**Scrum** 不是构建产品的一种过程或一项技术,而是一个框架,在这个框架里可以应用各种流程和技术。Scrum 能使产品管理和开发实践的相对功效显现出来,以便随时改进。 Scrum不是由一个大团队用很长的时间来开发一个大产品，而是由一个小团队用很短的时间来开发一个小功能。但定期集成，以构成整体。Scrum模式不会硬性规定任何具体的工程实践，这些都由团队自行决定，不过，在实践中，不纳入XP的实践而通过Scrum模式取得成功是非常困难的。

**Scrum 框架**包括 Scrum 团队及其相关的角色、事件、工件和规则。框架中的每个模块都有一个特定的目的,对 Scrum 的成功和使用都至关重要。 三个角色：产品负责人，ScrumMaster，团队四个仪式：Sprint计划会议，每日站会，Sprint评审会议，Sprint回顾会议三个物件：产品Backlog，SprintBacklog，燃尽图

**产品负责人**：负责最大化产品以及开发团队工作的价值。是管理产品待办事项列表的唯一责任人。负责待办事项列表（1清晰地表达产品代办事项列表条目2对产品代办事项列表中的条目进行排序,最好地实现目标和使命3确保开发团队所执行工作的价值4确保产品代办事项列表对所有人可见、透明、清晰,并且显示Scrum团队的下一步工作5确保开发团队对产品代办事项列表中的条目达到一定程度的理解）

**ScrumMaster**：负责确保 Scrum 被理解并实施（以各种方式服务于开发团队,包括:1指导开发团队自组织和跨功能2导并领导开发团队创造高价值的产品3移除开发团队进展过程中的障碍4按需推动Scrum事件5Scrum还未完全被采纳和理解的组织环境下指导开发团队6Coach and Teacher）

**Scrum团队：1**自组织团队选择如何最好地完成他们的工作,而不是由团队外的其他人来指使他们 2跨功能团队拥有完成工作所需要的全部技能,不需要依赖团队外部的3 Scrum 团队模式的目的是最大限度地优化适应性、创造性和生产力4开发团队不包括如测试或业务分析等负责特定领域的子团队。5人数3-9人（推荐5人，最多不要超过9人）

**Sprint：**将整个开发时间划分为多个短小的、固定的迭代周期或Sprint（通常为两周或三周）。开发团队自行决定每个迭代周期要完成多少个产品需求清单项。每个迭代周期最后都要演示已通过测试、能够发布的版本。

**产品Backlog：**是Scrum的核心，是按重要性排序的需求或故事（Story）的列表（客户语言描述的客户需求）1、作为[角色]，我想要[功能]，以便得到[价值] 2、为了[价值]，作为[角色]，我想要[功能] 编写方式**：**1选择一个最重要的用户，描述其user story。2再选择下一个用户，描述其user story,如果和前面的有重叠，则可以在前一个user story中添加一个用户。3迭代进行以上过程，直到写完所有user story。4对以上user story按照优先级进行排序。

**Sprint计划（全员参与，Product Owner也必须参与）**

1.计划准备（由ScrumMaster完成）：在sprint计划会议之前,要确保产品backlog的井然有序：一个产品仅一个backlog和负责人，所有backlog已根据重要性评分，产品负责人理解每个故事的含义（backlog由product owner提供，scrum master可以添加条目，但重要性归product owner写，backlog重要性只用大小代表重要性）

2.计划目标：1sprint目标2团队成员名单(以及他们的投入程度,如果不是100%的话)3 sprintbacklog(即sprint中包括的故事列表) 4确定好sprint演示日期 5确定好每日scrum会议的时间地点。

3.为了避免无休止的计划会议，需要一个会议日程安排表

4.为何ProductOwner参与会议：每个故事都有范围，重要性，估算三个变量，他们两两之间有强烈的依赖。前两点ProdutOwner设置，估算由团队设置，在会议上通过面对面对话，这三个变量会逐步调优。

5.确定Sprint长度：Sprint持续多久才算合适(实验总结最佳长度：两个星期)

6.确定Sprint目标：必须使用业务术语而非技术术语

7. 决定Sprint中要包含的Story：Sprint Backlog是产品Backlog的一个快照，包含了团队在下一个Sprint中能够完成的所有的Story。Sprint中包含的Story是由团队决定的, 会议上要对每个Story进行讨论，完成估算、重订优先级、确认细节、拆分。可以使用索引卡把握backlog记录story细节。

8.每日例会的时间和地点：这必须是每一个成员都能接受的时间和地点。

9.确定技术故事：需要完成但是不属于可交付物的东西

Sprint Backlog的形式白板、燃尽图：表示生产速度，提醒团队修改生产速度

每日例会：不超过15分钟，回答三个问题：昨天我做了什么。今天准备干什么。你遇到了什么障碍，需要其他人如何帮你。例会结束就要计算剩余工作故事点并更新燃尽图

**Sprint演示会议：** 定义Sprint结束于演示：1其他人可以了解你的团队在做什么2团队得到认可，团队成员感觉很好3不同的团队得到交流，讨论各自工作4演示可以吸引相干人士的关注，并得到重要的反馈5演示会迫使团队真正完成一些工作而不是貌似完成，这样不会污染下一个Sprint。

演示的检查列表：1确保明确阐述Sprint目标2集中精力演示可以实际工作的代码3演示保持快节奏4演示我们做了什么而不是我们怎么做的5不演示细碎bug的修复和微不足道的特性

**Sprint回顾：**是仅次于Sprint计划的第二重要的事件，这是做出改进的最佳时期。

根据要讨论的内容范围,设定时间为1至3个小时。参与者:产品负责人,整个团队还有我自己。在不受干扰的情况下讨论，指定某人当秘书。使用白板

流程：1.Scrum master展示sprint backlog,对sprint做总结。包括重要事件和决策等。2轮流发言。讲述他认为什么是好的,哪些可以做的更好,哪些需要在下个sprint中改变。3对预估生产率和实际生产率进行比较。4.Scrum master 对具体建议进行总结,得出下个 sprint 需要改进的地方。

Scrum中多数实践是非技术实践如用户故事、扑克牌估算等，实施起来并不复杂，但必须结合持续集成、测试驱动开发、简单设计、重构等一系列技术实践，否则不可能成功。

**测试驱动开发**意味着你要先写一个自动测试，然后编写恰好够用的代码，让它通过这个测试，接着对代码进行重构，主要是提高它的可读性和消除重复。整理一下，然后继续。

**估算**：在估算速度时，估计我们能在一个迭代周期内能够完成的story point

对于时间估算使用**计划纸牌**：1每个团队成员拿到一组卡片2.产品负责人或者一名团队成员扮演阅读者的角色，他负责阅读需要估算产品Backlog的条目，并且询问大家是否有疑问。

3. 团队讨论这个条目。4.每个团队成员按包括照自己的想法给出估算结果，并且选择对应的扑克出牌，估算结果不能告诉其他人。5.阅读者向大家确认是否都已经确定估算结果，确认后，大家同时展示估算结果。6.团队评估不同的估算结果.我们是否想法一致？我们是否存在分歧？有没有什么是我没有考虑到的？团队共同讨论估算的差异，最终达成一致。如果差异不可接受，返回3。7. 返回2，开始估算下一个条目。

估算有差异的原因：需求理解不同，技术不熟悉

为何团队估算：团队的智慧要高于某一个人的智慧，参与工作的人做出的估计要高于其他人的

估算的作用：1估算经常是错误的，当时估算过程仍然是有用的。可以发现误解、不一致以及需要进一步调查的地方2团队共同估算可以让大家建立对工作一致的理解。

**XP极限编程a method in or approach to software engineering**

极限编程是肯特•贝克于20世纪90年代中期创立的软件开发方法。该方法以简洁、沟通、反馈、勇气和尊重等价值观为基础。XP方法是与Scrum并行发展的，实际上包含了大多数相同要素。1结对编程：在一台工作站上进行结对编程，从而使学习效果最大化、设计质量最大化、缺陷最小化。2测试驱动开发：采用测试代码驱动系统的设计。编写自动化测试脚本，然后编写刚刚足够的代码以使其通过测试，然后从根本上重构代码，提高其可读性，移除重复代码。清理并重复这一过程。3集体代码所有权：允许（实际上是鼓励）开发团队的任何人编辑代码库的任何部分。这样可营造出团队所有权的意识，确保整个系统的设计都一致、易于理解。4增量式设计改进：从最简单的设计开始，然后运用重构等技术持续不断地改进设计，而不是一开始就做好完整的设计。

**XP应对不同风险的方式：**

ScheduleSlips：缩短发布、迭代，任务周期，XP优先做高优先级的任务，所以延期的一般是低优先级的

Project canceled：XP要求客户选择最小发布范围，可以包括大部分的业务目标，所以生产之前取消损失不大

Systemgoes sour：XP创建了覆盖面很大的测试集合，每次变更后都会执行以保证质量

Defect Rate：XP的测试有两个方面，从程序员出发的每个方法一个测试和客户出发的每个程序feature一个测试

业务目标误解：XP让客户成为团队的成员，项目规格在开发过程后总不断的重定义

业务目标更改：XP缩短发布周期，发布过程中欢迎客户可以用新功能替换未完成功能

False feature rich：XP坚持处理最高优先级的任务

Staff turnover：其它过程是管理者驱动的，XP是程序员驱动的，XP要求程序员接受评估和完成他们工作的职责，XP鼓励交流

**XP的心跳 the development episode：**This is where a programmer implements an engineering task and integrates it with the rest of the system.

1 Top task card 2 get a pair programming partners 3 讨论task 4讨论测试用例 5 写测试用例 6 测试fails 7 写代码 8 跑测试用例 9 迭代测试用例和代码 10 如果需要，重构 11 继承，包括测试

注意内容：结队编程，测试驱动开发，开发后立刻集成，Pairs add value to the analysis, design, implementation, and testing of the system.

**四个变量：COST TIME QUALITY SCOPE,无法同时兼顾这四点**

变量间的交互作用：Cost：更多钱可能有用，但太多钱带来的问题比解决的更多；Time：更多的时间可以提高质量增加范围，但迭代太长会损害反馈；Quality：牺牲质量可以获得短期利益，但会有很大的代价；Scope：小的范围会使质量更好（只要客户需求被满足），更便宜/迅速

客户不会准确的告诉我们他们想要的，需求从来不会一开始就明确。需要通过迭代，让用户看到产品。

Scope管理：1如果我们基于模型创建开发纪律，会解决质量，代价和范围的问题2如果马上要发布，总有内容可以移到下一次发布3不要尝试做太多事，可以保证按时保证质量完成

如何避免遗漏重要业务：1不断的去估计和修正2首先实现最重要的需求

**变更的代价**

某些情况下，指数级增长的变更代价可以被避免

变更代价的假设：我们一直假设变更的风险随时间指数上升，修复的代价也是指数上升。

但随着新技术的使用，有时候变更的代价在降低，如果变更的代价随时间变化不大的话，你的行为与变更代价增长剧烈的情况会有很大差异。

如果变更代价小：1 对于重要的决策，越晚做出越好，使决策正确的可能性更大2 你将只实现必须实现的内容，对未来的提前设计可能不会有用 3 只有元素可以简化已有代码或者使之后的编码更简单才会引入新的设计元素

保持低的变更代价：1 技术方便，对象（OO？）是核心技术2简单的设计，没有其他额外元素，不要对未来设计 3 自动化测试，如果偶然的改变了系统行为会收到提示 4 多次在实践中尝试变更.

**From Nothing, to Monumental, to Agile**:大部分软件开发是一个无秩序的活动，通常以code and fix为特征。当系统规模较小时表现良好；随着系统规模扩大，往系统里添加新功能变得原来越困难；bug变得越来越多并且更难修复；这类系统的典型标志是完成功能开发后长周期的测试。

**方法论**：方法论通过在软件开发中约束过程来达到提高可预测性和更有效率的目的；They do this by developing a detailed process with a strong emphasis on planning inspired by other engineering disciplines - engineering methodologies；They've not been noticeable for being terribly successful；They are even less noted for being popular.；The most frequent criticism of these methodologies is that they are bureaucratic. There's so much stuff to do to follow the methodology that the whole pace of development slows down.

**敏捷方法学**：一种反应；这些新方法试图在没有过程管理和过度过程管理中取得折中来获得合理回报；不同点在于更面向代码而不是文档；在更深层次上意味着敏捷方法是适应性的而不是预测性的，面向人员而不是面向过程

**伪装**：一个重大危险是伪装成你可以遵循一个可预测的过程而实际上你不可以；People who work on methodology are not very good at identifying boundary conditions

通过迭代控制不可预测的过程：清楚的知道我们在哪里；一个频繁反馈的机制；迭代开发

**迭代开发**：There is nothing like a tested, integrated system for bringing a forceful dose of reality into any project；Sit in front of a system and work with it；XP建议一次迭代在一到三周；SCRUM 建议一个月；Crystal will stretch further.

**度量的困难**：决策者和执行者的不一致，需要度量执行者的效率；在科学管理中，有强大的推动力去开发客观的方法来衡量人们的产出；尽管付出极大努力我们依然不能度量软件中最简单的事，比如生产率；没有对这些事情的衡量，任何形式的外部控制都是注定无效的；Introducing measured management without good measures leads to its own problems；必须在基于度量的管理和委托管理之间选择；基于度量的管理适用于简单重复工作，低知识含量以及可度量成果的工作，恰恰与软件开发相悖。

**敏捷四个价值**：交流、简化、反馈、勇气

**交流与简化**：交流越多，对需要完成什么和不需要完成什么会认识的更清晰；系统越简单，需要交流的越少，。

**反馈**：对系统现状的具体反馈是无价的。

**基本原则**：learn to drive, 四个价值，快速反馈，简单假设、渐进改变、拥抱变化、质量工作

实践相互支持，其中一个的缺点被其他的优点所遮盖。

一些敏捷措施的问题和这些措施是如何克服自己的缺点的

**规划游戏：1**虽然你不可能凭一个粗略的计划开始进行开发。2你不能不断地更新计划——这将耗费太长时间，令客户不安。但是：1根据程序员提供的计划，客户自己更新计划。2你一开始就有足够的计划，让客户大概了解未来几年可能发生的情况。3你使用简短发布，所以计划中的任何错误最多有几个星期或几个月的影响。4您的客户正在与团队在一起，所以他们可以快速发现潜在的变化和改进的机会。

**简短发布：**尽管你不可能几个月后投入生产但是1规划游戏帮助您处理最有价值的故事，因此即使是小型系统也具有商业价值。2您正在不断地整合，所以包装版本的成本是最小的。3您的测试足够减少缺陷率，因此您不必经过漫长的测试周期。4 你可以做一个简单的设计，足够支撑当前这个版本，而不是未来所有的。

**隐喻：**尽管你不能仅靠隐喻开发，这里没有足够的细节，并且可能有问题 但是1关于隐喻在实践中是否有效，你可以很快从实际的代码和测试中得出具体的反馈。2你的客户可以很舒服地以隐喻的形式描述系统。3你的重构不断地改善你对于隐喻在实践中的意义的理解。

**简单设计：**1 你习惯了重构，所以改变并不是一个烦恼。2你有一个清晰的整体隐喻，所以你确定未来的设计变化将趋向于遵循一个收敛的路径。3你正在和合作伙伴进行编程，所以你有信心做一个简单的设计，而不是一个愚蠢的设计。

**测试**：你不可能写出所有的测试，这需要花费太多的时间，程序员不会写测试。

1.设计很简单，所以写测试不是那么难。2.结对编程伙伴的压力。3.看到所有的测试通过，你感觉良好，你的客户对系统也感觉良好。

**重构：**不可能一直重构系统，这将花费时间，很难控制，很可能会破坏系统。

1.你习惯于集体所有制，所以你不介意在需要的地方进行更改。2.你有编码标准，所以你不必在重构之前重新格式化。3.你结对编程，所以你更有可能有勇气去应对一个艰难的重构，并且不太可能破坏。4.你有一个简单的设计，所以重构更容易。5.你有测试，所以你不太可能在不知道的情况下打破某些事情。6.你有持续集成，如果一定程度上造成破坏，或者影响了他人的模块，你讲很快收到报告7 你精力充沛，所以有勇气并且更难犯错

**结对编程：1.**编码标准减少了无聊的争吵。2.每个人都经历充沛会减少无用的回话3.结队共同编写测试，给了他们印证他们每个人的理解的机会.4.The pairs have the metaphor to ground their decisions about naming and basic design.5.结队做了简单的决策，他们将共同理解将发生什么。

**持续集成：**1.通过快速运行测试用例你可以知道你是否犯错2.结队编程，会减少需要集成的次数3.通过重构可以减少冲突的概率

!办公室环境——人件

**家具警察：**1.保持所有办公环境的一致性；2统一的塑料地下室；没有窗户（不好分配，统一）有窗户的空间感觉更好-更高的工作质量3.公司传呼系统，打断脑力工作者思考-flow，idea work time4.家具警察用设计监狱的方法来设计工作空间：以最小的成本来最大化利用空间；。只要员工们还拥挤在充满噪音，缺乏办公设施，破坏性的工作场所，除了改进工作场所之外没有别的东西更值得改进了。

更好的工作环境有助于人们工作的更好，或那些工作的更好的人倾向于加入那些提供更好工作空间的公司。

在空间上省钱后果：1减少空间，缺少私密性，充满噪音：省钱；2带来工作人员效率的降低；3对人员的投资大约是工作空间成本的20倍；4在良好工作空间工作的人工作的缺陷率低；5躲避，图书馆，咖啡厅；

**工作区域：**如果您没有合理的工作场所，您的项目将不会成功。对于一个团队来说，好的工作空间和糟糕的之间的区别是立竿见影的。最好的时间是开放空间，并为公共空间保留一点最好的空间。

**敏捷与精益：**精益和敏捷是两组具有高度兼容性的价值观和原则，都阐述了如何成功地进行产品开发。Scrum、XP和看板则是将这些原则运用到实践中的具体方法，换句话说，它们是精益和敏捷软件开发里轻度重叠的三种不同风格。

**敏捷宣言：1**个体和互动胜过流程和工具2可以工作的软件胜过详尽的文档3 客户合作胜过合同谈判4响应变化胜过遵循计划（虽然右项也具有其价值，但我们认为左项具有更大的价值）

**敏捷原则：1**我们最重要的目标，是通过持续不断地及早交付有价值的软件使客户满意。2欣然面对需求变化，即使在开发后期也一样。为了客户的竞争优势，要通过敏捷过程掌控变化。3经常地交付可工作的软件，比如相隔几星期或一两个月就交付，倾向于采取较短的周期。4业务人员和开发人员必须相互合作，项目中的每一天都不例外。5激发个体的斗志，以他们为核心搭建项目，提供所需的环境和支援，辅以信任，从而达成目标。6不论团队内外，传递信息效果最好、效率也最高的方式是面对面的交谈。7可工作的软件是进度的首要度量标准。8 敏捷过程倡导可持续开发。责任人、开发人员和用户要能够共同维持其步调稳定延续。9坚持不懈地追求技术卓越和良好设计，敏捷能力由此增强。10以简洁为本，它是极力减少不必要工作量的艺术。11最好的架构、需求和设计出自自组织团队。12团队定期地反思如何能提高成效，并依此调整自身的举止表现。

**精益原则：**

**全局优化**：局部的优化长期来说，会对系统整体优化不利。措施1专注于整体价值流：从概念到现金。从客户需求到软件部署。2交付完整产品：客户不要软件产品，他们要解决问题。完整的解决方案是由完整的团队构建的。3 着眼长期：警惕导致短期思维和优化局部业绩的治理和激励体系。

**消灭浪费**：浪费指所有那些不能增加客户价值的事项。软件开发中的三大浪费如下。1构架错误的功能：“没有什么比高效完成根本不应做的工作更无用。”2 拒绝学习：我们有很多策略都干扰了我们学习，例如，只按计划行事、频繁移交、决策与工作分离等，而学习则是开发的精髓。3辗转现象：那些干扰价值顺利流动的做法，例如，任务切换、请求清单冗长、大堆未完成的工作等等，都只能达到事倍功半的效果。

**品质为先**：如果在验证过程中总是能发现缺陷，那流程就有问题。1最终验证不应发现缺陷：所有软件开发流程的根本目的都是尽早在开发阶段发现并修复缺陷。2 采用测试先行的开发模式让流程具有防误机制：测试（包括单元测试、端对端测试和集成测试）必不可少，以此建立信心，保证系统在任何层次、在开发阶段任何时间点都始终正确无误。3打破依赖：系统架构应当支持随时添加功能。

**不断学习：**规划工作非常有用。学习则必不可少。1可预测的性能来自于反馈：可预测的组织不会猜测未 来并称之为计划;反之，他们会培养能力对未来做出 响应。2保持选择方案：初代码为实验——使其具有容变性。3最后可靠时刻：在做出不可逆转的决策之前尽可能学习。不要提前做出纠正代价高昂的决策，也不要事后才做出决定。

**尽快交付：**从一开始就深入了解所有干系人看重的价值。然后基于这样深入了解的价值观，创建稳定、连贯的工作流。1快速交付、高质量和低成本是完全相互兼容的：以速度竞争见长的公司拥有很高的成本优势，他们可以交付优质的产品，而且对客户需求更为敏感。2排队理论同样适用于开发，而不仅仅是服务行业：专注于使用性会造成交通堵塞，反而降低了使用性。以较小的批量、限制同时进行的工作数来缩短周期时间，大力限制等待清单和队列的长度。3 管理工作流比管理进度表要容易得多：建立可靠、可预测交付物的最佳方式是通过迭代和看板建立可靠、可重复的工作流。

**人人参与：**聪明、有创造力的人员的时间与精力，是当代经济的稀有资源和竞争优势的基础。获得公正薪资的人员在自主性、成长性和使命感等方面受到激励。

**不断提高：**结果不是重点——重点是培养人、发展体制，使之能够交付结果。1 失败是个学习机会：即使是非常小的失败都会被深入调查并纠正的，做到一丝不苟的时候，才可能获得最可靠的性能。2标准存在的目的就是要被质疑和提高的：将现行的、最知名的做法纳入人人都遵循的标准，与此同时鼓励所有人质疑并改变标准。3使用科学方法：教团队建立假设、开展大量快速实验、创建简明文档并实施最佳方案。

**看板：**看板是敏捷软件开发的精益方法。

看板规则：1可视化工作流：把产品切分成小块，将每一块写在一张卡片上，然后将卡片贴到墙上；墙上的每一栏都有名称，以此显示每张卡片在工作流中所处的位置。2限定在制品（WIP）：针对工作流的每个状态，明确限定正在运行中的工作项数量。3衡量并管理周期时间：完成一个工作项的平均时间，有时称为前置时间（流通时间）。优化流程，让周期时间尽可能短、尽可能可预测。

**Scrum，XP，看板对比：**Scrum专注于结构和沟通，XP增加了工程实践，看板则专注于将工作流可视化，并对瓶颈进行管理。

**拆分业务和技术责任**我们的策略的关键之一是让技术人员专注于技术问题，商业人士专注于业务问题。 该项目必须由业务决策驱动，但业务决策必须通过关于成本和风险的技术决策来形成。业务人选需要选择的：1业务范围，发布时间2产品功能的优先级3功能能范围。开发人员需要做的：1开发时间估算2估算可选技术方案的影响3选择适合他们的开发过程4他们将要开始的编程实践，以及评估编程结果、控制变更的方式

**技术的选择：**1不仅仅是技术选择，也是商业决策。2如果客户有要求，开发人员需解释相关后果，之后业务人员指定业务决策3如果引入了一个技术，在放弃之前必须由专人负责这个技术

**计划策略**：从概要计划到具体计划，时间范围从大到小。尽量快速低成本的制定计划，从而变更的代价降低。

**计划的目的**：1凝聚团队 2确定范围、优先级3估算日程、成本4给每开发人员信心，系统可以完成5提供反馈的基准

**影响计划的原则：1**只制定下一阶段的计划2接受责任，责任只能被接受，不能分配。3负责人进行估计4忽略模块间依赖5 Planning for priorities vs. planning for development

规划游戏的目的：最大化项目团队的软件产出价值。对于软件的价值，你要减去开发成本和开发时的风险

**拆分行动**：勘探阶段-找出系统能做的新东西；承诺阶段-决定下一阶段发布的产品需要满足的需求；转向阶段- Guide development as reality molds the plan

**勘探阶段**:给开发和业务人员就系统要做什么达成共识。1、写一个故事—业务人员描述系统业务场景;2、评估故事—开发人员评估业务场景，若无法整体评估，则拆分评估。任务有：1、将故事分解成任务2、分解合并任务

**承诺阶段**：目的在于为业务人员选择下次发布的功能范围和时间，为开发人员提供完成信心。1、价值排序，将业务分成三部分：a、如果缺失系统将无法工作b、缺失的话商业价值降低c、如果有这些功能会更好2、风险排序，a、可精确估计b、可合理估计c、完全无法估计；3、确定开发速率4、选择范围 **任务**：接受任务-开发人员接受任务职责；评估任务：评估理想状态下每个任务的开发时间；设置工作饱和度-可工作时间占总时间比率；平衡

**转向阶段**:目的在于基于开发和业务所了解的信息来更新计划。1、迭代2、恢复3、新故事4重新评估。任务：实现任务；记录过程； 恢复-如果过度使用；通过功能测试来验证故事

**持续集成:** 最基础，最有价值的实践之一，通过自动构建和测试允许一个团队在一天之内多次构建他们的项目。每日构建是最低要求。**1、**代码必须在提交后一段时间内被集成2**、**如果集成需要花费几个小时，它可能不太试用持续集成3有工具支持快速集成/构建/测试周期很重要4还需要合理完成可在几分钟内运行完成的测试用例集5能够接受持续集成的开销的一个重要原因是它大大降低了项目的风险。

**集体所有制：**没有测试项目会很容易失败；复杂的代码不会存在很久；提高个人在项目中的成就感；倾向于在团队中传播关于系统的知识。

**结对编程：**是两个共同编程（分析、设计、测试）的人之间的对话，在如何好的编程上相互理解；并不是一个人教另一个人；

**为什么结对编程会有效：**第一价值是沟通，有几种沟通的形式比面对面更加紧密；比单独完成更高效；最终代码质量更高；有些实践只能依靠结对编程；有利于重构、测试、持续集成

**设计策略：**做最可能做的最简单的事情（DTSTTCPW）

四个价值观：1交流，复杂的设计比简单的设计更难沟通2简单，我们应该有一个设计策略来制作简单的设计，且策略本身表述应该很简单。3反馈 - 简单的设计通过快速完成来反馈设计上出现的问题4勇气 - 最有勇气的事是在设计刚开始进行时果断停止；为了遵循这些价值观，我们必须做到：1遵循简单的策略来产生简单的设计2找到一个快速验证质量的方法3将我们学到的东西反馈进设计过程中4将整个过程的循环时间尽可能缩短

**预先设计**：如果你清楚的知道将会发生什么，并且清楚的知道如何去解决它，那么一般来说你可以现在就对以后可能的需求进行设计在任何一种情况下，你都必须选择要么承担超额设计的开销，要么承担不带来当前利益的更复杂设计的持续成本

**更好的设计策略:**1.以测试开始，因为我们会知道我们什么时候会完成（通过测试）。我们需要一定的设计来编写测试：他们的对象和可见方法是什么2.设计和实现必须足以使测试运行。你实现必须足以使当前以及之前所有的测试运行3.重复设计（repeat）4.如果你发觉到任何一个使设计更简单的机会，就去做

**最佳设计**的定义是能运行所有测试用例的最简单的设计；最简单的四个具有优先顺序的限制条件1.整个系统（包括代码和测试）必须能够传达你想要沟通的一切2.系统不能包含重复代码（1和2在一起构成了“一次且只有一次”规则）3.系统应该有尽可能少的类4.系统应该有尽可能少的方法

**图画（picture）在设计中的作用**：1首先，使用显式的图画而不是纯粹的讨论或文本模型来设计软件没有任何问题2使用图画设计的另外一个优势在于速度3使用图画的问题是，他们无法给你具体的反馈（风险）4小的初期投资 - 意味着我们一次画不多的几张图片5制胜计划——意味着我们不应该出于恐惧而使用图片（比如，我们因为不知道应该怎样设计而敷衍）

快速的反馈——意味着我们能够快速发现我们的图片是否符合目标6与直觉相处——意味着我们鼓励那些使用图片工作效果最好的人的图片7拥抱变化与轻装旅行——意味着我们不会保存哪些已经产生影响的图片，因为它们代表的决定很有可能会更改8XP策略是任何人都可以使用任何他们想要的图片设计，但一旦出现可以用代码回答的问题，设计人员必须转而使用代码解答9如果你使用文字性编程语言，建议绘制图片的时间不要超过10到15分钟

**体系结构的一部分可以用系统隐喻所描述。 如果你有一个很好的比喻，团队中的每个人都可以了解整个系统如何运作**。

第一次迭代的结果必须是整个系统的功能骨架。

您可以将整个架构投机地放置在合适的位置上，或者你可以将你所需要的所有架构放置在现在来满足当前的需求，并相信你可以在以后投入更多。

我们将在我们编写代码之前，逐分钟编写测试。 我们将永远保存这些测试，并经常运行它们。 我们还将从客户的角度进行测试

**测试的两个原则：**隔离和自动化：1首先，每个测试都不会与您写的其他测试交互2当人们工作太多时，判断可能失败， 所以测试必须自动的

测试取得成功的一种方式就是测试取得了你意想不到的结果

**程序员的测试：**1如果方法的接口不清楚，则在编写方法之前先写一个测试2如果接口清晰，且你觉得实现不复杂，你在编写该方法之前先写一个测试3如果你想到一个代码运行的异常情况，你可以编写测试来展示异常4如果以后发现问题，写一个隔离问题的测试5如果你要重构一些代码，并且不确定应该如何表现，首先要写一个测试

**顾客的测试—**根据故事编写测试：他们需要问自己的问题是：“在我确信这个故事已经完成之前，需要检查什么？” 他们提出的每个场景都变成一个测试，这是一个功能测试。客户通常无法自己编写功能测试，这就是为什么任何大小的XP团队至少有一个专用测试人员

**其他的测试：1**并行测试 - 旨在证明新系统与旧系统完全一致的测试。2压力测试 - 测试设计，以模拟最差的可能负载。 压力测试对于性能特征不容易预测的复杂系统是有效的3猴子测试 - 一个测试，旨在确保系统在无意义的输入面前合理反应

**XP中的设计？**XP似乎宣告了程序设计的死亡。像UML，灵活的框架，甚至连模式这些设计技巧都被忽视。**实际上，**XP中包括很多的设计，只是不同于以往软件开发流程中的做法。XP通过允许进化的实践技巧使演进式设计成为一种可行的设计策略。它还设计人员去学习如何使设计简单，如何利用重构保持设计的整洁，如何在一个演进的形式下使用模式。

**演进式设计：**演进式设计的本质是系统的设计随着软件开发的过程增长。随着程序代码的发展，设计也跟着调整。 在常见的使用中，演进式设计实在是彻底的失败。设计的结果是一堆为了某些特殊条件而巧妙安排的决定所组成，每个条件都会让程序代码更难修改。设计要能够让你可以长期很简单地修改软件。当设计不如预期时，你应该能够做有效的更改。

**Planned Design：**在设计图中确定所有的细节，使用经验或数学分析制定出如何设计结构体的法则。当设计图完成，就可以交给另一个施工的公司按图施工。Designer 先定出重要的部份，他们不负责撰写，只负责设计。所以 designer 可以利用像 UML 这样的技术，不需要太注重撰写程序代码的细节问题，而在一个比较属于抽象的层次上工作。一旦设计的部份完成了，他们就可以将它交给另一个团队 去 "建造"。因为 designer 朝着大方向思考，所以他们能够避免因为策略方面不断的更改而导致软件的失序。Programmer 就可以依循设计好的方向 (如果有遵循设计) 写出好的系统。**缺点**:1是当你在进行设计时，你不可能同时把所有必须处理的问题都想清楚。所以将无可避免的遇到一些 让人对原先设计产生质疑的问题。如果 designer 在完成工作之后就转移到其它项目，Programmer 开始迁就设计来写程序，于是软件开始趋于混乱。就算找到 designer，花时间整理设计，变更设计图，然后修改程序代码。但是必须面临更短的时程以及更大的压力来修改问题，又是混乱的开端。 2软件开发文化方面的问题。 Designer 因为专精的技术和丰富的经验而成为一位 designer，而没有时间写程序代码。但是，开发软件的工具发展迅速，当你不再撰写程序代码时，你不只是错失了技术潮流所发生的改变，同时也失去了对于那些实际撰写程序代码的人的尊敬。

**变更需求** 处理变更需求的方式之一是做有弹性的设 计，于是当需求有所更改，你就可以轻易的变更设计。然而，这是需要先见之明去猜测将来你可能会做怎样的变更，这是非常困难的。一项预留处理易变性质的设计可能对于将来的需求变更有所帮助，但是对于意外的变化却没有帮助 (甚至有害)。部份需求问题归咎于对需求的了解不够清楚，所以有人专注于研究需求处理，希望得到适切的需求以避免后来对设计的修改。但是一样无法对症下药。很多变更起因于瞬息万变的商业环境，是无法预料的。

XP因为许多原因而备受争议，其中之一就是它主张演进式设计而不是 planned design。

XP的基本假设是它可以将修正错误的指数曲线拉平。XP使曲线更平缓并能运用这种优势。这是因为 XP 实作技巧之间的耦合效果，这些有效的实作技巧包括：1如果没有 testing 提供保障，其它的 XP 实作技巧都不可行。2Continuous Integration 可以保持团队成员信息同步，所以当你有改变的部份，不必担心与其它成员资料整合会有问题。 3Refactoring 具有类似的成效，这么做要比无章法或是特殊方式的 restructuring 明显的更有效率

**XP 大声疾呼的两个口号是** “Do The Simplest Thing that Could Possibly Work”和 “You Aren‘t Going to Need It”。两项都是XP实务中简单设计的表现形式。意思是现在不要为了将来可能用到的功能加入任何程序代码。问题则出在像 framework、重用组件、和弹性化设计，这些东西本来就很复杂。你事先付出额外的成本去打造它们，希望稍后将这些花费都赚回来。这个事先弹性设计的 想法被认为是软件设计有效率的关键部份。

**XP的建议**:在处理第一个问题时不要因为可能需要某项功能，就建造出弹性的组件组及框架出来。让整体结构随着需要成长。 1效益。这种经济效益上的限制是因为我们有可能出错。 2复杂的设计比简单的设计还要令人难懂。所以随着渐增的系统复杂度，更加难以对系统做任何修改。如此，若系统必须加入更复杂的设计时，成本势必增加。

除非到了往后的阶段有所需要，否则你不会浪费精神去增加新的功能。即使不会多花成本，你也不会这样做，因为就算现在加入这些功能并不增加成本，但是却会增加将来做修改时的成本。

**简单系统订了四个评量标准**，依序是1通过所有测试。2呈现所有的意图3避免重复4最少数量的类别或方法:"呈现所有的意图" 这句话的本意就是简单明了的程序代码。 “巧妙的程序代码” 这个字眼经常被滥用，它是指意图清楚的程序代码。

XP 的 “Once and Only Once” 以及DRY(Don‘t Repeat Yourself) 都专注在去除重复的程序代码。这些良好的建议都有很显著而且惊人的效果。只要依照这个方式，项目就可以一路顺利的运作。但是它也不能解决所有问题，简单化 还是不容易达成。

“重构过度设计的系统要比重构没有设计的要来的容易多了” 做一个简单一点的设计是最好的，但是，稍微复杂一点点也不是什么严重的事情。

不要太在意什么是最简单的设计。毕竟后来你可以，应该，也会再重构。愿意在最后重构，比知道如何做简单的设计重要得多。

Does Refactoring Violate YAGNI?

基本上这个问题起因于重构需要耗费时间却没有增加新的功能。而 YAGNI 的观点是假设你为了眼前的需要做设计而不是未来，这样算是互相抵触吗？

YAGNI 的观点是不要增加一些现阶段不需要的复杂功能，这也是简单设计这项技巧的部份精神。重构也是为了满足尽可能保持系统的简单性这个需要，所以当你觉得可以让系统变得更简单的时候，就进行重构。

简单设计不但利用了 XP 的实务技巧，本身也是其中一项有用的实务技巧。

唯有伴随着测试，持续整合，及重构的运用，才能有效地做出简单设计。同时，让研发异动曲线保持平缓的基础也 就是保持设计的简单。任何不必要的复杂都会让系统变得难于调整，除非这个复杂性是你为了所预测的弹性而加入的。不过，人们的预测通常都不太准确，所以最好 还是努力地保持简单性。

不管怎样，人们不太可能第一次就能够获得最简单的东西，因此你需要重构来帮助你更接近这个目标。

**Patterns and XP:**很多人都认为 patterns 似乎与 XP 是有冲突的。 争论的本质在于 patterns 常被过度滥用。有效的设计告诉我们 pattern 值得付出代价去学习。我们需要更熟悉于如何逐步地运用 patterns。就这一点而言，XP 只是与一般使用 patterns 的方式不同而已，并没有抹煞它的价值。 XP 也许是开发的一种流程，但 patterns 可是设计知识的骨干，不管是哪种流程这些知识都是很有用的。不同的流程使用 patterns 的方式也就不同，XP 强调等到需要时才使用 patterns 以及透过简单的实作逐步导入 patterns。所以 patterns 仍然是一种关键知识。

**采用 XP 的人使用 patterns 的建议 :1**花点时间学习 patterns。2留意使用 patterns 的时机 (但是别太早)。3留意如何先以最简单的方式使用 patterns，然后再慢慢增加复杂度。3如果用了一种 pattern 却觉得没有多大帮助－不用怕，再次把它去掉。

**Growing an Architecture** 架构这个字眼代表系统核心组件的概念，也就是难以改变的部份，剩下的都必须建造在这基础上。XP 的批评者再一次地声称 XP 忽视架构，因为 XP 使用的方法是尽快地写程序，然后相信重构会解决所有设计的问题.我认为一个概括性的初始架构有它的用处在。像是一开始要怎么将应用分层，如何与数据库互动等,关键性的差异是在于这些初期架构的决定是可以更改的，只要团队认为他们早期的判断有误时，就应该要有勇气去修正.

**所以我建议**从评估架构可能的样子开始。 假如你看到将会有多个使用者使用到大量的资料，一开始就直接使用数据库。若你看到很复杂的商业逻辑，就套用 domain model。所以你要有所准备，在发现所使用的结构没有帮助时尽快简化你的结构

**UML and XP :两者**有些不兼容。XP 显然不重视 diagram。虽然台面上大家对 “好用就用” 有共识，但是实际上却是 “实际上采用 XP 的人不画蓝图”。

这种情形来自两个因素，其一是有人觉得软件蓝图有用，而有人不觉得有用。

另一种情形是软件蓝图常引人进入繁重的流程中，这些流程耗时费力却不见得有用，甚至还会产生坏处。我认为应该教导人们如何使用蓝图却不落入这样的陷阱，

**建议 1** 首先别忘了你画这些图的目的，主要的价值在于沟通。2有效的沟通意味着选择重要的部份而忽略不重要的部份。这样的选择也是有效运用 UML 的关键。3不必把全部的 class 都画出来，画出重要的就好。

有一个使用蓝图的通病就是人们通常希望详细完整的把图表现出来。其实程序代码就是提供完整信息的最佳来源，同时程序代码本身也是保持信息同步最简单的方式。因为图形的巨细靡遗就是一目了然的敌人。

**建议**:1保持简短。2不要做得太详细(只挑重要的做)。3把结果当作是草图，而不是定案。

XP中不提倡架构师概念。

XP中的教练。在 XP 技术的领导特质是透过教导程序员和帮助他们做决定而呈现。

Things that are difficult to refactor in

我们能用 refactoring 来处理所有设计方面的决定吗？

或者，有些问题太普遍而且难以在将来加 入设计中？此时，XP 的正统做法是所有需求都可以轻易的在需要的时候增加，所以 YAGNI 总是能够适用。我猜是不是有例外？

我的想法是，如果你知道怎么做，你 就要考虑现在做和将来做，两种情形之间不同的成本。

反过来说，如果你没有处理过那样的问题，不仅是你无法正确判断需要的成本，你也比较不可能把事情作好。 这种情形，你就要选择将来再做。如果你还是执意做了，而且尝到苦果，可能会比不做的情况更糟。

stories 的顺序：

XP 偏向于只让商业价值这一个因素影响 stories 的顺序。

我相信在商业价值和技术风险之间能找到平衡点。

So is Design Dead?

没什么原因，只是设计的本质已经改变。

XP 的设计追求以下的技巧：

持续保持清爽的程序代码，越简单越好。

重构的技巧，所以当你觉得必要的时候都可以有信心的动手。

具有 patterns 的知识：不只是照它的解法，更要感觉何时可以应用，或是如何导入 patterns。