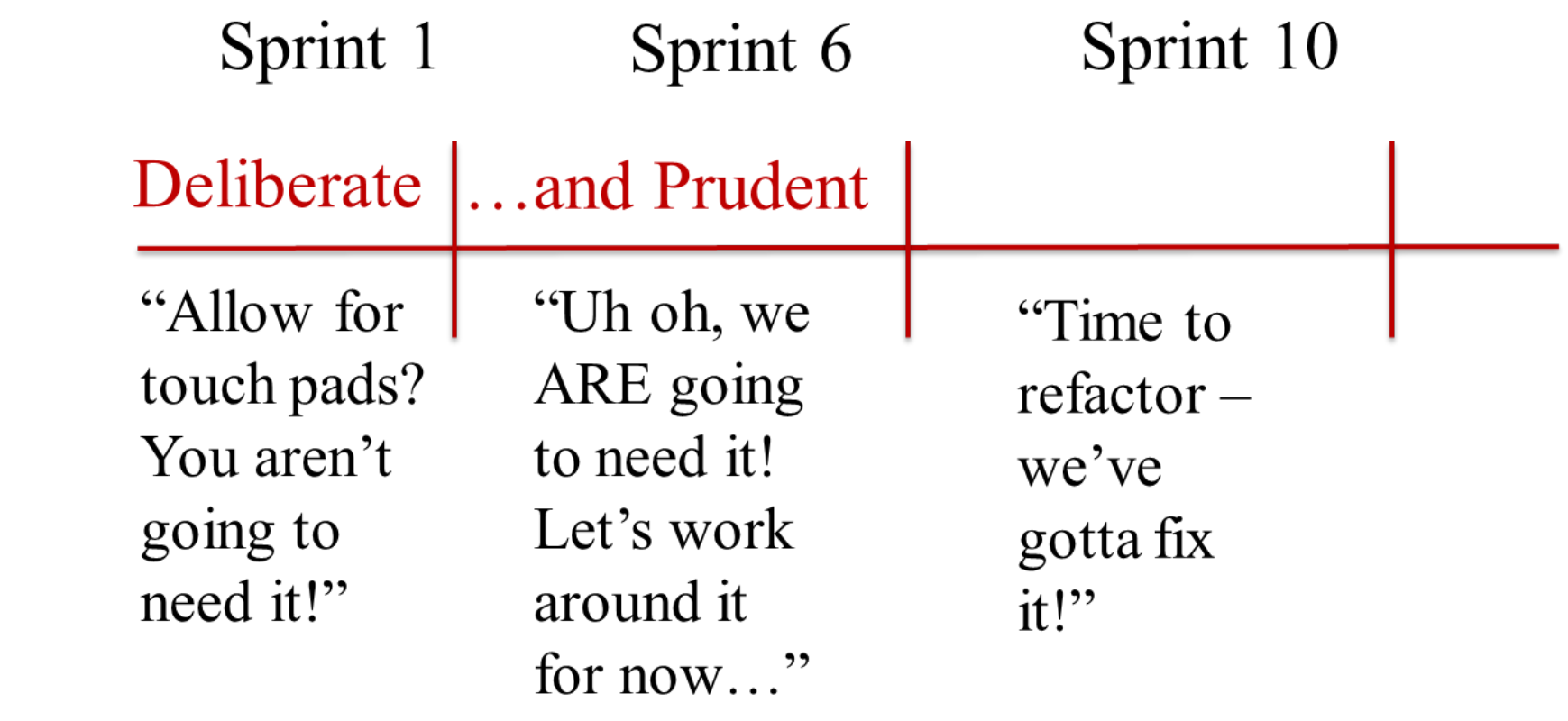
What is Technical Debt

- **技术债务 (TD)**:
  - 它是一个在1992年提出的隐喻 。这个隐喻表明，采用"快速而粗糙"的方式做事会产生技术债务。就像金融债务一样，技术债务也会产生利息支付，这些利息表现为未来开发过程中，由于这种快速而粗糙的设计选择，开发者必须投入额外的努力。

    is a metaphor originally proposed in 1992. The metaphor is that doing things in a "quick and dirty" way creates a technical debt (TD). Like a financial debt, TD incurs interest payments, which come in the form of the extra effort that developers must dedicate in future development because of this quick and dirty design choice.

  - 是一种短期内看似方便的设计或构建方法，但在长期内会造成技术环境，使得相同的工作比现在花费更多（包括随着时间推移的成本增加）。

    is a design or construction approach that is expedient in the short term but that creates a technical context in which **the same work will cost more to do later than it would cost to do now** (including increased cost over time)



技术债务的三个阶段：产生、支付利息、偿还

Three stages of technical debt: creation, interest payment, repayment

并非所有延迟或未完成的工作都是技术债务：

Not all delayed or unfinished work is technical debt:

1. 许多类型的延迟或不完整的工作并不是技术债务

   Many kinds of delayed or incomplete work are not technical debt:

   - 功能积压、延期的功能、削减的功能

     feature backlog, deferred features, cut features

2. 一般来说，如果"同样的开发工作在未来不会比现在更昂贵"，则它就不是技术债务。

   In general, if "the same (development) work **will not** cost more to do later than it would cost to do now", then it is not a technical debt.

技术债务的概念主要关注那些在短期内采取捷径或妥协，导致未来开发成本增加的情况。因此，判断某项工作是否为技术债务的标准在于它是否会增加未来的开发成本。

The concept of technical debt focuses on situations where shortcuts or compromises are taken in the short term, resulting in increased development costs in the future. Thus, the criterion for determining whether an undertaking is technical debt lies in whether it will increase future development costs.

尽管技术债务在实际项目中普遍存在，但它往往并未明确列在官方的积压工作中，而是通过其他方式进行跟踪。此外，对于减少技术债务来说，重构是一种重要的策略。

Although technical debt is prevalent in actual projects, it is often not explicitly listed in official backlogs and is tracked in other ways. In addition, refactoring is an important strategy for reducing technical debt.

在一些大型组织中，用于管理技术债务的开发时间相当大

In some large organizations, **development time** dedicated to managing Technical Debt is **substantial**

- 平均占总体开发工作的25%

an average of 25% of the overall development

- 有些甚至高达40%-50%

  Some said 40%-50%

许多公司倾向于用频繁发布新版本、高质量、快速反馈和小投入的长期能力来换取短期功能，这些短期功能可能会让他们快速解决问题，但从长远来看却会拖慢他们的步伐。为了追求短期利益而积累技术债务可能会损害公司的长期竞争力。这样的公司可能在短期内获得了一些优势，但在长期内却可能因为技术债务而变得脆弱，难以应对竞争压力。

Many companies tend to trade the long-term capabilities of frequent new releases, high quality, fast feedback, and small investments for short-term features that may allow them to solve problems quickly but slow them down in the long run. Accumulating technical debt in pursuit of short-term gains can harm a company's long-term competitiveness. Such firms may gain some advantage in the short term, but in the long term they may become vulnerable to competitive pressures because of technical debt.

业务部门希望尽快将产品推向市场并保护初期的投资，他们可能没有意识到这样做可能导致技术债务的产生。而技术部门则看到了技术债务带来的负面影响，包括生产力下降和无法专注于新功能的开发。这种差异反映了业务决策和技术实现之间的冲突，业务人员可能过于乐观地看待收益和成本，而技术人员则更加悲观地认识到这些问题。

Business departments want to bring products to market as quickly as possible and protect initial investments, and they may not realise that this can lead to the creation of technical debt. Technology departments, on the other hand, see the negative impacts of technical debt, including reduced productivity and an inability to focus on the development of new features. This difference reflects the conflict between business decisions and technical implementation, where business people may be overly optimistic about benefits and costs, while technical people are more pessimistic about these issues.

技术债务按维度分类：

Technical debts have been classified by dimension:

- 类型（例如设计、测试或文档债务）

  Type (e.g., design, testing, or documentation debt),

- 意图性（有意或无意）

  Intentionality (intentionally or unintentionally),

- 时间范围（短期或长期）

  Time horizon (short or long term)

- 关注程度

  Degree of focus.

(Popular) 技术债务管理理念： Technical Debts management idea

- 识别技术债务并通过软件项目的产品待办事项列表跟踪它们，随后讨论它们

  Identify a technical debt and **track** it through product backlog of the software project. **Discuss** them

- 示例：在截止日期前我们没有时间合并这两个数据库，所以我们会编写一些保持同步的胶水代码，在发货后进行合并

  E.g., put the following into the backlog: “We don't have time to reconcile these two databases before our deadline, so we'll write some glue code that keeps them synchronized for now and reconcile them after we ship.”

种类:

Types:

- 技术债务不只是代码问题。

Technical Debt is not just a coding problem

- 重构代码并不能解决所有问题。

  Refactoring out the code smells **cannot** solve them all.

技术债务除代码问题外，还包括架构、测试、文档等方面。

Technical debt includes architecture, testing, and documentation in addition to code issues.

意图性

Intentionality

有意： Intentional:

- 有意引起的技术债务

  Debt incurred by intention

- 例子：“我们现在没有时间为多个平台提供通用支持。我们将首先支持iOS，并稍后为Android等其他平台添加支持。”

  e.g., “We don’t have time to build in general support for multiple platforms. We’ll support iOS now and build in support for Android, etc., later.”

- 例子：“我们在项目的最后两个月里没有时间为所有代码编写单元测试。我们将在发布之后再编写那些测试。”

  e.g., “We didn’t have time to write unit tests for all the code we wrote the last 2 months of the project. We’ll write those after we release.”

无意： Unintentional:

- 因低质量工作而意外产生债务

  Debt incurred unintentionally due to low quality work

- 例子：初级程序员写出糟糕的代码

  e.g., A junior programmer writes bad code

- 例子：主要的设计策略（比如决定使用模型-视图-控制器架构连接软件的所有主要模块）效果不佳

  e.g., A major design strategy (e.g., decide to use the model-view-controller architecture to connect all major modules in the software) turns out poorly

时间范围:

Time Horizon

短期:

Short-Term

- 常常因短期战术原因被动产生

  Usually incurred **reactively**, for tactical reasons

- 例子：跳过一些集成测试以便快速发布产品

  e.g., Skipping some integration tests to get a release out the door

- 例子：“我们没时间正确地实现这个功能；先用临时方案，以后再修复。”

  e.g., "We don't have time to implement this the right way; just hack it in, and we'll fix it after we ship."

长期:

Long-Term

- 常常因长期战略原因主动产生

  Usually incurred **proactively**, for strategic reasons

- 例子：“我们认为至少三年内不需要支持第二个平台，所以我们的设计只考虑一个平台。”

  e.g., “We don’t think we’re going to need to support a second platform for at least 3 years, so our design supports only one platform.”

关注程度：

Degree of Focus

Focused Debt（专注型债务）：有意积累并管理的债务。 Debts that are intentionally incurred and managed.

- 例子：为加快上市速度而故意采取的捷径

  Deliberate shortcuts for time-to-market

- 例子：为了更快交付而进行的不充分测试

  Inadequate testing for quicker delivery

  Unfocused debt（非专注型债务）：当团队在管理和解决技术债务方面没有明确的战略或优先级时积累的债务。

  Debts accumulated when the team has no clear strategy or priority for managing and addressing technical debts

- 例子：新增特性时没有添加单元测试，导致易碎的代码库。

  New features are added without unit tests, leading to a fragile codebase that breaks easily.

- 例子：应用程序使用了已知安全问题的老框架，并且没有更新计划，增加了风险。

  The application uses an old framework with known security issues and no plans for updating, increasing risk.

Identify Technical Debts

技术债务增加的常见模式：

There are some well-known patterns in software development to *increase* technical debts

- **时间压力**（Schedule Pressure）：当一个团队受到不合理的时间承诺约束时，他们很可能会采取捷径以满足管理层的期望。例如，新特性的范围逐渐扩大、团队构成的变化以及延迟集成等都会导致这种压力。

  Schedule Pressure:When a team is held to a commitment that is unreasonable, they are bound to cut corners to meet management expectations. E.g., creeping scope of new features, change in team composition, late integration

  - 解决方式：采用更加灵活的计划方式，以便更好地应对变化和不确定性

    Solution: Adopt a more flexible planning approach to better cope with change and uncertainty

- **代码复制**（Duplication of Code）：许多原因导致代码复制例如成员经验缺乏、复制粘贴习惯、压力迫使

  Many reasons for code duplications: Lack of experience on part of the team members, Copy-and-paste programming practice , Pressure to deliver

  - 解决方法：使用静态分析工具辅助；结对编程，提升团队成员的知识和能力；立即偿还债务或跟踪记录稍后再还

    Solution: Use static analysis tool for assistance ; Pair programming To spread the knowledge and improve competence of team members; *Either* (a) Repay debts now *or* (b) Track it if replay later,

- **尝试一次做好**（Get it “right” the first time）：开发初期就创建大量强大的设计，导致产品被过度设计，致使修复成本持续增长。

  create a lot of powerful design up-front, Resulting in an over-engineered product that continues to increase the cost of fixing it

Manage Technical Debts

如果我们无法避免技术债务，就必须对其进行管理。 这就意味着要认识它、跟踪它、对它做出合理的决策，并防止它带来最严重的后果。

If we can’t avoid technical debt, we must manage it. This means recognizing it, tracking it, making reasoned decisions about it, and preventing its worst consequences.

追踪和衡量技术债务

Tracking Technical Debts

- 跟踪技术债务（TD）对于管理它们至关重要。

  Tracking technical debts (TD) is necessary to manage them.

  - 所有“良好债务”都可以被追踪（至少从定义上来说）

    All “good debt” can be tracked (at least by definition)

  - 将其作为缺陷记录下来或包括在产品待办事项中

    Log as defects or Include in product backlogs

  - 监控项目速度

    Monitor project velocity

  - 监控返工量

    Monitor amount of rework

- 测量债务的方式

  Ways to Measure Debt

  - 产品待办事项中的总债务

    Total of debt in product backlog

  - 维护预算（或维护预算的比例）

    Maintenance budget (or fraction of maintenance budget)

  - 用金钱而非特性来衡量债务，比如，“50%的研发预算用于非生产性维护工作”

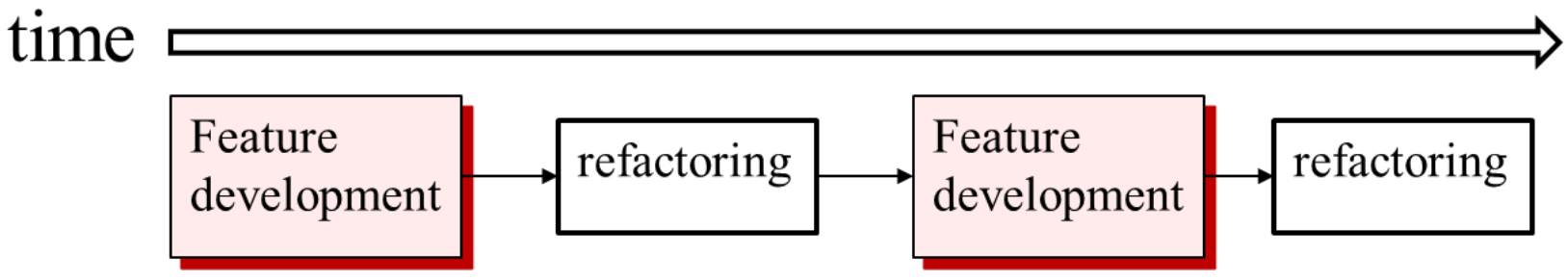
    Measure debt in money, not features, e.g., “50% of R&D budget is nonproductive maintenance work”

偿还技术债务

Repay Technical Debts

- 一个有效的策略是定期预留时间来处理技术债务。

  An effective tactic is to reserve a timeslot on a periodic basis to deal with technical debt.



- 可以通过延长重构的时间来减少重构周期的数量，反之亦然。

  Longer time for refactoring to trade for smaller number of refactoring cycles, and vice versa.

- 首先优先考虑解决最严重的问题区域的重构任务。

  Prioritize refactoring tasks to address the critical areas first.

- 涉及到产品负责人和利益相关者，获取他们的观点。

  Involve product owners and stakeholders to get their perspectives

在软件开发过程中，我们需要平衡新功能开发和代码重构的时间，确保代码质量的同时保持项目的进度。同时，优先处理最关键部分的重构任务，并且让产品负责人和利益相关者参与到这个过程中，可以更好地管理和偿还技术债务。

During software development, we need to balance the time between new feature development and code refactoring to ensure code quality while keeping the project on schedule. At the same time, prioritising refactoring tasks for the most critical parts and involving product owners and stakeholders in the process allows for better management and repayment of technical debt.

讨论技术债务

Discuss Technical Debts

- 帮助教育技术人员了解项目中的商业决策过程，使他们融入业务环境。

  Helps to educate technical staff about business decision making involved in the project. Fit developers into the business context

- 帮助教育业务人员了解技术决策，使其成为他们商业或运营决策的输入之一。

  Helps to educate business staff about technical decision making, Also serve as an aspect of the inputs to their business or operational decisions

- 提高对项目中经常被掩盖的重要问题的认识和透明度，作为一种风险规避措施。

  Raises awareness/transparency of important issues that are often covered up in the project, As a risk aversion measure

- 允许更明确和直观地管理技术债务，作为逐一解决问题的起点。

  Allows technical debts to be managed more explicitly and visually, As a starting point for a project resolve them one by one

讨论技术债务可以帮助团队成员更好地理解和沟通，提高整体效率和项目成功率。

Discussing technical debt can help team members understand and communicate better, improving overall efficiency and project success.

State of the Practices in Managing Technical Debt

实际管理技术债务的一些策略现状：

Managing Technical Debts in Practice

- 虽然大多数开发者可能希望有更好的方法来做这些事情，但他们并没有这样做。

  Although most developers may have expressed a desire for better ways of doing all these things, and yet they did not do it.

- 工作场所中的策略（来自一项针对26家公司的调查）：

  Strategies in workplaces (from a survey on 26 companies)

  - 不采取任何行动——“如果没坏，就别修它”——因为债务可能永远不会被客户看到。

    Do nothing—“if it ain’t broke, don’t fix it”—because the debt might not ever become visible to the customer.

  - 使用风险管理方法来评估和确定技术债务的成本和价值。例如，在每个发布周期中分配5%至10%的时间来处理技术债务。

    Use a risk management approach to evaluating and prioritizing technical debt’s cost and value. E.g., allocate 5 to 10 percent of each release cycle to addressing technical debt.

  - 管理客户和非技术利益相关者的期望，让他们成为平等的伙伴，并促进关于债务影响的开放对话。

Manage the expectations of customers and nontechnical stakeholders by making them equal partners and facilitating open dialogue about the debt's implications.

- 与开发团队一起进行审计，使技术债务变得可见并使用积压工作/任务板进行跟踪。

  Conduct audits with the development team to make technical debt visible and explicit; track it using a backlog/task board

有效： **Effective**

- 积压工作、静态分析器和"lint"程序都可以增加追踪级别。它们也是开销最低的工具，因此似乎是当前追踪TD的最佳实践。

  Backlogs, static analyzers and "lint" programs all increase the tracking level, but we cannot see a big difference (although static code analyzers seem to contribute better to the participants' awareness). They are also the ones with the least overhead. They therefore seem to be considered the **best practices** at the moment **to track TD**.

- 积压工作是参与者中最常用的工具。特别是，最常用的积压工作工具有Jira、Hansoft和Excel。

  Backlogs are the most used tool among the participants. In particular, the most used backlog tools are Jira, Hansoft, and Excel.

无效： **Ineffective**

- 注释可以提高意识，但不被认为是追踪

  Comments in the code help awareness, but they are **not considered tracking**

- 文档可以提高TD意识，但它不被视为高水平的追踪，而且开销最高。

  Documentation increases TD awareness, but it is **not considered as a high level of tracking**, and it has the **highest overhead**

- 使用bug系统追踪TD被认为不如其他技术贡献大，且开销稍高。

  Using a **bug system** for tracking TD is not considered as contributing to a better level of awareness or tracking compared to the other techniques, and it has a slightly higher overhead

- 测试覆盖率似乎对意识和追踪水平贡献不大，虽然开销不高。

  Test coverage does not seem to **contribute** too much **to the awareness and tracking level**, although it does not involve much overhead

6.Code with Quality

在敏捷开发方法中编写代码的过程。首先，开发者会草拟软件设计，然后逐步编写代码，并专注于完成我们需要关注的任务。

In an Agile method, developers sketch the software design, write codes incrementally, and focus on delivering *the tasks we pay attention to*.

由于这个过程没有长期规划（即非系统性），不可避免地会出现错误。同时，我们的代码可能会有意无意地与这些定制任务的特点紧密相连。

We unavoidably **make mistakes** during this "no-long-term-planning" (i.e., unsystematic) process. Our code is **(un)intentionally** coupled with the customized nature of these tasks.

随着开发者对要构建系统的理解加深，设计和代码很可能会发生变化。因此，基本策略是确保我们的代码保持干净、最小化并且可维护。

The design and code will likely change as developers' understanding of the system to build is deepened. The basic strategy is: Our code should be **clean, minimal, and maintainable**.

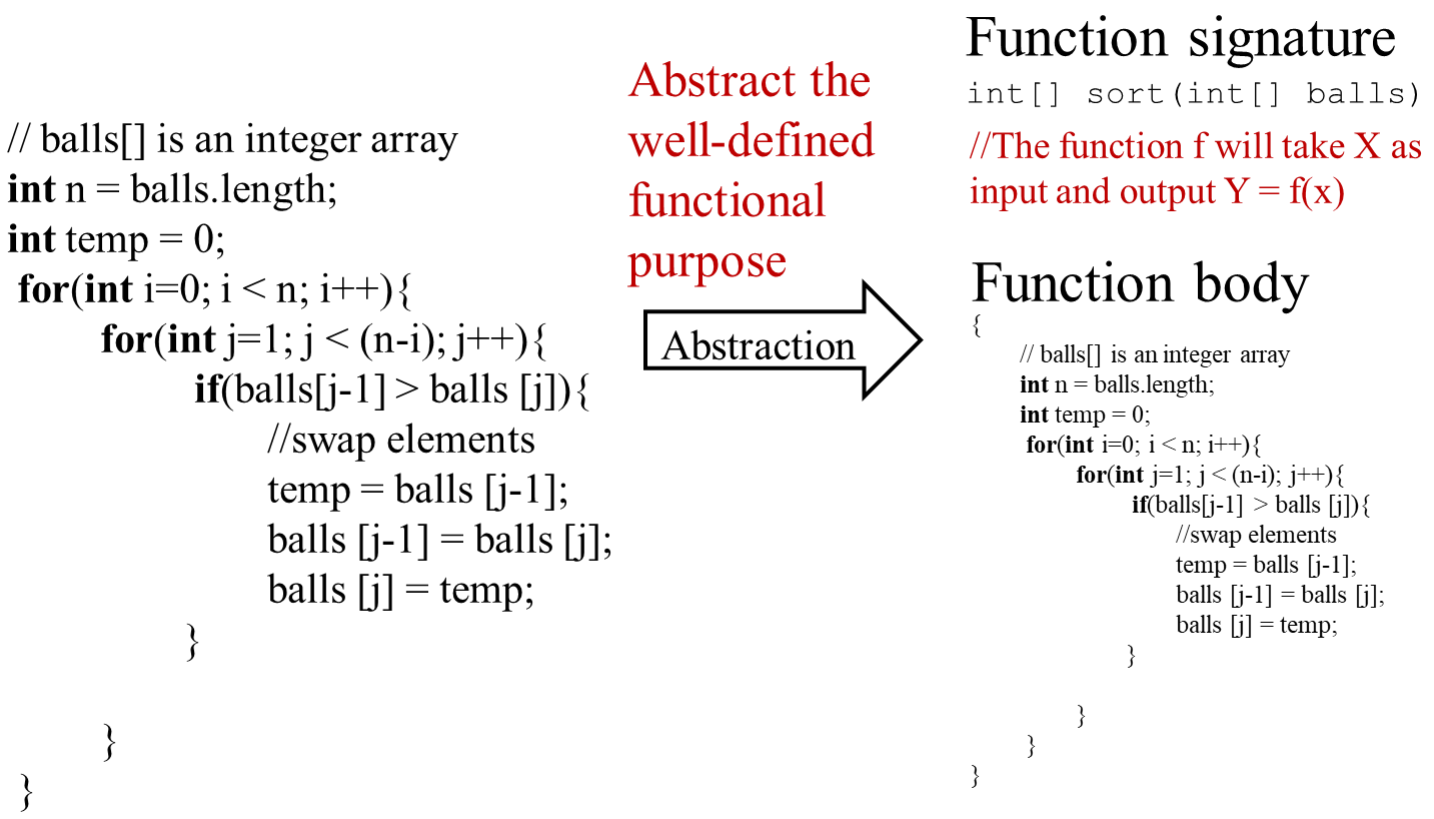
Basic Code Design Principles

Abstraction（抽象）：指将复杂细节隐藏起来，只保留必要的信息来解决问题。

It refers to hiding complex details and keeping only the necessary information to solve the problem.

例如下图将代码抽象成为函数名+实现细节的形式

For example, the following diagram abstracts the code into the form of function name + implementation details



抽象是一种重要的编程理论概念，旨在避免信息和/或人力的重复。当有一个具有明确功能的代码片段C时，可以将其放入具有特定函数签名F的函数体内，从而形成一个抽象。好处如下：

Abstraction is a fundamental concept in programming theory. It aims to avoid duplication of information and/or human effort. Let C be a piece of code with a **well-defined purpose**, we can put C into the body of a function having a function signature F to form a abstraction.

- 程序P只需包含一次C，而在需要的地方调用F，而不是多次复制C，减少了冗余代码。

  Instead of copying C multiple times, program P needs to include C only once and call F where it is needed, reducing redundant code

- F的用户（client code U）不需要理解C的具体实现，他们可以依赖于F及其明确的功能来完成自己的任务。

  Users of F (client code U) do not need to understand the specific implementation of C. They can rely on F and its explicit functionality to fulfil their tasks.

- 如果C发生变更或修订，这些更改仅限于F，不会影响到U。

  If changes or revisions are made to C, these changes are limited to F and will not affect U.

- U变得更加易读，因为它的逻辑集中在实现自身的目标上。

  U becomes more readable because its logic is focussed on achieving its own goals.

除了将代码C隐藏在一个具有明确功能的函数F中之外，抽象还可以包括寻找相似性和/或公共方面以及忽略不重要的差异的过程。如果有一组具有相似功能目的的代码{C1, ..., Ci, ..., Cn}，那么F是它们的一个（公共）抽象。这意味着：

In addition to hiding code C in a function F with an explicit function, abstraction can also include the process of finding similarities and/or common aspects and ignoring unimportant differences. If there is a set of codes with similar functional purposes {C1, ... , Ci, ... , Cn}, then F is a (common) abstraction for them. This means:

- 在C1, ..., Ci, ..., Cn之间个体参数的不同被忘记，并统一为F中的相同参数集。

  nThe differences in parameters for individual ones among C1, ..., Ci,..., Cn are forgotten and unified as the same set of parameters in F.

- C1, ..., Ci, ..., Cn之间的相似之处被组织在F的函数体中，而它们之间的差异则放在不同的控制流结构下。

  Similarities among C1, ..., Ci,..., Cn are structured in F's function body, and differences among them are put under different parts of the control flow structure in C.

- 在面向对象编程中，可以通过创建类层次结构来实现这种抽象。

  Or in OO programming, we create a class hierarchy

- F的功能目的是C1, ..., Ci, ..., Cn的个体功能目的的共同点。

  The functional purpose of F is the **common** functional purpose of the individual functional purposes for C1, ..., Ci,..., Cn

调用F可以让U的开发者减少集成的努力。例如，理解F的功能比阅读所有代码细节更简单；避免显式地维护代码列表。这使得U更具可维护性；此外，F在U中的通用抽象可以帮助U忽略由F抽象掉的所有差异，即保持干净和最小化

Rather than including {C1, ..., Ci, Cj,..., Cn} in the client code U, calling F can make developers of U incur less effort in integration.E.g., understanding the functional purpose of F is easier than reading all the code details; avoiding maintaining the code list explicitly, which makes U more maintainable. The common abstraction F used in U helps U to ignore all differences among the details abstracted away by F, make it clean and minimal

数据抽象是在功能抽象的基础上，对与数据结构紧密相关的函数进行抽象和分组，使其代表数据结构的一种单个功能目的。具体来说：

Data abstraction is the abstraction and grouping of functions closely related to a data structure so that they represent a single functional purpose of the data structure, based on functional abstraction. Specifically:

- 功能抽象是对代码中的过程步骤进行抽象。

  Functional abstraction is to abstract the procedural steps in code

- 如果我们抽象并分组与数据结构密切相关的一组函数，则这个集合代表了数据结构的单一功能目的。

  If we abstract and group functions **critically related to** a data structure into a set, the set represents a **single functional purpose** of the data structure.

- 具有功能抽象的函数等同于数据结构的操作符。

  Function with functional abstraction = **operators** on the **data structure**

- 函数名等于接口。

  functional signatures = **interfaces**

数据抽象则是将数据结构上的操作符建模为一组相关函数（称为接口），其完成数据结构内的操作例如入栈出栈等。相比之下，数据结构的实现被这些接口所隐藏。数据抽象时一定要注意不要暴露数据结构的内部实现

Data Abstraction is to model the operators on a data structure as a set of *related* functions (called **interfaces**). In contrast, the data structure implementations are concealed by the interfaces.

简而言之，功能抽象让你专注于“做什么”，而数据抽象则让你专注于“怎么组织和管理数据”。

In short, functional abstraction lets you focus on "what to do", while data abstraction lets you focus on "how to organise and manage data".

- Good example

`Stack()`	create a new stack
`s.isEmpty()`	is `s` empty?
`s.length()`	number of items in `s`
`s.push(int x)`	push `x` onto `s`.
`s.pop()`	remove and return the item most recently pushed onto `s`.

- Bad example

`Stack()`	create a new stack
`s.isEqual(int[] X)`	is `s` equal to the integer array `X`?
`s.length()`	number of items in `s`
`s.set(int x,int i)`	put `x` at the position `s[i]`.
`s.get(int i)`	remove and return the item `s[i]`, shorten the stack `s`.
`s.push(int x, int i)`	implemented as `s.set(int x, int i)`

上图中坏的例子并没有遵守栈的先进后出特性，而且它试图将栈当作一个普通的数组来操作，这是不符合数据抽象原则的。一个好的数据抽象应该只提供那些符合数据结构特性的操作，而不是暴露其内部实现细节

The bad example in the diagram above does not observe the first-in, first-out nature of the stack, and it attempts to manipulate the stack as if it were a normal array, which is not consistent with data abstraction principles. A good data abstraction should provide only those operations that are consistent with the characteristics of the data structure, rather than exposing its internal implementation details

坏的例子看似能适应更多环境，但其不遵循数据抽象规则，在调用时并不简洁，容易引起问题。

Bad examples may seem adaptable to a wider range of environments, but they don't follow the data abstraction rules, aren't succinct when called, and are prone to causing problems.

Consider two pieces of client code using **Bad Example**:

```
for (int i = 0; i < n; i++) { …;s1.push(i,i);…}
for (int i = n; i > 0; i--)  { …;s2.push(i,i);…}
```

Consider two pieces of client code using **Good Example**:

```
for (int i = 0; i < n; i++) { …;s1.push(i);…}
for (int i = n; i > 0; i--)  { …;s2.push(i);…}
```

好例子可以轻易逆序入栈，而坏例子这样调用却无法轻易逆序。要实现逆序入栈势必会增加复杂度。

Good examples can be easily reversed into the stack, while bad examples called this way cannot be easily reversed. Implementing a reverse order stack is bound to increase complexity.

- 每次调用一个抽象数据类型的运算符（如push()）后，数据结构的完整性应该保持不变。为了确保这一点，对数据结构所做的任何改变都应仅由这些运算符执行。

  The integrity of the data structure should remain intact after each call to an operator of an abstract data type (e.g. push()). To ensure this, any changes made to the data structure should be performed only by these operators.

- 对于使用好例子中的堆栈接口，检查数据结构完整性的过程可以更轻量级，从而产生更高效且复杂的代码。

  For using the stack interface in the good example, the process of checking the integrity of the data structure can be more lightweight, resulting in more efficient and complex code.

Information Hiding（信息隐藏）：模块内部的数据对外部不可见，减少模块之间的耦合度。

Information Hiding: The data inside the module is not visible to the outside world, reducing the coupling between modules.

- 接口应该尽可能少地揭示其内部运作方式。

  Interfaces should be chosen to reveal **as little as possible about** its **internal workings**.

- 考虑接口中操作符参数的详细程度是否必要

  We should ask whether the level of details specified in the parameters of an operator in an interface is essential.

接口设计应该尽量减少暴露不必要的细节，以提高安全性并降低耦合度

Interface design should minimise the exposure of unnecessary details to improve security and reduce coupling

Encapsulation（封装）：把数据和操作数据的方法绑定在一起，对使用者来说不需要知道具体实现细节。

Binding data to methods that manipulate it is not necessary for the user to know the implementation details.

- 封装是指保护X内部的信息不被外部获取。

  Encapsulation on X refers to **ensuring** outsiders cannot **gain knowledge** about certain information internal to X.

- X可以是类、模块、文件、数据记录等。

  X can be a class, a module, a file, a data record, etc.

如果我们能用最少的操作知识描述功能目的，并且函数签名没有超出功能目的所声明的内容，我们就实现了良好的封装。否则，客户端代码可能会利用这些知识来定制代码，破坏数据结构的完整性。

We achieve good encapsulation if we can describe the functional purpose using the least “working known-how” of the operators of a data structure and the functional signature does not specify more than what the functional purpose states. If not, the client code U knows the details of the function. If U uses this knowledge to make customized code, the integrity of the data structure will be compromised.

Coupling（耦合）：模块间相互依赖的程度，尽量降低耦合度以提高灵活性。

Coupling: the degree of interdependence between modules, minimising coupling to increase flexibility.

- 耦合关注的是语法上的依赖关系。

  Coupling is about **syntactic** dependency

- 我们的目标是减少代码中的耦合度。

  Our aim: minimize coupling in the code listing

- 实体X在编译时结构上直接依赖于另一个实体Y。这意味着存在一个从X到Y的直接结构依赖。

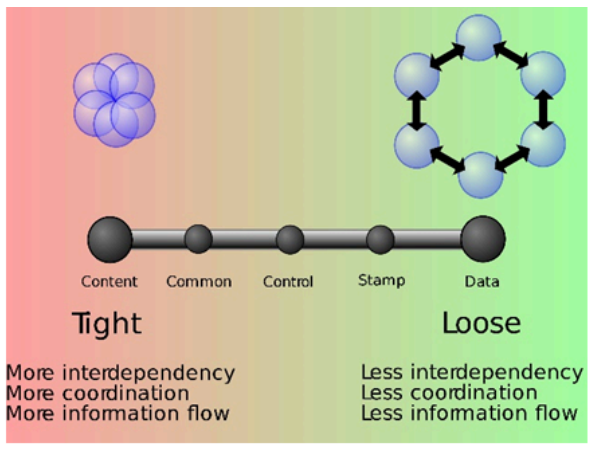
  Entity X has a direct compile-time structural dependency on another entity Y. This implies that there is a direct structural dependency from X to Y.

- 程序员需要这样的依赖信息来提高计算效率。编译器也需要这些信息来进行代码优化。

  Programmers need such dependency information to improve computational efficiency. Compilers also need this information for code optimisation.

减少编译时的耦合或延迟依赖关系可以改善程序的耦合情况。

Reducing compile-time coupling or delayed dependencies can improve the coupling of a programme.



worse ⟶ better

Cohesion（内聚）：模块内部各部分之间的关联程度，高内聚表示模块内部功能相对集中。

Cohesion: the degree of association between parts within a module, with high cohesion indicating a relative concentration of functionality within the module.

- 我们无法从代码中看出对内聚的依赖。

  We cannot see the dependency on cohesion from the code

- 如果两个实体有联系，它们应该被分组在一起并一起使用。

  Our aim:If two entities are related, they should be grouped together and/or used together.

更高的内聚和更低的耦合会带来更好的质量，包括可维护性、可重用性和较低的复杂性。

Higher cohesion and lower coupling leads to better qualities, including maintainability, reusability and lower complexity.

- 7 types of cohesion
  - Coincidental (grouped without a reason)
  - Logical (grouped because “we” treat them as one category)
  - Temporal (A part of each entity in the group is processed similarly)
  - Procedural (A group of functions, when composed, represent different sequences of control of abstraction (e.g., a file))
  - Communicational (grouped because of operating on the same data)
  - Sequential (the input of one depends on the output of another)
  - Functional (to perform different aspects of a well-defined task)

worse ↓ better