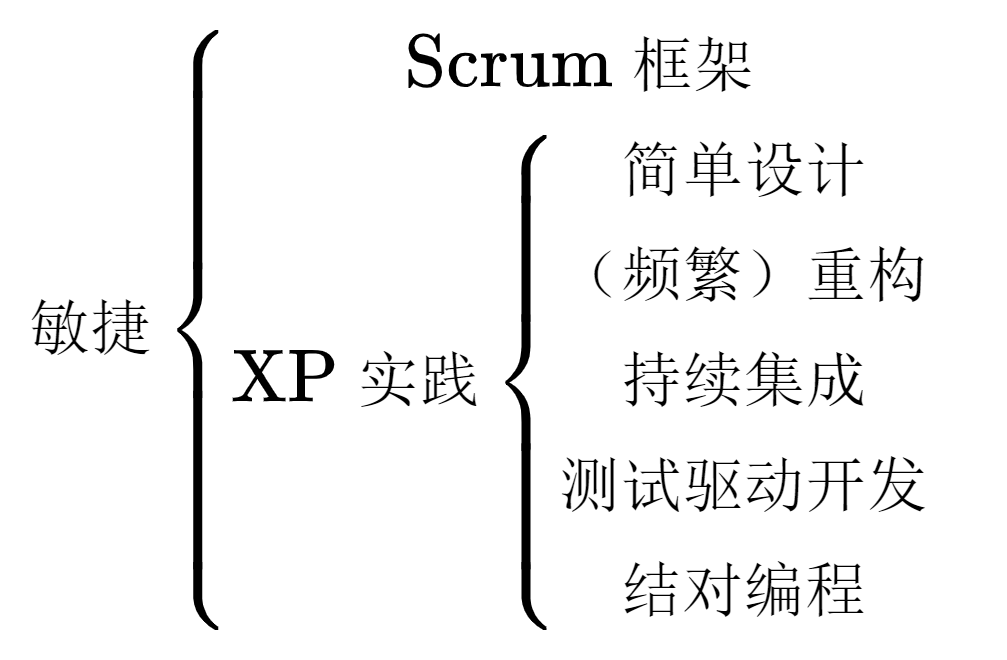
**声明：本文档不适合预习。请在基本理解各种软件过程和方法的内涵之后再阅读本文档。**

**软件工程定义：**(1)将系统化、规范化、可量化的方法应用于软件的开发、运行和维护，即将工程学应用于软件的过程。(2)对(1)中的方法的研究。

# 《没有银弹》

(1)没有任何一种单纯的技术或管理上的进展，能够独立地承诺十年内使生产率、可靠性或简洁性获得数量级上的进步(2)所有大家看到的技术、管理方法都不会给软件开发带来意想不到的效果。(3)软件开发在根本上就是困难的。

**软件工程中的根本和次要任务:**(1)根本任务—打造由抽象软件实体构成的复杂概念结构。(2)次要任务—使用编程语言表达这些抽象实体，在空间和时间限制内将它们映射成机器语言。(3)除非次要任务占了所有工作的9/10，否则即使全部次要任务的时间缩减到零，也不会给生产率带来数量级上的提高。

**为什么软件的发展比硬件慢？**(1)不是软件发展慢，而是硬件发展太快。(2)“从人类文明开始，没有任何其他产业技术的性价比，能在30年之内取得6个数量级的提高，也没有任何一个产业可以在性能提高或者降低成本方面取得如此的进步。这些进步来自计算机制造产业的转变，从装配工业转变成流水线工业。

**摩尔定律:**是由戈登·摩尔于1965年提出的观察和预测，描述了计算机硬件中集成电路的演变。指出每隔18至24个月，集成电路上的晶体管数量大约会翻倍，同时成本降低。

**根本困难-软件特性中固有的困难：**(1)我认为软件开发中困难的部分是规格化、设计和测试这些概念上的结构，而不是对概念进行表达和对实现逼真程度进行验证。(2)如果这是事实，那么软件开发总是非常困难的。天生就没有银弹。(3)现代软件系统中无法规避的内在特性：复杂度、一致性、可变性和不可见性。

**复杂度：**(1)软件实体可能比任何由人类创造的其他实体都要复杂，因为没有任何两个软件部分是相同的，如果有我们会将它们合并。(2)数字计算机本身就比人类建造的大多数东西复杂。计算机拥有大量的状态，这使得构思、描述和测试都非常困难。软件系统的状态又比计算机系统状态多若干个数量级。(3)软件实体的扩展也不仅仅是相同元素重复添加，而必须是不同元素实体的添加。整个软件的复杂度以很大的非线性级数增长。(4)软件的复杂度是必要属性，不是次要因素。抽掉复杂度的软件实体描述常常也去掉了一些本质属性。(5)复杂度问题造成软件产品开发问题-团队成员之间的沟通非常困难，导致了产品瑕疵、成本超支和进度延迟-由于复杂度，列举和理解所有可能的状态十分困难，影响了产品的可靠性(6)由于函数的复杂度，函数调用变得困难，导致程序难以使用-由于结构性复杂度，程序难以在不产生副作用的情况下用新函数扩充-由于结构性复杂度，造成很多安全机制状态上的不可见性(7)复杂度引发管理上的问题-全面理解问题变得困难，从而妨碍了概念上的完整性-它使所有离散出口难以寻找和控制-它引起了大量学习和理解上的负担，使开发慢慢演变成了一场灾难.

**一致性(配合性)：**(1)物理学家面对异常复杂的事物，他们坚信必定存在着某种通用原理。物理学是研究上帝创造的东西。(2)软件开发面对的复杂度往往是随心所欲、毫无规则可言的，来自若干必须遵循的人为惯例和系统。软件开发面对的是人，不是上帝。(3)很多复杂性来自保持与其他接口的一致。

**一致性导致的复杂度**:(1)不同于硬件或物理工程，软件必须和许多不同的系统进行交互，从硬件到网络、从数据库到用户界面，每一部分都需要密切配合，这使得每一次更新或改动都会引发系统其他部分的变动。(2)想象软件开发就像是一个多层次的复杂齿轮系统，所有齿轮彼此紧密咬合。每个齿轮代表一个软件模块，它们必须无缝协作才能让整个系统顺利运行。问题在于，修改任何一个齿轮都会影响其他齿轮的运行，必须调整多个其他齿轮才能保持系统的正常运转。(3)当你修改系统中的某个部分时，配合性的问题要求你不仅要调整这一部分，还必须确保所有相连的模块也与之协调工作。这导致了小小的修改可能引发连锁反应，使得整个系统的复杂度迅速上升。

**可变性：**软件实体经常会遭受到持续的变更压力。-汽车、建筑可以修改，但很少有人修改，大家都知道成本很高。-软件包含了很多功能。-软件可以很容易地进行修改——它是纯粹思维活动的产物，可以无限扩展。

**不可见性**：软件是不可见的和无法可视化的-抽象的功能：几何抽象、机械制图、化学分子模型-帮助我们捕获物理存在的几何特性。•软件的客观存在不具有空间的形体特征-现在没有任何一种2维、3维的图形可以描述软件。•这限制了个人的设计过程，也严重的阻碍了相互之间的交流。•UML一定程度上可以展示软件结构，但并不能把软件变得可见

# 《大教堂和市集》

第一次以大教堂模式和开放市集模式的比喻形象生动地将自由软件和商业封闭软件区分开来——“一种是封闭的、垂直的、集中式的开发模式，反映一种由权利关系所预先控制的极权制度；而另一种则是并行的、点对点的、动态的开发模式。”他在文中论证了自由软件不仅仅是一种乌托邦的理想，而是在开发模式上真正代表着“先进生产力”，代表着历史发展趋势的必然。

1998年2月3日，硅谷的一次会议上，“开放源代码（OpenSource）”被正式提出，后来发展成为开源运动。雷蒙德成了领导这场运动的理论家和开放源代码促进会（OpenSourceInitiative）的主要创办人之一。

**开源定义:1.**自由再分发 2.源代码 3.衍生作品 4.作者源代码的完整性 5.不得歧视个人或群体 6.不得歧视努力的领域的研究 7.许可证随程序分发 8. 许可证不得特定于某个产品 9.许可证不得限制其他软件10.许可证必须是技术中立的

**LinusTorvalds**的开发风格：尽早尽多的发布，委托所有可以委托的事，对所有的改动和融合开放。

**Fecthmail**:1.每个好的软件工作都开始于搔到了开发者本人的痒处。-2.好程序员知道该写什么，伟大的程序员知道该重写（和重用）什么。3.“计划好抛弃，无论如何，你会的”4.如果你有正确的态度，有趣的问题会找上你的。5.当你对一个程序失去兴趣时，你最后的责任就是把它传给一个能干的后继者。

拥有用户的重要性:把用户当做协作开发者是快速改进代码和高效调试的无可争辩的方式。

**Linus定律:**如果有一个足够大的beta测试人员和协作开发人员的基础，几乎所有的问题都可以被快速的找出并被一些人纠正。----我认为这是建造教堂和集市模式的核心区别，在建造教堂模式的编程模式看来，错误和编程问题是狡猾的、阴险的、隐藏很深的现象，花费几个月的仔细检查，也不能给你多大确保把它们都挑出来的信心，因此很长的发布周期，和在长期等待之后并没有得到完美的版本发布所引起的失望都是不可避免的。

**Delhpi效应**:社会学家在几年前已经发现一群相同专业的（或相同无知的）观察者的平均观点比在其中随机挑选一个来得更加可靠，他们称此为“Delhpi效应”，Linus所显示的证明在调试一个操作系统时它也适用——Delphi效应甚至可以战胜操作系统内核一级的复杂度。

**市集风格的必要的先决条件**：不能以市集模式从头开发一个软件，我们可以以市集模式测试、调试和改进，但是以市集模式从头开始一个项目将是非常困难的。•当你开始创建社团时，你需要演示的是一个诺言，你的程序不需要工作的很好，它可以很粗糙、很笨拙、不完整和缺少文档、它不能忽略的东西是要吸引大家卷入一个有趣的项目。

**交流与沟通：**在《人月神话》一书中，FredBrooks观察到程序员的工作时间是不可替代的：在一个误了工期的软件项目中增加开发人员只会让它拖得更久，他声称项目的复杂度和通讯开销以开发人员的平方增长，而工作成绩只是以线性增长，这个说法被称为“**Brooks定律**”，被普遍当作真理，但如果**Brooks定律**就是全部，那Linux就不可能成功。Brooks定律（以及随之而来对开发团队规模的恐惧）建立在这样的假设上：项目的沟通结构是一个完全图，即人人之间都沟通。但在开源项目中，外围开发者实际工作在分散而并行的子任务上，他们之间几乎不交流；代码修改和bug报告都会流向核心团队，只有在那个小的核心团队里才会有Brooks开销。

**为什么开源软件没有开创出云计算和移动计算？**开源软件没有直接开创云计算和移动计算，主要原因在于这些技术的兴起依赖于硬件进步和商业模式创新。云计算的崛起得益于虚拟化和分布式计算技术的成熟，这些技术由像亚马逊等大公司率先推动，通过大规模基础设施的商业化来满足企业需求。移动计算则主要依赖于硬件的小型化、电池技术和无线通信的突破，这些方面多由私营企业主导。虽然开源软件在系统层如Linux和Android中扮演了重要角色，但硬件研发、基础设施建设以及市场化的推动主要由商业力量完成。（1.硬件发展和商业模式的驱动2.市场和用户需求的推动3.开源软件的重点在软件本身4.复杂的商业化模式5.企业的整合与创新能力6.云计算和移动计算的产品化)

# 开源软件经济学

**开源软件遵循经济原理：**开放源码并不能免于经济法则。•很多有责任尽量提升股东价值的大型上市公司，投入很多资金支持开放源码软件（通常是负担大型程序团队的开发费用）。而这正可以用互补物的原理来解释。

**软件和硬件**：软件是很容易让硬件普及化的（只要写个小小的硬件抽象层，比像WindowsNT的HAL就只有很小一段程序），不过硬件想让软件普及化却是难上加难。就如StarOffice营销团队所学到的教训，软件是不能互换的。即使价格为零，由微软Office切换过来还是需要成本。除非切换成本变成零，否则桌面办公室软件不可能真的普及化。

# **Characterizing the Software Process**

**理想软件过程：**可预测：成本估算和进度承诺是（结果）具有合理的一致性，且（产品/服务）的质量，最终产品通常能满足用户需求。

**什么是统计控制？**1.我们定义“处于统计控制”如下：如果对过程的测量结果足够接近过程模型预测的结果，则认为过程处于统计控制状态。 2.当一个过程处于统计控制状态时，它表现出稳定的、可预测的行为，这样就可以通过历史数据来估计将来的性能。在控制图上，如果数据点随机地分布在中心线两侧，并且都在上控制限（UCL）和下控制限（LCL）之内，没有任何非随机的模式或趋势，那么可以认为过程是处于统计控制状态的。

**统计控制** 1.软件过程管理的基本原则是，如果开发过程处于统计控制之下，就可以实现一致的更好的结果。2.如果过程没有处于统计控制之下，就不可能持续进步，直到它处于控制之下(来自日本工业的经验) 3. 当一个过程处于统计控制之下时，以大致相同的方式重复工作将产生大致相同的结果。 4.过程决定结果！

**测量**:1.统计控制背后的基本原则是测量。2.正如一个世纪前Lord Kelvin所说：“……当你能够测量你所谈论的内容，并用数字表达它时，你对它有所了解；但当你无法测量它，当你无法用数字表达它时，你的知识是贫乏和不满意的；它可能是知识的开始，但你在思想上几乎没有达到科学阶段……”**几个因素**：1.也许最重要的是，仅仅测量人类过程的行为就会改变它们。由于涉及人们的恐惧和动机，因此必须以不同于自然现象数据的方式来看待结果。2.还必须将测量限制在那些真正会被使用的几个项目上。测量既昂贵又有破坏性。

**开发过程改进：** 1.应对软件问题的第一步是将整个开发任务视为一个可以控制、测量和改进的过程。 2.我们定义一个过程为一系列任务，当正确执行时，会产生期望的结果。

**提高软件能力**： 1. 了解他们开发过程或过程的当前状态. 2. 制定理想过程的愿景. 3. 按优先顺序建立所需的过程改进行动清单. 4. 制定完成这些行动的计划.5. 承诺资源来执行计划。

**过程成熟度级别**1.初始级。在过程处于统计控制之下之前，不可能有秩序的过程改进。2.可重复级。组织通过启动严格的项目管理承诺、成本、进度和变更，实现了具有可重复统计控制水平的稳定过程。3.已定义级：组织已经定义了过程，以确保一致的实施，并为更好地理解过程提供基础。此时，可以有用地引入先进技术。4.已管理级：组织已经开始全面的进程测量，超出了成本和进度表现的范畴。这是最显著的质量改进开始的时候。5.优化级：组织现在有了持续改进和优化过程的基础。

**初始过程：1.**初始过程可以被称为临时的，甚至经常是混乱的。2.组织通常没有正式的程序、成本估算和项目计划。3.工具与过程既不集成也不统一应用。 4. 变更控制不严格，几乎没有高级管理层对问题和问题的了解。5.由于问题经常被推迟甚至被遗忘，软件安装和维护经常存在严重问题。

**尽管初始过程的组织可能已经制定了正式的项目控制程序，但并没有管理机制来确保这些程序被实际使用。1.**最好的测试方法是观察这样的组织在危机中的表现。如果它放弃了已建立的程序，只是回归到简单的编码和测试阶段，那么它可能处于初始的流程阶段。2.如果技术和方法是合适的，它们在危机中必须得到应用；如果不合适，那它们根本不应该被使用。3.组织行为混乱的一个原因是，它们缺乏足够的经验来理解这种行为的后果。4.因为许多有效的软件活动（如设计和代码评审或测试数据分析）看起来并不能直接支持产品的交付

**改进初始流程：**(1)项目管理。1)项目管理系统的基本作用是确保对承诺的有效控制。2)这需要充分准备，明确责任，公开申报。(2)管理监督。1)一个有纪律的软件开发组织必须有高级管理层的监督。这包括在正式承诺之前审查和批准所有主要发展计划。(3)质量保证。1)质量保证小组负责向管理层保证软件开发工作实际上是按照应有的方式完成的。2)为了有效，保证组织必须对高级管理层有独立的汇报关系和足够的资源。(4)变更控制。1）控制软件开发中的变更对于业务和财务控制以及技术稳定性至关重要。2）要以可预测的时间表开发高质量的软件，必须在整个开发过程中以合理的稳定性建立和维护需求。3)必须进行更改，但必须有条不紊地进行管理和引入。

**可重复级**：1.与初始过程相比，可重复过程具有一个重要的优势:它提供了承诺控制。2.与最初的流程相比，这是一个巨大的进步，以至于组织中的人们倾向于相信他们已经掌握了软件问题。3.可重复流程级别的组织在面临新的挑战时面临重大风险。

**可重复的过程这一级别的挑战：**(1)新的工具和方法可能会影响流程的执行方式，从而破坏组织所依赖的直观历史基础的相关性。(2)当组织必须开发一种新产品时，它正在进入新的领域。例如，具有开发编译器经验的软件组如果被指派编写控制程序，则可能会遇到设计，调度和估计问题。(3)重大的组织变化可能具有高度破坏性。在可重复的过程组织中，新经理没有有序的基础来理解正在发生的事情，新团队成员必须通过口口相传来学习诀窍。

**改进的关键：**(1)建立一个过程组。过程组是一个专注于改进软件开发流程的技术小组。过程组的职责包括定义开发流程、识别技术需求和机会，并每季度进行一次管理评审，检查流程的状态和绩效(2)建立软件开发过程体系结构，描述正确执行开发过程所需的技术和管理活动。体系结构是将开发周期分解为任务的结构。分解继续进行，直到每个定义的任务由单个或单个管理单元执行。(3)如果它们还没有到位，请引入一系列软件工程方法和技术。其中包括设计和代码检查，正式设计方法，库控制系统以及综合测试方法。还应该考虑原型设计，以及采用现代实现语言。

**已定义级：**通过确定的过程，本组织已经取得了重大持续进展的基础。例如，开发小组在面临危机时，可能会继续使用已定义的流程。

**已定义级需要改进的东西**：定义的过程仍然只是定性的:几乎没有数据表明正在发生什么或过程的有效性。关于软件过程度量的价值和使用的最佳度量存在相当大的争论。

**已管理级的关键步骤：**1.建立一套最低限度的、基本的过程测量，以确定每个过程步骤的质量和成本参数。2.建立流程数据库，并配备资源进行管理和维护。3.提供足够的过程资源来收集和维护这些数据，并就其使用向项目成员提供建议。4.评估每个产品的相对质量，并通知管理层没有达到质量目标。

**已管理级**：1.管理流程最大的潜在问题是收集数据的成本。收集和维护数据的成本很高。2.谨慎处理数据收集，并预先精确定义每条数据。3.除非明确定义，否则生产力数据通常是无意义的。例如：新代码或更改的代码，非注释、非空行、可执行指令或等效汇编指令。4.测量数据的趋势是使用它来比较几个组，并对排名最低的组施加压力。这是对过程数据的错误应用。5.过程数据不得用于比较项目或个人。其目的是阐明正在开发的产品，并为改进过程提供知情的基础。

**从已管理级到优化级**：1.支持过程数据的自动采集。有些数据无法手动收集，所有手动收集的数据都会出现错误和遗漏。2.使用此数据既分析又修改流程，以防止问题并提高效率。

**优化级：**1.到现在，软件开发经理主要关注他们的产品，通常只收集和分析与产品改进直接相关的数据。2.在优化过程中，数据可用于实际调整过程本身。

**修复bug示例**：1.作者经验:通过检查发现并修复一个bug大约需要一到四个工作小时，在功能或系统测试中发现并修复一个bug大约需要15到20个工作小时。2.但是，某些类型的错误要么不经济，要么几乎不可能找到，除非通过机器。3.为了降低清除bug的成本，应将检查与任何其他具有成本效益的技术一起强调。

# 《人月神话》

**进度：**在众多软件项目中，缺乏合理的时间进度是造成项目滞后的最主要原因，它比其他所有因素加起来的影响还大。首先，我们对估算技术缺乏有效的研究。第二，我们采用的估算技术隐含地假设人和月可以互换，错误地将进度与工作量相互混淆。第三，由于对自己的估算缺乏信心，软件经理通常不会有耐心持续地进行估算这项工作。第四，对进度缺少跟踪和监督。其他工程领域中，经过验证的跟踪技术和常规监督程序，在软件工程中常常被认为是无谓的举动。第五，当意识到进度的偏移时，下意识（以及传统）的反应是增加人力。这就像使用汽油灭火一样，只会使事情更糟。越来越大的火势需要更多的汽油，从而进入了一场注定会导致灾难的循环。

**DorothySayers**在她的“TheMindoftheMaker”一书中，将创造性活动分为三个阶段：构思、实现和交流。1.书籍、计算机、或者程序的出现，首先是作为一个构思或模型出现在作者的脑海中2.借助钢笔、墨水和纸，或者电线、硅片和铁氧体，在现实的时间和空间中实现它们3.当某人阅读书本、使用计算机和运行程序的时候，他与作者的思想相互沟通，从而创作过程得以结束。

**乐观主义**：所有的编程人员都是乐观主义者。所有系统编程的进度安排背后的第一个假设是：一切都将运作良好，每一项任务仅花费它所“应该”花费的时间。

**人月：1.**第二个谬误的思考方式是在估计和进度安排中使用的工作量单位：人月。2.人数和时间的互换仅仅适用于以下情况：某个任务可以分解给参与人员，并且他们之间不需要相互的交流。这在割小麦或收获棉花的工作中是可行的；而在系统编程中近乎不可能。3.当任务由于次序上的限制不能分解时，人手的添加对进度没有帮助4.对于可以分解，但子任务之间需要相互沟通和交流的任务，必须在计划工作中考虑沟通的工作量。因此，相同人月的前提下，采用增加人手来减少时间得到的最好情况，也比未调整前要差一些5.沟通所增加的负担由两个部分组成，培训和相互的交流。每个成员需要进行技术、项目目标以及总体策略上的培训。这种培训不能分解，因此这部分增加的工作量随人员的数量呈线性变化.6.相互之间交流的情况更糟一些。如果任务的每个部分必须分别和其他部分单独协作，则工作量按照n(n-1)/2递增。

**系统测试**：1.由于乐观主义，通常实际出现的缺陷数量比预料的要多得多。2.进度安排：1/3计划1/6编码1/4构件测试和早期系统测试1/4系统测试，所有的构件已完成。

**Brooks法则**：向进度落后的项目中增加人手，只会使进度更加落后。众多软件项目中，缺乏合理的时间进度是造成项目滞后的最主要原因，它比其他所有因素加起来的影响还要大。

# SCRUM

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。

**三个角色**：ScrumMaster、ProductOwner（产品负责人）和Team（团队）。•三个工件：ProductBacklog（产品待办事项）、SprintBacklog（Sprint待办事项）和可交付产品增量（也有说是燃尽图）。•五大仪式（事件）：Sprint（冲刺）、SprintPlanning（Sprint规划）、SprintDailyStandup（每日站会）、SprintReview（Sprint评审）和SprintRetrospective（回顾）。•五大价值观：Courage（勇气）、Openness（开放）、Focus（专注）、Commitment（承诺）和Respect（尊重）。

**ProductOwner**:1.产品负责人负责最大化产品以及开发团队工作的价值。产品负责人是唯一有权要求团队做事以及改变列表条目优先级的人。2.持有产品愿景、代表业务（thebusiness）、代表客户、拥有产品列表、划定故事优先级、设立故事的接收标准、有空回答团队成员们的问题。3.产品负责人和团队其他人之间有一层天生的紧张关系，产品负责人总想要更多，而团队则必须维护可持续的速率。只要不是单方面说了算，这层紧张关系就还是有益的。4.不要合并产品负责人和scrummaster。5.产品负责人是管理产品待办事项列表的唯一责任人。产品待办事项列表的管理包括:•清晰地表达产品代办事项列表条目•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•确保开发团队所执行工作的价值•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•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**1.ScrumMaster负责确保Scrum被理解并实施。2.Scrummaster担当教练角色，引领团队达到更高级的凝聚力、自组织和表现。ScrumMaster以各种方式服务于开发团队,包括:•指导开发团队自组织和跨功能•教导并领导开发团队创造高价值的产品•移除开发团队进展过程中的障碍•按需推动Scrum事件•在Scrum还未完全被采纳和理解的组织环境下指导开发团队•CoachandTeacher.3.Scrummaster也有可能一并担当直接贡献的责任。这种情况下我们称之为工作型（working)scrummaster，或贡献型(contributor)scrummaster。

**团队**•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。•全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人（特性团队）。•通常是七个人，再或多或少两个人，也就是说，五个到九个人。一共11人。

**全功能团队、特性团队:**全功能团队是具有不同职能专业或多学科技能的团队。当一个团队拥有满足需求的所有技能和资源时，就称为真正的跨职能。这意味着它不依赖于跟其他团队的工作交接，也不用等待其他团队的工作。•反例：职能团队，UI团队、业务逻辑团队、测试团队和数自组织团队些人将其称之为特性团队。这个团队对于某个特性从开始到结束全程负责。•反例：组件团队，即只是负责数据访问层或用户接口层的团队。当按照这种方式组建团队时，就要承担开发特性时一个团队等待另一个团队的风险。

**自组织团队:**1.《为什么我们需要自组织团队》提出必须让团队竭尽他们所有的专业能力，不仅仅是完成他们的工作任务，还要自我监督和控制，自己做决定，甚至设计自己的流程。2.自组织性高的团队往往也能够比其他团队带来更多的好处，比如能够传递更多的商业价值、更加高效地协同工作以及学的更快等。3.降低管理成本。

**微观管理及其危害**：1.微观管理（Micromanagement）一般是指：在对员工的工作管理中，管理者过度关注和控制工作细节的管理风格和管理行为。•客观地说，关注细节并非坏事。但是一旦过于关注和控制细节，就会带来种种问题：•第一，会导致团队成员失去主观能动性。由于完全沦为执行者，导致员工只能按照管理者的思路走，从而失去了自主性和创造力。•第二，会导致工作中出现决策等待和低效。当实际工作场景与管理者最初设想的不一致时，员工无法继续按照管理者最初的思路工作，就只能把问题反馈给管理者，等待管理者作出决策。•第三，会影响到团队的积极性和士气。由于团队成员经常处于要向管理者汇报的压力中，经常处于被管理者纠偏的状态，很容易让团队成员感到不自信和缺乏成就感。2.微观管理是打造自组织团队的最大障碍之一。

**用户故事**•在20世纪90年代末，Kent在开发软件的过程中发现，其中最大的问题莫过于使用文档来精确描述我们想要的东西，即需求。•同样的一份文档，阅读的人不同，各自得到的信息也不一样。这种缺乏共识的情况，称为“低质量的需求”•停止写出完美文档的“执念”。•用户故事之所以得名，并不是要人们如何写出更好的用户故事，而是如何在协作中更好地使用它。

**RonJeffries的3C原则**：1.卡片（Card）（placeholder，占位符）：在一堆卡片上写下你期望的软件特性2.交谈（Conversation）：聚在一起对要开发的软件进行深人讨论3.确认（Confirmation）：对完工条件进行确认

**用户故事是占位符**•用户故事不是完整的需求或说明书，它们是占位符。•它们的信息量足以提醒团队有东西要完成，但我们刻意地不过多探讨细节......直到必需之时。•需要阐述用户故事细节时，我们喜欢使用召集相关团队成员参与交谈的形式。交谈的目标在于，对故事内容以及所需完成的工作达成共识。相对于依赖书面文档，使用实时对话的方式能更高效地达成此目标。它有更多的信息流动。

**用户故事接收标准**•交谈中，如果某一刻大家觉得对用户故事已有共识，那就该写验收标准了。找出一列测试，直到所有参与交谈的人都认同测试通过意味着故事按预期实现即可，再用简单易懂的话记录下来。•接收标准回答如下的问题:“我们如何得知它何时已完工?”•理想情况下，团队应该可以根据接收标准写出自动化测试，甚至是在功能实现之前（TDD）。位于产品列表下方的故事可能不会很快被实现，接收标准可以降低精度。

**“DefinitionofDone”（DoD）**:“DefinitionofDone”（DoD，完成的定义）是敏捷软件开发中的一个关键概念，用于描述一个用户故事、任务或功能何时可以被认为真正“完成”。它是团队对完成工作的标准化定义，确保开发过程中每个增量都符合质量要求并准备好交付。•1.提高透明度：DoD明确了完成的标准，减少了团队之间的沟通误解。•2.保证质量：通过对代码质量、测试和文档的明确要求，确保产出的增量软件可用且可靠。•3.支持验收流程：DoD是验收用户故事或功能的依据。如果工作未达到DoD，任务不能被标记为完成。•4.防止技术债：通过明确完成标准，避免开发过程中留下未解决的问题。

**行为驱动开发（Behavior-DrivenDevelopment，BDD）:**行为驱动开发（BDD）是一种基于敏捷的软件开发方法论，其核心思想是通过定义软件的行为来驱动开发过程。BDD的重点是增强团队对需求的理解，并确保开发的软件满足业务目标。•BDD是从测试驱动开发（TDD）演化而来的，强调在开发开始之前，用自然语言描述软件应如何行为。它通过让技术人员、业务人员和测试人员围绕共同的需求语言进行协作，消除了沟通中的歧义。

**BDD的实践流程:**1.编写用户故事：•用户故事采用以下模板描述功能需求：•Asa[userrole],•Iwant[feature/goal],•Sothat[businessvalue].例子：•作为一名电子商务网站用户，•我希望能够将商品添加到购物车，•以便稍后结算。•2.定义验收标准（AcceptanceCriteria）：•验收标准是用户故事“完成”的具体条件，通常以场景的形式定义：Scenario:[场景描述]Given[初始状态]When[执行的操作]Then[预期的结果]

**用户故事INVEST原则**•独立性（Independent）—要尽可能的让一个用户故事独立于其他的用户故事。•可协商性（Negotiable）—一个用户故事的内容要是可以协商的，用户故事不是合同。•有价值（Valuable）—每个故事必须对客户具有价值（无论是用户还是购买方）。•可以估算性（Estimable）—开发团队需要去估计一个用户故事以便确定优先级，工作量，安排计划。•短小（Small）—一个好的故事在工作量上要尽量短小，至少要确保的是在一个迭代或Sprint中能够完成。•可测试性（Testable）—一个用户故事要是可以测试的，以便于确认它是可以完成的。

**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Ba语cklogI言tem描述的客户需求）

**什么是用户故事地图**：•用户故事地图是一门在需求拆分过程中保持全景图的技术。•敏捷软件开发中使用用户故事地图来发现、管理需求。•2008年提出。•44%使用Storymapping实践。

**故事地图解决的问题**：1.全局视角缺失：传统的待办事项列表（Backlog）往往只关注单个功能，缺乏对整体产品的全局视角，导致团队难以理解产品的整体结构和用户体验流程。用户故事地图通过横向展示用户活动，纵向排列功能细节，提供了产品的全貌视图，帮助团队更好地理解用户的使用路径。2.需求优先级难以确定：在复杂的项目中，众多需求可能让团队难以确定哪些功能应优先开发。用户故事地图通过将用户故事按照用户活动和任务进行组织，清晰地展示各功能的相对重要性，便于团队合理安排开发顺序。3.缺乏用户需求聚焦：开发过程中，团队可能过于关注技术实现，忽视了用户的真实需求。用户故事地图强调从用户角度出发，确保开发的功能真正满足用户需求，提升用户满意度。4.难以理解功能之间的关系：在大型项目中，不同功能之间的关系可能复杂，团队难以把握。用户故事地图通过结构化的方式展示功能之间的关联，帮助团队更好地理解和管理这些关系。5.发布计划不明确：在敏捷开发中，确定每次发布的功能范围至关重要。用户故事地图通过清晰地展示各功能的优先级和依赖关系，帮助团队制定合理的发布计划，确保每次发布都能为用户提供有价值的功能。

**Sprint计划会议准备**：1.所有重要的backlog条目都已经根据重要性被评过分,不同的重要程度对应不同的分数。分数只是用来根据重要性对backlog条目排序。2.所有人都可以编写添加条目，但只有ProductOwner才能决定优先级。

**Sprint计划会议目标**：以终为始•sprint目标（尽可能简单的语言，团队成员认同）。•团队成员名单（以及他们的投入程度,如果不是100%的话）。•sprintbacklog（即sprint中包括的故事列表）。•确定好sprint演示日期。•确定好时间地点,供举行每日scrum会议。

**一个Sprint有多长？**：1.确定Sprint长度：Sprint持续多久才算合适？•时间短：“敏捷”——短反馈周期=频繁交付=频繁客户反馈=错误方向持续时间短=学习改进速度快……•时间长：更多时间作充分准备、解决问题、达成目标，不会被接二连三的会议压的不堪重负。2.当前，Scrum周期通常为2个星期。

**估算：**•估算是很困难的，因为这是在预测未来。•估算经常是错误的，但是估算过程仍然是有用的。估算过程是管理前方的不确定性的契机，它可以被当做风险管理的工具，让你可以发现误解、不一致以及需要进一步调査的地方。•团队共同完成估算可以让大家建立对工作一致的理解。但是，根据收益递减原理，你不应在估算上花太多的时间。你可以做出一个快速但不那么准确的估计，也可以再多花一点时间做一个更准确的估计，但花上几天的时间得出精确的小时数就没用了。

**估算单位（storypoint）**：Storypoint：故事点，选取可识别的最小用例为2个storypoint.其它估算都是相对值，在所有sprint中保持该相对值一致。•另外的好处，可以在估算时缩小人与人的能力误差（10倍程序员）。我们在估算速度时，估计我们能在一个迭代周期内能够完成的storypoint。

**估算过程—工具“计划扑克”**：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.当团队理解了这个条目之后，每个团队成员按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，出牌时数字朝下扣在桌面上。5.阅读者向大家确认是否都已经确定估算结果，确认后，数”1,2,3",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**：•1、需求理解不同。例如：复杂登陆和简单登陆。•2、技术不熟悉。例如：没有嵌入式经验的工程师估算嵌入式系统开发内容。

**扑克牌估算法价值**•传统估算通常是一个人在思考，而使用估算扑克估算时，鼓励跨职能团队的多个团队成员参与估算，团队成员可以从不同的视角来思考和分析问题，估算的过程中考虑的更加全面、估算也更加准确。•在估算的过程中，团队对估算的结果进行讨论和评判，在一个高度透明的环境下，估算的结果更加真实和客观。这样也避免了很多时候过于武断，或是拍脑袋做出的决定。•估算的过程也是一个知识分享和学习的过程，对某一个条目不清楚的成员通过其他成员的阐述会增加对该条目涉及到的要点的认识。

**卡片队列估算法：**(1)把每个条目的描述写在单独的卡片上，这个应该在估算前就准备好。(2)把所有的卡片摞成一摞放在桌上。(3)拿出第一张卡片，放在旁边。(4)再拿出一张，问:“这个比第一个大还是小?”为了回答这个问题，你可能需要和其他人讨论一下这个工作项是干什么的。这需要时间，但别太长，记住这是一个快速估算。(5)当你决定了它更大或更小后，把它放在原先卡片的上方或下方，形成一列。(6)重复第4和第5步，直到用完所有的卡片。(7)现在你可以把所有的卡片分组了—这也是该技术的第二阶段。从桌上底部(最小)的卡片开始，声明第一组的估算是“小的”。(8)继续向上，决定到何时为止卡片已经变大到需要用一个新的估算值（如“中等”）了，让团队决定。(9)很快，你就可以把整列卡片再过一遍，并加总（5个的估计是S,4个M,等等）以得到整个列的总和，这也就是需求的总工作量估计。从队列最下面的部分开始，第一部分可以被当做S。

**“金发女孩”估算技术**：作为对工作项大小估计的替代，你可以把工作项调整到合适的大小。你不再是指定每个工作项的大小，而是去分割或组合工作项，让它们的规模大致相当，并便于后续操作。

**故事（story）与任务（task）**•Story是可以交付的东西，Task是不可以交付的，ProductOwner对Task不关心；

**计划会议**：1.定下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。2.确定技术故事：需要完成但是不属于可交付物的东西，如：-安装持续构建服务器-编写系统设计概览。

**跟踪进度—燃尽图**：包含信息：•该Sprint共需要完成70个故事点；•目前团队还剩下15个故事点要完成•估计：按照目前速度该Sprint目标可以达到。

敏捷软件开发工作环境：•让团队坐在一起•让ProductOwner无路可走•让经理和教练无路可走。

**静默时间**•“flow”心流时间，全神贯注在某件事情，以至于忘记了时间的流逝。•公司层面可以为程序员设计专门的静默时间。

**每日站会—DailyScrum**1.不超过15分钟2.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。”3.移动任务板上的即时贴到对应的地方4.每日例会一结束就要计算剩余工作故事点并更新燃尽图5.团队每日报到、简短、及时开始与结束、聚焦重点、有规律性

**Sprint演示会议**：为什么定义Sprint结束于演示：•其他人可以了解你的团队在做什么•团队得到认可，团队成员感觉很好•不同的团队得到交流，讨论各自工作•演示可以吸引相干人士的关注，并得到重要的反馈•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。

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

**Sprint回顾会议**•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。•主题：我们怎样才能在下个Sprint中做的更好，不是追究责任！

**Sprint回顾——活动列表：**1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。•我们换到一个封闭的房间中,或者舒适的沙发角,或者屋顶平台等等类似的场所。只要能够在不受干扰的情况下讨论就好。•我们一般不会在团队房间中进行回顾,因为这往往会分散大家的注意力。•指定某人当秘书。2.Scrummaster向大家展示sprintbacklog,在团队的帮助下对sprint做总结。包括重要事件和决策等。3.我们会轮流发言。每个人都有机会在不被人打断的情况下讲出自己的想法,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。4.我们对预估生产率和实际生产率进行比较。如果差异比较大的话,我们会分析原因。5.快结束的时候,Scrummaster对具体建议进行总结,得出下个sprint需要改进的地方。

Sprint回顾——使用白板•Good：哪些做法可以保持•Couldhavebeenbetter：那些做法需要改变•Improvements：具体改进想法

**Scrum局限：**1.没有技术实践！2.可以使用极限编程技术实践：测试驱动开发、简单设计、重构、持续集成等等。

# XP极限编程

**XP的心跳——开发周期**：这是程序员实现一个工程任务（最小的调度单位）并与系统其余部分集成的地方。

**开发周期**：1.1张顶层任务卡：“用户管理：登录、登出、组。”2.获取结对编程伙伴3.讨论任务4.测试用例是什么？如果对任何事情不确定，向他人寻求帮助。5.编写测试用例。**开发周期**：1.测试失败2.编写代码3.运行所有测试用例4.迭代测试用例和代码5.如有需要，重构6.集成，包括测试

注意事项：1.程序员成对编程。2.开发由测试驱动。3.成对不仅仅是使测试用例运行。成对为系统的分析、设计、实现和测试增加了价值。4.集成紧随开发之后，包括集成测试。

**变更成本**:在某些情况下，软件变更成本随时间指数上升的趋势可以被平缓化。如果我们能够平缓这个曲线，关于软件开发最佳方式的传统假设就不再成立。

变更成本的假设:1.软件工程的一个普遍假设是，随着时间的推移，修改程序的成本会指数上升。2.“软件中修复一个问题的成本随时间指数上升。如果在需求分析阶段发现问题，修复成本可能只需要一美元，但一旦软件投入生产，修复成本可能高达数千美元。”

**降低变更成本的技术**:软件开发界在最近几十年投入了大量资源试图降低变更成本——更好的语言、更好的数据库技术、更好的编程实践、更好的环境和工具、新的符号。

XP的技术前提:如果变更成本随时间缓慢上升，你的行动方式将与在成本指数上升的假设下完全不同。

**如果变更成本很小，你会怎么做?**1.你会尽可能晚地做出重大决策，以推迟决策成本，并尽可能确保决策正确。2.你只会实现你必须实现的部分，希望你预见到的明天的需求不会实现。3.你只会在设计中引入简化现有代码或使编写下一段代码更简单的元素。

**保持变更成本低:**1.在技术方面，对象是一个关键技术。2.简单的设计，没有额外的设计元素——没有尚未使用但预计将来会使用的想法。3.自动化测试，以便我们有信心知道我们是否意外改变了系统的现有行为。4.大量修改设计的实践，以便在需要更改系统时，我们不会害怕尝试。

**学习驾驶:问题与解决方案资源**1.问题—风险的巨大成本，以及通过选择来管理该风险的机会2.塑造解决方案所需的资源：在周期后期进行更改而不显著增加成本的自由。小幅度调整与关注1.我们需要通过进行许多小幅度调整来控制软件的开发，而不是通过进行几次大幅度调整，有点像驾驶汽车。2.始终保持关注。

**四个价值观**:沟通、简单、反馈、勇气

**沟通**:1.开发者与开发者之间2.开发者与客户之间3.开发者与管理层之间4.XP强制开发者进行沟通：单元测试、结对编程、任务估算

**简单**：1.“能够起作用的最简单的事情是什么？”2.XP认为，今天做一件简单的事情，明天如果需要再花一点代价去改变它，比今天做一件可能永远都不会用到的更复杂的事情要好。

**沟通与简单**：1.你沟通得越多，就越能清楚地看到究竟需要做什么，对那些真正不需要做的事情就越有信心。2.你的系统越简单，需要沟通的内容就越少，这会导致更完整的沟通，特别是如果你能简化系统到只需要更少的程序员时。

**反馈**：1.关于系统当前状态的具体反馈是绝对无价的。2.首先，反馈在分钟和天的尺度上起作用：开发者的单元测试、开发者对客户的即时估算、进度跟踪人员向整个团队提供反馈3.反馈也在周和月的尺度上起作用：客户的功能测试运行软件

**反馈、沟通与简单**：1.你拥有的反馈越多，沟通就越容易。2.简单的系统更容易测试（反馈）。3.编写测试为你提供了系统可以有多简单的焦点。

**勇气——例子**：1.在30周中的25周，存在一个大问题2.架构缺陷3.修复缺陷。这使得50%的测试无法通过。

**勇气——丢弃代码**：再做一次。

**勇气——爬山算法**：XP的设计策略类似于爬山算法。你先得到一个简单的设计，然后使其稍微复杂一些，再稍微简单一些，然后再稍微复杂一些。爬山算法的问题在于达到局部最优解，此时没有小的改变可以改善情况，但大的改变可以。

**勇气与其他价值观**：1.如果没有前三个价值观，勇气本身只是单纯的黑客行为。2.沟通支持勇气，因为它打开了进行更多高风险、高回报实验的可能性。3.简单支持勇气，因为你可以负担得起在一个简单的系统中更加勇敢。4.具体反馈支持勇气，因为如果你可以按下一个按钮并在最后看到测试变为绿色，你会感到更安全地对代码进行激进的手术。

**价值观的实践——尊重**：1.一个隐藏在其他四个价值观之下的——尊重。2.如果团队成员彼此不在乎以及他们正在做的事情，XP就注定失败。3如果团队成员不在乎项目，没有什么可以拯救它。

**回归基础：必须做的事情**：1.我们希望做我们必须做的一切，以实现稳定、可预测的软件开发。2.开发的四个基本活动是编码、测试、倾听和设计。

**编码**：1.它是你的工作成果。2.其他方面：学习、沟通：精确

**测试(1)**：1.“无法测量的东西就不存在”—科学哲学（对编程也成立）2.测试让我有机会在实现方式之外思考我想要的东西。然后测试告诉我我是否实现了我认为自己实现的东西。

**测试(2)—测试感染**：1.测试告诉你何时完成—当测试运行时，你暂时完成了编码。2.当你想不到任何可能失败的测试来编写时，你就完全完成了。

**测试(3)—为什么测试？**：1.长期答案是测试可以让程序更长久地运行（如果测试被运行和维护）。你可以更长时间地修改程序。2.短期原因：信心3.编程和测试结合在一起也比单纯编程更快：生产力的提高来自于减少调试所花费的时间。

**测试(4)—什么类型的测试？**1.我们将有程序员编写的单元测试，以说服他们自己的程序按照他们认为的方式工作。2.我们还将有客户编写（或至少由客户指定）的功能测试，以说服他们整个系统按照他们认为整个系统应该的方式工作。

**倾听**：1.程序员向业务人员请教以获得项目的业务视角。2.程序员帮助业务人员了解软件中什么是容易的，什么是困难的。3.相互倾听（程序员之间、程序员与客户之间）

**设计**：1.仅仅倾听、编写测试用例、使其运行、再倾听、编写测试用例、使其运行是不够的吗？不是的2.唯一使下一个测试用例运行的方法是破坏另一个。3.或者唯一使测试用例运行的方法是远比值得的麻烦。4.良好的设计：良好的设计将逻辑组织起来，使得对系统一个部分的更改不总是需要对系统的另一个部分进行更改良好的设计确保系统中的每一段逻辑都有且只有一个归属地、良好的设计将逻辑放在它操作的数据附近、良好的设计允许通过仅在一个地方进行更改来扩展系统。

**结论**：1.因此，你编码是因为如果你不编码，你什么也没做。2.你测试是因为如果你不测试，你就不知道何时完成了编码。3.你倾听是因为如果你不倾听，你就不知道要编码或测试什么。4.你设计是为了能够无限期地继续编码、测试和倾听。

**快速概述**

**介绍**：我们将依赖于简单实践之间的协同作用，这些实践通常在几十年前被认为不切实际或天真而被放弃。

**计划游戏**：1.商业考虑和技术考虑都不应占据主导地位。2.软件开发始终是可能与理想之间不断演变的对话。3.商业人员需要决定：范围—为使系统在生产中具有价值，必须解决多少问题？优先级—如果最初只能选择A或B，你想要哪一个？发布的组成—在软件比没有软件更好之前，需要做多少或做多少？发布的日期—哪些重要日期软件（或部分软件）的存在会带来很大不同**?**4.商业不能在真空中做出这些决定。开发需要做出技术决策，为商业决策提供原材料。5.技术人员决定：估算—实现一个功能需要多长时间？后果—有些战略性的商业决策只有在了解技术后果后才能做出。过程—工作和团队将如何组织？详细计划—在一个发布中，哪些故事将首先完成？

**小规模发布**:1.每次发布都应该尽可能小，包含最有价值的业务需求。2.发布作为一个整体必须有意义。3.计划一个月或两个月的时间要比计划六个月或一年的时间要好得多。

**隐喻**:1.每个XP软件项目都由一个单一的总体隐喻指导。2.XP中的隐喻取代了其他人所说的“架构”的大部分内容。3.架构并不一定使系统具有任何意义上的内聚性。4.选择一个系统隐喻，通过一致地命名类和方法来保持团队在同一页面上。5.例如，克莱斯勒的工资系统是作为生产线构建的。在另一家汽车制造商中，汽车销售被构建为材料清单。6.搜索引擎是一大群蜘蛛，在网上四处寻找要捕捉的东西，然后把东西带回巢穴。

**简单设计**1.运行所有测试。2.没有重复的逻辑。警惕隐藏的重复，如平行类层次结构。3.表达程序员认为重要的每一个意图。4.拥有尽可能少的类和方法。5.反对：“为今天实现，为明天设计。”

**测试**:1.没有自动化测试的任何程序功能实际上都不存在。2.程序员编写单元测试，以便他们对程序运行的信心可以成为程序本身的一部分。3.客户编写功能测试，以便他们对程序运行的信心也可以成为程序的一部分。

**重构**:1.在实现程序功能时，程序员总是会问是否有办法改变现有程序，使添加功能变得简单。2.你不会基于猜测进行重构；当系统要求你这样做时，你才会进行重构:当系统要求你复制代码时，它就是在要求重构。

**结对编程**:所有生产代码都是由两个人看着一台机器编写的，使用一个键盘和一个鼠标。1一个伙伴，拥有键盘和鼠标的人，正在思考如何最好地实现这个方法。2另一个伙伴则更具战略性地思考：这种整体方法会起作用吗？还有哪些测试用例可能还没有通过？是否有办法简化整个系统，使当前问题消失？3.如果两个人在早上结对，下午他们可能很容易与其他同事结对。4.如果你负责一个你不熟悉的领域的任务，你可能会请最近有经验的人与你结对。

**集体所有权**:1.任何看到有机会为代码任何部分增加价值的人都被要求随时这样做。2.没有所有权。在过去，没有人拥有任何特定的代码部分。如果有人想更改一些代码，他们会根据自己的目的进行更改，不管它是否与已有的代码很好地结合。3.个人代码所有权。只有代码的官方所有者才能更改代码的一部分。任何其他看到代码需要更改的人都必须向所有者提交请求。人们不愿打扰代码所有者。

**持续集成**1.代码在几小时后集成和测试——最多一天的开发。2.一种简单的方法是专门有一台机器用于集成。3.当机器空闲时，有代码要集成的结对会坐下来，加载当前版本，加载他们的更改（检查并解决任何冲突），并运行测试直到通过（100%正确）。4.一次集成一组更改效果很好，因为谁应该修复失败的测试显而易见——应该是我们，因为我们肯定破坏了它，因为上一对离开时测试是100%。

**40小时工作制**:1.加班是项目上严重问题的症状。2.是否将其转化为每周在工作场所正好40小时并不太重要。

**现场客户**:1.真正的客户必须与团队坐在一起，随时回答问题，解决争端，并设定小规模优先级。2.对此规则的主要反对意见是，正在开发的系统的真正用户太有价值，不能交给团队:管理者将不得不决定哪个更有价值——让软件更早、更好地工作，还是拥有一个人或两个人的输出。

**编码标准**:如果你将让所有这些程序员从系统的一个部分换到另一个部分，每天换几次伙伴，并不断重构彼此的代码，你根本无法负担拥有不同的编码实践。

**这样如何能行**？这些实践相互支持。一个实践的弱点被其他实践的优势所弥补。

**本章如何组织**？1.上述实践没有一个是独特或原创的。2.它们自编写程序以来就已经被使用。3.大多数这些实践由于其弱点变得明显而被更复杂、开销更高的实践所取代4.如果这些弱点现在被其他实践的优势所弥补会怎样？

**计划游戏**1.你不可能仅凭一个粗略的计划就开始开发。2.你不能不断更新计划——那会花费太多时间并让客户不安:客户根据程序员提供的估算自己更新计划。你一开始有足够的计划，让客户对接下来几年可能实现的内容有一个大致的了解。3.你进行短周期发布，因此计划中的任何错误最多只会影响几周或几个月。4.你的客户与团队坐在一起，因此他们可以迅速发现潜在的变化和改进机会。

**短周期发布**:1.你不可能在几个月后就投入生产。计划游戏帮助你专注于最有价值的故事，因此即使是小型系统也具有商业价值、你持续集成，因此打包发布成本很小。2.你的测试将缺陷率降低到足够低，因此你不需要在允许软件发布之前进行漫长的测试周期。3.你可以进行简单的设计，足以应对此次发布，而不是永远。

**隐喻**：1.不可能仅凭一个隐喻就开始开发。那里没有足够的细节，而且，如果你错了怎么办？：你很快就能从实际代码和测试中获得关于隐喻是否在实践中有效的具体反馈。你的客户能够用隐喻来谈论系统。你通过重构不断细化对隐喻在实践中意义的理解。

**简单设计**：1.你不可能仅凭今天的代码就拥有足够的设计。你会将自己设计到一个死胡同，然后你将无法继续系统的发展。你习惯于重构，因此进行更改不是问题。2.你有一个清晰的总体隐喻，因此你确信未来的更改会沿着一个收敛的方向发展。3.你与伙伴一起编程，因此你有信心你正在做出一个简单的设计，而不是愚蠢的设计。

**测试**：1.你不可能编写所有这些测试。那会花费太多时间。程序员不会编写测试。设计尽可能简单，因此编写测试并不那么困难。结对编程——伙伴的压力当你看到所有测试都在运行时，你会感觉很好。当客户看到他们所有的测试都在运行时，他们会感觉系统很好。

**重构**：1.你不可能一直重构系统的结构。那会花费太长时间，难以控制，而且很可能破2.你成对编程，因此你更有可能有勇气进行艰难的重构，而且你不太可能破坏东西。3.你有简单的结构，因此重构更容易。4.你有测试，因此你不太可能在不知情的情况下破坏东西。5.你持续集成，因此如果你不小心破坏了远处的东西，或者你的重构与他人的工作冲突，你将在几小时内知道。6.你休息得很好，因此你更有勇气，也更不可能犯错。

**结对编程**：1.你不可能成对编写所有生产代码。那会太慢。如果两个人不和怎么办？编码标准减少了琐碎的争吵。每个人都休息得很好，进一步减少了无利可图的……呃……讨论的机会。2.结对编写测试，让他们在着手实现核心内容之前有机会对理解进行对齐。3.结对有隐喻来指导他们关于命名和基本设计的决策。4.结对在简单的结构中工作，因此他们都能理解发生了什么。

**集体所有权**：1.你不可能让每个人都有可能更改任何地方的东西。人们会到处破坏东西，集成的成本会大幅上升。你在足够短的时间内集成，因此冲突的机会减少。2.你编写并运行测试，因此意外破坏东西的机会减少。3.你成对编程，因此你不太可能破坏代码，程序员更快地了解他们可以有利地更改的内容。4.你遵循编码标准，因此你不会陷入可怕的花括号大战。

**持续集成**：1.你不可能在仅工作几小时后就进行集成。集成花费的时间太长，冲突太多，破坏东西的机会也太多。2.你可以快速运行测试，因此你知道你没有破坏任何东西。3.你成对编程，因此需要集成的更改流减少了一半。4.你重构，因此有更多的小块，减少了冲突的机会。

**40小时工作制**：1.你不可能每周工作40小时。计划游戏为你提供了更有价值的工作。计划游戏和测试的结合减少了你比预期有更多的事情要做的糟糕惊喜的频率。整体实践帮助你以最快速度编程，因此你无法更快。

**现场客户**：1.你不可能让真正的客户全职坐在团队中。他们可以在其他地方为业务创造更多的价值。他们可以通过编写功能测试为项目创造价值。他们可以通过为程序员做出小规模的优先级和范围决策为项目创造价值。

**编码标准**：1.你不可能要求团队按照共同的标准编码。程序员非常个人主义，宁愿辞职也不愿将花括号放在其他地方。整个XP让他们更有可能成为获胜团队的一员。

**测试策略**：我们将在编码之前编写测试，每分钟都进行。我们将永久保留这些测试，并经常一起运行它们。我们还将从客户的角度推导出测试。

**独立且自动**：1.首先，每个测试不与其他你编写的测试交互。2.测试也是自动的。3.当压力水平上升，当人们工作过度，当人类判断开始失效时，测试最有价值。因此，测试必须是自动的——给出一个明确的系统是否按预期运行的指示。

**无法绝对测试一切**：1.你应该测试可能会出错的事情。2.测试是一种赌博。3.测试可以通过的一种方式是当你没有预料到会成功的测试却成功了。4.你只会编写那些能够带来回报的测试。

**谁编写测试？**——程序员：1.如果方法的接口有任何不清晰之处，你在编写方法之前编写测试。2.如果接口清晰，但你认为实现会有一点复杂，你在编写方法之前编写测试。3.如果你想到了代码应该按预期工作的一个不寻常的情况，你编写一个测试来传达这种情况。4.如果你后来发现一个问题，你编写一个测试来隔离这个问题。5.如果你即将重构一些代码，并且你不确定它应该如何表现，并且没有针对所讨论的行为方面的测试，你先编写测试。

**程序员测试**：1.程序员逐方法编写测试。2.程序员编写的单元测试始终运行在100%。3.因为程序员控制单元测试的编写和执行，他们可以保持测试完全同步。

**谁编写测试？**——客户：1.客户逐故事编写测试。2.他们需要问自己的问题是，“在我对这个故事完成有信心之前，需要检查什么？”他们想到的每个场景都会变成一个测试，在这种情况下是一个功能测试。

**客户测试**：1.功能测试不一定始终运行在100%。2.客户通常不能自己编写功能测试。3.这就是为什么任何规模的XP团队至少需要一名专职测试人员。

**其他测试**：平行测试——旨在证明新系统与旧系统完全相同的一种测试。2.压力测试——旨在模拟最糟糕的负载的一种测试。压力测试适用于性能特性不易预测的复杂系统。3.猴子测试——旨在确保系统在面对无意义输入时表现得合理的一种测试。

**设计已死**

**XP中的设计？**1.对于许多初次接触极限编程（ExtremeProgramming,XP）的人来说，XP似乎宣告了程序设计的死亡。不仅许多设计行为被嘲笑为“冗余的前期设计”（BigUpFrontDesign），甚至像UML、灵活的框架（Framework）、模式（patterns）这些设计技巧也被轻视乃至被完全忽视。2.实际上，XP中包括很多设计，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（evolutionarydesign）成为一种可行的设计策略。它还为设计人员（Designers）提供了新的挑战与技巧，让他们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**：1.演进式设计。它的本质是系统的设计随着软件开发的过程增长。设计（design）是撰写程序代码过程的一部分，随着程序代码的发展，设计也跟着调整。2.在常见的使用中，演进式设计实在是彻底的失败。设计的结果其实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**：所谓的设计（design）是要能够让你可以长期很简单地修改软件。当设计（design）不如预期时，你应该能够做有效的更改。一段时间之后，设计变得越来越糟，你也体会到这个软件混乱的程度。

**计划式设计**1.计划式设计的做法正好相反（借鉴其他工程学科）。2.在设计图中确定所有的细节，一部分使用数学分析，但大部分都是使用建筑规范。所谓的建筑规范就是根据成功的经验（有些是数学分析）制定出如何设计结构体的法则。当设计图完成，她们公司就可以将设计图交给另一个施工的公司按图施工。

**计划式设计在软件开发中的应用**：1.计划式设计将同样的方式应用在软件开发。2.Designer先定出重要的部分，程序代码不是由他们来撰写，因为软件并不是他们“建造”的，他们只负责设计。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。一旦设计的部分完成了，他们就可以将它交给另一个团队（或甚至是另一家公司）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。

**计划式设计的缺点**：1.计划式设计方法从七○年代出现。2.在很多方面它比codeandfix渐进式设计要来的好。3.缺点：第一个缺点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。可是如果designer在完成工作之后就转移到其他项目，那怎么办？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。第二：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。然而，他们忙于从事设计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变化，同时也失去了对于那些实际撰写程序代码的人的尊敬。

**建造者与设计者的关系**：建造者和设计者之间这种微妙的关系在建筑界也看得到，只是在软件界更加凸显而已。之所以会如此强烈是因为一个关键性的差异。在建筑界，设计师和工程师的技术有清楚的分野；在软件界就比较分不清楚了。任何在高度注重design的环境工作的programmer都必须具备良好的技术，他的能力足够对designer的设计提出质疑，尤其是当designer对于新的发展工具或平台越来越不熟悉的情况下。

**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。一项预留处理易变性质的设计可能对于将来的需

# 持续集成

**概述**：1.持续集成（ContinuousIntegration,CI）是一种软件开发实践，在实践中项目成员频繁地进行集成，通常每个成员每天都会做集成工作，如此，每天整个项目将会有多次集成。2.许多团队都发现这种方法大大地减少了集成问题，并且能够快速地开发出高内聚性的软件。

**单次瀑布模型与敏捷软件开发**：1.单次瀑布模型：软件集成是一个漫长并且无法预测的过程。2.敏捷软件开发：每个开发人员离共享的工程状态只有咫尺之遥，并且可以在几分钟之内将自己的代码集成进去。任何集成错误都能被快速地发现并得到快速的修正。3.“持续集成”源自于极限编程（XP），并且是XP最初的12种实践之一。4.“持续集成”服务：Jenkins（https://jenkins.io/）

**简单持续集成实践描述**：1.假设我们需要向软件添加一点功能，至于是什么样的功能并不重要，我们假定它很小并且可以在几个小时内完成。2.（1）首先我们需要在本地机器上保留一份当前已经处于集成状态的代码的拷贝。通过代码管理系统（GIT、SVN等）在代码库的主线（mainline）上拉下（checkout）一份工作代码拷贝。（2）为了完成软件的功能添加，对本地代码进行修改，其中既包括修改产品代码，也包括添加自动化测试。持续集成非常看重测试，并且在软件代码本身中达到了测试自动化——自测试代码，通常使用流行的XUnit测试框架的某个版本。

**持续集成关键实践**：1.维护一个单一的代码库2.使构建自动化3.使构建自测试4.每人每天都向代码库提交代码5.每次提交都应在集成服务器上进行构建6.快速构建7.在与生产环境相同的环境中运行测试8.使任何人都能轻易获得可执行文件9.人人都能看到正在发生什么10.自动化部署

**维护一个单一的代码库**：1.作为最基本的持续集成实践，请保证你使用一款代码管理系统。2.GIT、SVN……3.当你有了代码管理系统之后，确保每个开发者都能方便地获得到源代码。所有东西都必须在代码库里。4.原则是：在一台新机器上checkout代码后构建也能构建成功。新机器上的东西应该尽量的少，通常包括很大的，难于安装的，并且稳定的软件，比如操作系统，Java开发环境或者数据库管理系统等。

**使构建自动化**：1.将源代码变成一个能运行的软件系统通常是一个复杂的过程，包括编译，文件搬移，加载数据库模式等等。但其中大多数任务都是可以自动化的，并且也应该被自动化。让人去输入奇怪的命令或点击对话框是非常耗时的，而且从根本上来说也是个错误的做法。2.构建所需的自动化环境对于软件系统来说是一个通用功能。

**一个常见的错误**：1.在自动化构建里并没有完全包括构建所需的东西，比如构建过程中应该从代码库里取得数据库模式文件并自动执行之。2.任何人都应该能够在一台新机器上拉下代码库中的代码，并只用一个命令将系统运行起来。3.优秀的构建工具能够分析出哪些地方需要做相应的修改，并将这个分析过程本身做为整个构建过程的一部分。4.根据自己的需要（考虑到集成代价），你可以选择不同的东西进行构建。构建中既可以包括测试，也可以不包括，甚至可以包括不同的测试板块。有些组件可以进行单独构建。构建脚本应该能够允许你针对不同的情形进行不同的构建目标。

**使构建自测试**：1.传统意义上的构建包括只编译，链接等过程。此时程序也许能运行起来，但这并不意味着系统就能正确地运行。2.一种快速并高效发现bug的方法是将自动化测试包含到构建过程中。当然，测试也不见得完美，但的确能发现很多bug——足够多了。随着极限编程（XP）的流行，测试驱动开发（TDD）也使自测试代码流行起来，越来越多的人开始注意到这种技术的价值所在。3.对于自测试代码而言，你需要一组自动化测试来检测大部分代码库中的bug。测试能通过一个简单得命令来运行并且具备自检功能。测试的结果应该能指出哪些测试是失败的。对于自测试的构建来说，测试失败应导致构建失败。

**在这种频繁提交下**：1.开发者可以快速地发现自己代码与他人代码之间的冲突。快速解决问题的关键在于快速地发现问题。几个小时的提交间隔使得代码冲突也可以在几个小时内发现，此时大家的修改都不多，冲突也不大，因此解决冲突也很简单。对于好几周都发现不了的冲突，通常是很难解决的。2.基本原则是：每个开发者每天都应当向代码库进行提交。

**每次提交都应在集成服务器上进行构建**：1.在实践中，有出错的时候，原因之一在于纪律——有人并没有在提交之前进行本地更新和构建。另外，不同开发机器之间的环境不同也是一个原因。2.应该保证在集成服务器上进行构建，只有当集服务器机上构建成功后，才表明你的任务完成了。由于提交者需要对自己的提交负责，他就得盯着主线上的构建，如果失败，马上修改。3.主线构建：一是手动构建，二是使用持续集成服务器（Jenkins）。

**快速构建**：1.持续集成的关键在于快速反馈，需要长时间构建的CI是极其糟糕的。2.对于多数项目来说，将构建时间维持在10分钟之内是合理的，这也是XP的方针之一。3.引入阶段性构建---对于企业级应用来说，我们发现构建时间的瓶颈通常发生在测试上，特别是那些需要于外部交互的测试——比如数据库。

**一个简单的例子**：1.是将构建分为两个阶段，第一个阶段完成编译，并且跑那些不需要外部交互的单元测试，数据库交互也通过stub的方式完全消除掉。2.在这种情况下，通常将第一阶段视为提交构建，并将此做为主要的CI周期。第二阶段则可在有必要时才进行，如果这个阶段构建失败，它也不需要像第一阶段那样“停下全部手头的工作”，但也应该得到尽快的修改。

**在与生产环境相同的环境中运行测试**：1.测试旨在发现可能在生产环境中出现的问题，因此如果你的测试环境与生产环境不同，那么测试很有可能发现不了生产环境中的bug。2.虚拟化技术：虚拟机、Docker

**使任何人都能轻易获得可执行文件**：1.软件开发最困难的事情之一便是你不能保证所开发的是正确的软件。人们往往很难预知自己究竟想要什么，而相反，对已有的东西进行评判和修改却容易的多。2.项目中的所有成员都应能够获得最新的可执行文件并能成功的运行，目的可以包括做演示，浏览测试或者仅仅看看项目本周有何修改。3.确保一个通用的地方来存放最新

**人人都能看到正在发生什么**：1.持续集成主要在于交流，因此应当保证每人都能轻易看到当前系统的状态和已做的修改。2.CI服务器：Jenkins

**自动化部署**：1.自动化部署脚本，不仅包括测试环境的脚本，也包括针对生产环境的部署脚本。虽然我们不是每天都向生产环境部署，但自动化部署不仅可以加速部署过程，并且能够减少部署错误。2.如果你已经有了生产环境的自动化部署，那么也应该考虑一下相应的自动化回滚。由于失败是时而会发生的事情，在这种情况下，我们希望能快速回滚到失败之前的状态。3.在集群环境中，有每次只向一个节点部署的情况，由此在几个小时之内逐渐完成所有节点的部署。4.对于一些面向公众的Web应用，我所了解的另外一种很有趣的部署方式是，先试验性针对一部分用户进行部署，再通过这些用户的试用情况来决定是否向所有用户部署。（灰度发布、A/B测试）

**持续集成的好处**：1.降低风险！2.已经处于项目的末期，但是仍然不知道何时才能结束。3.延期集成的缺点在于，很难预测集成到底要花多少时间，更糟的是，你很难了解集成的进展情况。4.持续集成正好解决了这些问题。每次集成的时间都不长，任何时候你都知道自己所处的情况，软件的哪些地方在工作，哪些没有。

**Bug**：1.持续集成并不能消除bug，却能帮你快速的发现bug并予以清除。Bug也存在积累性，bug越多，越难清除。部分原因在于bug之间存在牵连。另外也存在心理因素，bug一多，人便没那么多精力去修了——这就是所谓的“BrokenWindows综合征”。2.持续部署：有了持续集成，频繁部署也不是什么难事了。频繁部署的价值在于，你的客户可以快速的享用软件的新功能，并能快速的提出反馈。这将有利于清除客户和开发之间的障碍——我认为这是软件开发最大的障碍。

**引入持续集成**（1）第一步需要将构建自动化，并将你所需的所有东西都放在代码管理系统中，可以通过一个命令来构建整个系统。（2）在构建中引入一些自动化测试，试着确定出现问题的主要范围，并用自动化测试去发现这些问题。（3）使提交构建快速完成。（4）对于新项目，从项目开始就采用持续集成。（5）寻找帮助，找有经验的人帮助你。

# 看板

**Work in Progress(WIP)**

1.在制品（WIP）是同时进行中的工作数量，减少在制品使其快速流过整个工作流，即前置时间缩短。2.我们不再关注是否每个工人的工作都是最有效率的；3.在最后一轮中，前置时间是最短的，但每个工人的工作时间都延长了，从个人来看工作效率下降了，但整个团队的效率最高。

**限制在制品**1：致力于减少同时处理的工作项2.批量规模越小，前置时间越短3.流动效率提升的同时资源效率会有所降低4.立即实施:停止立项并开始完成5.限制在制品将使改进机会浮出水面着手改进后会获得更快的流动6.不要企图找到一个唯一正确的数字作为团队的在制品限制规模.7在制品限制不是仅为了设立规则，而是为了触发讨论

**看板不是？**1.看板并不是一种软件开发生命周期的方法学，也不是一种项目管理的方法。

2.实施看板时，需要当前已经有一些在运行的过程，这样便可应用看板来逐步改变当前运行的过程。

**Kanban与 Scrum 的相似性：·**都是既精益又敏捷、都是拉动式计划、都限制了 WIP。·Scrum 的 WIP 按单位时间限制；看板的 WIP 按流程状态限制。·都以透明的方式驱动过程改进。·都关注于尽早交付、频繁交付可发布的软件。·根基都是自组织型团队。·都需要把工作拆分。·发布计划都是根据经验数据（生产率/生产周期）不断优化的。

**看板与Scrum的差异：1.**Scrum规定了固定时长的迭代。看板固定时长的迭代是可选的。计划、发布、过程改进等活动可以各有各的节奏。它可以由事件驱动，不用非要固定时长。2.Scrum团队承诺当前迭代做完一定量的工作。看板承诺是可选的。3.Scrum用生产率作为计划和过程改进的默认度量手段。看板用生产周期作为计划和过程改进的默认度量手段。4.Scrum规定了跨功能团队。看板跨功能团队是可选的。可以有专职团队。5.Scrum任务必须分解，以便在 1 个 Sprint 里面能做完。看板没规定任务规模。6.Scrum规定了燃尽图。看板没规定专门的图表形式。7.Scrum间接限制（每个 Sprint 的）WIP。看板直接限制（每个工作流状态的）WIP。8.Scrum规定了估算。看板估算是可选的。9.Scrum不能往进行中的 Sprint 里面加任务。看板只要有人手富余就可以加任务。10.Scrum一个 Sprint Backlog 归一个团队所有。看板一张看板图可以由多个团队或多人共用。11.Scrum规定了三种角色（PO、SM、Team）看板没有规定任何角色。12.Scrum每个 Sprint 之间重置 Scrum 板。看板看板图一直保留着。13.Scrum规定了经过优先级排序的产品 backlog。看板优先级排序是可选的。

**敏捷：**

**为什么敏捷与精益出现在软件开发行业？**1.软件开发本质属性：复杂性、一致性、可变性、不可见性。2.敏捷与精益本质上是帮助我们处理软件开发的复杂性、可变性。敏捷的本质，是承认软件开发的复杂性。而且承认，这种复杂性，达到了这样一种程度：“无法通过足够充分的前期准备，而消除后续的风险。甚至于，前期准备得越是充分，后续的风险越大。”敏捷软件开发是当前应对模糊需求、快速变化需求的最佳方式。

**软件项目成功？**传统观点1.成功的（Successful）:按时完成，费用不超过预算，而且所有特性和功能都符合原先的设计规格。2.不太成功的（Challenged）:已完成并且可以运行，但费用超出了预算，没有如期完成，拥有的特性和功能少于原先的设计规格。3.失败的（Impaired）:在开发周期的某个时刻被取消了。

**软件项目成功？**敏捷观点1.为客户创造价值是评价成功的最重要标准。2.A:110万收益，100万成本，延期后果严重（普通外包项目）3.B:5000万收益，100万成本，延期后果可以接受（GoogleEarth,Gmail,Instagram）4.A项目控制很重要（否则会亏损）；B项目控制没有那么重要。5.所有软件开发实践都应该以提升项目收益为首要目标！

**如何做到敏捷:**1.敏捷并不仅仅是一种可以遵循的具体过程。2.一种软件开发哲学、一组实践、一套互补的原则和一个社区。（KentBeck描述XP）3.敏捷软件开发是一种思考软件开发的方式：敏捷宣言和敏捷原则

**差异？**1.A知道的技术比我多，并且我们共同知道的技术他比我更擅长。2.技术是重要的，如果不掘土、种植，你就不是在做园艺。3.实践。是你天天做的事情。4.对实践进行详细描述是有用的，它们是明确和客观的。比如在编码前写测试代码，即使你不懂为什么，这种实践仍然是有价值的。

**只懂实践？**1.即使我懂了A所知晓的所有园艺技术，我还不能被称为一个园艺师。2.园艺方面什么是好的，什么是坏的，A有高度发达的判断力。3.我可能自豪于正确剪枝的能力，A却可能认识到整棵树都应该除掉。4.A对花园中有用的东西有个整体的概念，而我没有。

**价值观**:1.价值观是知识和理解的另外一个层次，价值观是在某种处境中我们喜欢或不喜欢某事情的根源。2.沟通,简单,反馈,勇气,尊重（XP价值观）

**价值观与实践**:1.没有价值观，实践很快会变成生搬硬套，缺乏目的或方向。2.价值观和实践相结合程序员才能高效的执行实践。3.实践是价值观的表现。4.价值观是在一个高层次上的表达，可以以价值观的名义做任何事。5.“因为我重视沟通，所以写了这份1000页的文档”，也许是这回事，也许不是。如果每天15分钟的交谈更有效的话，文档就不能说明我重视沟通。6.实践让价值观清晰可见。7.实践是清晰明了的。每个人都知道我是否参加了早上的站立式会议，但我是否重视沟通并不是那么明确和具体的。

**原则**:1.在价值观和实践之间架起桥梁的是原则。2.原则是生活中具体领域的指导方针。

**敏捷宣言**：1.个体和互动高于流程和工具，敏捷力的基本宗旨之一就是，干活的人最清楚该如何完成工作。2.工作的软件高于详尽的文档。3.客户合作高于合同谈判，敏捷价值观着重强调，开发团队和客户之间要保持尽可能公开和顺畅的对话。4.响应变化高于遵循计划，计划驱动型组织通常都有“变化控制”流程。

# 新方法学

**敏捷方法和工程方法的区别**：表面区别：敏捷型与工程型方法有一些显著的区别。其中一个显而易见的不同反映在文档上。敏捷型不太依赖文档，对于一项任务，通常只要求尽可能少的文档，更多是“面向源码”（code-oriented）。最根本的文档应该是源码。本质区别：文档减少只是表象，反映了两个更深层的特点：敏捷型方法是“适应性”而非“预见性”。工程方法试图对软件开发项目进行长期详细规划，而敏捷型方法则欢迎变化，并能适应变化，甚至调整自身来应对变化。敏捷型方法是“面向人”的（people-oriented），而非“面向过程”的（process-oriented）。工程方法侧重定义一个通用过程，敏捷型则认为过程是辅助，开发人员的技能才是关键。

**将设计和建造分离开来**：设计与建造是两类非常不同的活动。设计难以预见，需要创造性人员，而建造则相对容易预设。设计完成后，建造可以有明确的计划，并具备较高的可预见性。在软件开发中，传统工程方法追求可预见的生产进度计划，依赖低技能人员进行建造。然而，软件设计的复杂性使得传统工程方法的预见性不适用。使用像UML这样的工具进行设计可能导致在编码过程中出现问题，因为设计与实际编程之间往往存在缺陷。此外，软件开发的设计占比远大于建造。在土木工程中，设计费用占比通常为10%，建造费用占比为90%，但在软件开发中，编码和测试仅占15%，设计则占到50%。这表明软件设计的复杂性和不可预见性。软件开发中的所有工作都属于设计，需要创造性人员，而具体的建造工作（如编码）所占时间较少。因此，传统的工程方法不适用于软件开发，需要不同的过程和方法来应对这种创造性的设计活动。

**需求的不可预见性**：在软件开发中，需求变更是常态，问题在于如何处理它。一种方法是将需求变更归因于需求工程（requirementsengineering）没有做好，通常需求工程是要在开发前全面理解系统需求并取得客户认可。准确获取所有需求非常困难，特别是当开发商无法提供某些需求的费用信息时。软件开发的费用估算不容易，因为它是一种设计活动，难以精确计划，而且系统的基本材料变化非常快，开发活动高度依赖人员，且个体难以预测和量化。软件的“不可触摸”性也是一个原因，功能的具体价值往往只有在实际使用系统时才能判断。需求不仅是可变的，简直是应该变的，尤其当客户参与开发并知道软件容易修改时。即便固定下所有需求，软件开发仍可能面临问题。商业环境的快速变化意味着某些功能的价值可能会下降，因此，需求的不可预测性使得软件开发无法制定可预见性计划。

**预见性是不可能的吗？**一般来说，预见性在软件开发中是不可能的，除非是像航天飞机软件开发这样的大型项目，要求大量会议、充足的时间和稳定的需求。然而，对于一般商用软件开发，预见性并不适用，因此需要不同的开发过程。如果你强行使用预见性方法而无法满足其条件，这是非常危险的。许多方法学者希望其方法适用于所有情况，但实际上，许多方法只在特定条件下有效，错误地使用它们会导致问题。使用预见性方法具有诱惑力，因为预见性是非常重要的。然而，当无法实现预见性时，计划的偏离可能会导致计划崩溃，这往往会带来痛苦的后果。因此，在不可预见性的环境中，应避免使用预见性方法。认识到这一点是一个重要的转折，它意味着我们需要寻找适应性过程来应对不可预见性，而不是依赖传统的预见性方法。

**不可预见过程的控制－迭代：**为了应对不可预测的世界，最重要的是随时了解开发进程，这需要一个诚实的反馈机制。

迭代式开发方法是这一机制的关键。这种方法的要点是不断地生产出系统的工作版本，每个版本逐步实现所需功能，并经过整合和测试。

迭代式开发的理由是，集成和测试过的系统能提供最真实的反馈，问题会在实际使用中暴露出来。虽然迭代式开发也可用于可预见性环境，但它主要用于适应性过程，及时应对需求变更。需求变更使得长期计划不稳定，迭代式开发提供短期的计划基础，通常是“迭代周期”。不同方法对周期时长有不同建议，一般倾向于尽可能短，以获得频繁的反馈，了解开发进展。

**适应性的客户**：适应性过程需要与客户建立新型关系。大多数客户倾向于固定价格合同，但这种合同要求稳定需求，与适应性过程的特点不符。固定价格模式与适应性过程结合会导致问题，客户和开发者都会受到伤害。这并不意味着不能为软件固定预算，而是不能固定时间、价格和范围。敏捷方法通常固定时间和价格，允许范围变化。适应性过程让客户能够深入控制开发过程，在每个迭代阶段检查进度并调整方向。这种密切的合作关系是适应性过程成功的基础。这种开发方式对客户有诸多益处，特别是“回应性”强，系统能尽早投入使用，客户可以根据实际情况和变更需求要求修改功能。MaryPoppendieck总结道“后期的需求变化是个很大的优势”。在开发过程中，客户逐步明确哪些功能重要，敏捷方法通过鼓励业务人员不断梳理需求，尽快整合变化。成功项目的衡量标准对预见性项目来说是按计划执行，而对敏捷型项目来说是商业价值—客户得到的软件是否大于投入。成功的敏捷项目可能与初期计划不同，但提供更高价值的软件。

**把人放在第一位**：实施一个适应性过程并不容易，特别是它要求一组高效的开发人员。高效既体现在高素质的个体，也体现在有能让团队协调一致的工作方式

**可兼容性程序插件：**传统方法的目标之一是将项目参与人员视为可替代的部件，强调角色而非个体。这种观点假设，个体的不同并不重要，只有资源的数量影响项目计划。然而，AlistairCockburn明确反对这种观点，他认为人是软件开发中最重要的因素。传统方法将人视为“部件”是错误的，因为人是高度可变和非线性的，每个人有其独特的成功或失败模式。如果方法未能充分考虑这些因素，项目很容易偏离轨道。Cockburn提倡以人为中心的开发理念，这已成为软件行业的共识。问题在于，许多方法与这一理念背道而驰，导致团队士气低落，优秀人才流失，最终可能只得到“可互替的编程插件”。作出“以人为优先”的决策需要决心，这与泰勒的“科学管理”方法相反。尽管泰勒主义适用于管理工厂，但对于创意性和专业性强的软件开发，它并不适用。

**程序员是负责任的专业人员：**泰勒主义认为，干活的人并非最懂如何做好工作的人员，这在工厂中可能成立，但在软件开发中并不适用。历史证明，软件行业吸引了大量优秀人才，这些人才既具有高水平的能力，也得到丰厚回报。要聘到并留住这些人才，必须认识到他们是专业人员，最有资格决定如何做好自己的工作。泰勒主义的做法只有在计划者比实际操作人员更懂得如何执行时才有效，但在软件开发中，程序员往往是最了解技术细节的人。

**面向人的过程的管理：**实施敏捷型过程的关键是让团队接受而非强加一个过程。通常软件开发过程由管理人员决定，可能会遭到开发人员的抵制，尤其是当管理人员与实际开发活动脱离很久时。开发人员应有权做出所有技术决策，XP强调这一点，特别是在前期计划阶段，开发人员应该估算完成一项工作的时间。对于许多管理人员来说，这种形式的技术领导是一种极大的转变，它要求开发人员和管理人员在项目领导方面具有同等的地位。管理人员仍需发挥自己的角色，但必须认识并尊重开发人员的专业知识。IT行业的技术变化迅速，新技术可能几年后就过时，因此管理层应信任和依靠当前的开发人员。

**度量的困难性：**在“科学管理”中，存在强烈的驱动力发展客观的度量方法来衡量员工的工作效率。度量软件工作是非常困难的，尽管人们已做出很大努力，但我们仍无法有效度量一些简单的方面，如生产率。没有有效的度量方法，外部控制就会变得困难。没有有效的度量方法而强行引入度量，将导致管理出现问题。RobertAustin指出，进行度量时必须考虑所有影响度量的因素，缺失的因素会导致工作方式的改变，从而影响工作效果。基于度量的管理方法因此存在“失效”问题。Austin的结论是，你必须选择两种方法之一：基于度量的管理或委托式管理。基于度量的管理适用于简单、重复性工作，而软件开发则完全不同，复杂且难以度量。因此，敏捷开发者认为，传统的度量管理方式在软件开发中会导致“失效”，而委托式管理更加有效，这也是敏捷论者的核心观点。

**业务专家的引领作用：**技术人员需要应用系统的需求引导，这就要求他们与业务专家保持紧密联系。敏捷过程的一个重要方面是，开发人员和业务专家之间的联系需要比一般项目更为紧密。如果两者之间的沟通仅是偶尔进行，敏捷型过程就无法顺利进行。这是因为敏捷开发依赖于快速变化的环境，频繁的沟通和联系是确保每个成员能够及时了解变化的必要条件。

**自适应过程：**自适应过程指的是随着时间推移，开发团队会根据实际情况不断修改和改进过程。项目开始时使用的适应性过程会随着项目进展而变化，团队会发现哪些方式最有效，并调过程以适应团队需求。改善自适应过程通过思考哪些部分做得好，哪些有待改进，团队可以在下一次迭代中调整过程，逐步完善，以更好地适应开发需求。

**敏捷开发的不同风格：**"敏捷"是指软件开发的一种理念，涵盖了多种方法，如极限编程（XP）、Scrum、精益开发（LeanDevelopment）等。每种方法有自己独特的思路、社群和领军人物。尽管如此，所有敏捷方法都遵循一些共同的原则，并且各社群之间常常借鉴彼此的思想与技术。

**你是否应走向敏捷？**引入敏捷方法可以带来比边写边改（codeandfix）更有效的开发方式。敏捷的主要优点是步骤较少，适合已经习惯无过程的团队。找到合适的项目来试验敏捷方法。由于敏捷方法“面向人”，团队成员必须愿意使用敏捷方法。强行推行敏捷与其根本思想背道而驰。

从一个便于管理的小系统开始，适合初次试验敏捷方法。尽管敏捷方法也适用于大项目，但从小项目入手更易于管理。选择对业务影响小的项目开始。

什么情况下不应使用敏捷方法？如果团队成员不愿意进行密切合作，强行推行敏捷方法会困难重重。因此，不能强迫团队使用敏捷。

# 小测

# 001 - 没有银弹

2.在《没有银弹》中，Brooks主张：存在能够显著提高软件生产力的技术。；软件开发的主要困难是复杂性、 一致性、变化性和不可见性。 √；通过采用新的编程语言，软件危机可以完全得到解决。；大多数的软件开发问题是由于使用了错误的工具。

3.《没有银弹》文章中提到的“银弹”是指：一种新的编程语言。；一个具有魔法般效果的解决方案。√；一种新的软件开发方法。；一个新的软件工具。

4.根据《没有银弹》的观点，下列哪个因素被认为不是造成软件危机的直接原因：软件开发周期的增长；软件的复杂性和变化性；新技术的快速迭代 √；人力成本的提高

5.Brooks提到，即使解决了所有的偶然困难，软件的什么困难仍然会存在：软件的复杂性 √；软件的可靠性；软件的维护性；软件的开发速度

6.在讨论软件的本质复杂性时，Brooks强调了软件系统的哪种属性：其可以被多次复制而不产生额外成本；其通常与硬件紧密结合；其设计不能简化，而只能被概念化 √；其经常需要进行大量的维护和修改

7.你的团队在开发一个大型的在线银行系统。这个系统需要处理数百万用户的交易和查询，同时保证数据的安全 性、准确性和实时性。为了应对这一挑战，你的团队已经采用了最新的编程语言、最先进的数据库技术和高效 的算法。但即使如此，开发进度仍然迟缓，而且出现了很多不可预见的问题。请问，根据Brooks的《没有银弹》,这种情况的主要原因可能是什么?：因为团队采用了错误的技术和工具。；大型软件项目固有的本质复杂性使得问题的解决和管理变得困难。 √；团队中的开发者缺乏必要的技能和经验。；在线银行系统的用户需求太多，使得开发困难

8. 您是一家软件公司的CTO,正在考虑采用新技术来改进软件开发过程。您的团队成员将Brooks的《没有银弹》提 出的观点引入了讨论。您的团队成员表示，他们认为通过引入最新的编程语言可以显著提高生产力。根据

Brooks的论文，以下哪项最能反映他的观点：新技术将使软件开发速度提高一个数量级。；尽管新技术可能提供一些提高，但不会根本性地解决软件复杂性问题。 √；新技术永远不会带来任何实质性的进步。；新技术总是值得追求的。

9.数字创新公司正在开发一个全新的电商平台。由于市场的快速变化和用户需求的多样性，该平台需要频繁地进行迭代和更新。公司的开发团队为了满足这些变化，决定采用一种被广泛推崇的新编程语言，希望它能简化代 码的修改和扩展。然而，尽管新语言确实为团队提供了一些便利，但在应对频繁变化的需求时，团队仍然面临 着巨大的挑战。请问，根据Brooks的《没有银弹》,这种情况的最可能的解释是什么：新编程语言并不适合开发电商平台。；开发团队没有充分利用新编程语言的特性。；软件的可变性是其本质特性之一，即使采用新的编程语言，也无法完全消除与之相关的复杂性。 √；团队中的开发者对新编程语言的掌握程度不足。

10.星辰科技公司正在开发一款复杂的数据分析软件，旨在帮助商业用户更好地理解他们的销售数据。为了确保产 品的质量，公司进行了多轮的内部审查。然而，在审查过程中，许多决策者发现很难理解软件的内部逻辑和数据处理机制，尽管开发团队提供了详细的文档和解释。请问，根据Brooks的《没有银弹》,这种情况最可能的原因是什么：开发团队的文档撰写技能不佳，导致解释不够清晰。；决策者们缺乏足够的技术背景，因此难以理解软件细节。软件本身的不可见性导致其结构、行为和复杂性难以通过传统的方式(如文档或图表)完全传达。√；数据分析软件本身太过复杂，即使是专家也难以完全理解。

11.网络魔法公司正在开发一款云端的软件开发平台。为了简化代码编写过程，他们决定集成一种基于大语言模型(类似ChatGPT) 的代码生成工具，允许开发者通过自然语言描述需求来自动生成代码。初步测试表明，该工具 可以准确地生成简单的代码片段，节省了开发时间。然而，在项目实施中，开发团队发现大型项目中仍然需要 大量的手工编码，并且需要经常调整和优化由工具生成的代码。这种情况的最可能的解释是什么：ChatGPT 的技术还不够成熟，不能满足实际开发需求。；软件工程的本质复杂性决定了无法通过自动化工具完全替代人的创造性和决策。 √；网络魔法公司的开发团队没有充分利用工具的特性。；自动化代码生成工具在定义需求时需要更严格的自然语言描述。

# 002 - 大教堂与市集

2.《大教堂与市集》中，哪种开发模式是基于严格的集中控制和封闭性的：大教堂式 √；市集式；两者都是；两者都不是

3.在市集式开发模式中，哪个因素被认为是软件质量高和错误迅速被检测出的主要原因：有严格的代码审查机制；开发者数量众多；“多眼睛原则“或“给足够多的眼睛看，所有的问题都显而易见” √；开发者通常是专业的

4.为什么开源软件往往被认为能够更快地适应变化和需求:；因为它们有大量的财务支持；因为它们的开发者更有经验；因为开源社区可以迅速吸纳反馈并进行修改 √；通常使用更高效的编程语言

5.《大教堂与市集》中提到，开源软件的一个主要优势是什么: 免费；无需进行严格的质量检查；能够更好地进行商业化利用；社区驱动，能够吸纳大量的用户反馈和贡献 √

6.在《大教堂与市集》中，以下哪个观点与市集式开发不符:更快速的迭代会导致更好的软件；软件的复杂性应当被一个核心团队管理；用户是最好的测试员；任何问题在发布前都应被固定的开发团队解决 √

7.根据《大教堂与市集》,开源的成功与否与哪个因素最不相关:一个活跃的贡献者社区；软件的原始设计质量；项目的广告和推广策略 √；早期和频繁的发布

8.在市集模式中，为什么认为“失败是成功之母”：因为失败会增加开发者的经验；因为市集模式鼓励快速失败以尽早找到错误 √；因为所有的软件都是完美的，不会失败；因为市集模式依赖于高质量的代码审查

9.根据《大教堂与市集》,哪种策略不太可能增强一个开源项目的生态系统:提高代码的可读性；限制外部贡献者的访问 √；鼓励社区的反馈；早期和频繁的发布

10.在《大教堂与市集》中，哪项不是大教堂模式的典型特征:长时间的发布周期；高度的结构和组织；社区驱动的开发 √；由一个核心团队完全控制

11.公司XYZ 正在决策关于他们新软件项目的开发方法。他们正在考虑“大教堂”式的开发(集中、封闭、由一个固定 的团队来开发)和“市集”式的开发(开放、协作、来自社区的贡献)。根据《大教堂与市集》的描述，哪种开发 方法更可能鼓励来自全球的开发者参与：大教堂式；市集式 √；两者都不鼓励；两者都鼓励

# 003-CharacterizingtheSoftwareProcess

2.SEI的成熟度框架如何帮助不同的软件开发组织之间进行比较：通过提供一个统一的工资和福利标准；通过为每个组织分配一个绩效排名；通过定义一个共同的过程成熟度量级√；通过制定统一的软件开发技术标准

3.Humphrey描述了一个软件过程成熟度的框架，该框架用于评估和改进软件开发过程。在这个框架中，哪一级强调了对项目管理和过程控制的正式化?：初始级别；已定义级别√；已管理级别

4.根据Humphrey关于软件过程改进的观点，哪种方法能最有效地启动过程改进活动?：从引入新的开发工具开始；通过管理层的强制命令；通过底层员工的草根努力；通过全面评估当前过程并确定改进点√

# 004-Scrum

2.Scrum中，产品负责人(ProductOwner)的主要责任是什么：编写代码；测试软件；最大化产品价值 √ ；管理团队成员

3.Scrum 框架中的三个主要角色是什么?：产品负责人、Scrum Master、客户；产品负责人、Scrum Master、团队 √ ；设计师、分析师、编程员

4.在Scrum中，一个Sprint通常持续多长时间?：1-2天；1-4周 √ ；3-6个月；1年

5.Scrum框架中，哪个角色负责移除开发过程中的障碍：产品负责人；ScrumMaster √ ；团队成员；客户

6.Scrum中的“猪”和“鸡”的比喻指的是什么：猪是项目的负责人，而鸡是团队成员；猪是团队成员，而鸡是不参与项目决策的人 √ ；猪指的是客户，而鸡指的是开发者；猪是Scrum Master,而鸡是产品负责人

7.用户故事在Scrum中的作用是什么：定义项目的最终目标；提供详细的项目需求说明；作为产品功能的简要描述和协作讨论的基础 √ ；设计UI/UX原型

8.在 Scrum 中，什么是Sprint回顾会议的主要目的：审查团队成员的工作表现；制定下一个Sprint的计划；讨论并改进团队的工作方式 √ ；展示给客户最终产品

9.Scrum中的每日站会(DailyScrum)的目的是什么：为了报告给管理层；讨论技术问题和解决方案；快速同步团队进展和识别障碍 √ ；审批团队的工作时间表

10.在一个软件开发项目中，团队正在使用Scrum 框架。产品负责人提出了以下用户故事：“作为一个手机应用 用户，我希望应用能够根据我的位置显示附近的餐馆，以便我可以快速找到就餐地点。”根据3C原 则(Card,Conversation,Confirmation), 哪个选项最准确地描述了对这个用户故事的处理：Card:在卡片上写下用户故事，团队成员不需要进行进一步的沟通，直接开始开发；Conversation:团队成员与产品负责人进行简短会谈，确认功能的技术细节，然后开始编码；Confirmation:团队确定了一个清晰的验收标准，即用户能够看到他们周围一英里内的所有餐馆；Card,Conversation,Confirmation:首先将用户故事简明地记录在卡片上。然后，团队与产品负责人进行深入的讨论，以理解和细 化需求。最后，共同定义清晰的验收标准，比如用户能够看到周围特定距离内的餐馆，且应用能够根据不同位置更新这些信息。 √

11.在Scrum框架中，一个团队正在进行UserStoryMapping以规划他们的产品开发。关于UserStoryMapping的正确描述是哪一项：User Story Mapping主要用于跟踪项目的进度，确保按时完成每个Sprint的目标。；User Story Mapping是一个动态过程，主要关注于定义和优先排序技术任务和bug 修复。；在 User Story Mapping中，团队将用户故事按照功能的技术复杂度排序，以优化开发流程。；User Story Mapping涉及创建一个可视化地图，展示用户故事按照它们在用户需求中不同抽象层级的排列，帮助团队理解产品如何 满足用户需求。 √

12.背景：假设你是一个使用Scrum框架的软件开发团队的成员。团队正在准备进行Sprint计划会议，需要对几个新的用户故事进行估算。团队决定使用扑克牌估算法来进行。用户故事：1.“作为一个社交媒体应用的用户，我希望能够通过我的个人资料页直接上传和编辑我的头像。” 2. “作为网站的管理员，我需要一个报告系统、以便追踪和分析用户的活动数据。”在使用扑克牌估算法进行估算时，团队成员的估算结果分别为：3,5,8,13,3,5。接下来团队应如何处理这 种估算结果：接受最高的估算值(13),因为它考虑到了潜在的风险和不确定性。；选择中间值(5),作为折中的估算结果。；组织一轮讨论，让给出高估算和低估算的成员解释他们的理由，然后进行第二轮估算。 √ ；忽略最高和最低的估算值，只考虑其它估算值。

# 005-eXtremeProgramming

2.XP开发过程中的“开发阶段”(DevelopmentEpisode)指的是什么：系统集成；用户管理；程序员实现工程任务并与系统其他部分集成 √ ；对软件进行测试

3.在XP的“开发阶段”,以下哪项是必须的：独立编程；结对编程 √ ；每小时完成一次编码；每天更换编程伙伴

4.XP开发中测试的作用是什么：仅用于最终验收；驱动开发 √ ；评估性能；文档记录

5.在XP开发中，关于变更成本的假设是什么：变更成本随时间呈线性增长 √ ；变更成本随时间呈指数增长；变更成本随时间先增后减；变更成本保持不变

6.在XP中，如果变更成本较低，应该如何处理决策：尽早做出重大决策；延迟做出重大决策 √ ；避免任何决策；随时做出决策

7.XP实践中强调的四个核心价值观包括以下哪些：效率、透明性、创新、尊重；沟通、简单性、反馈、勇气 √ ；速度、质量、灵活性、团队合作；规划、执行、监控、调整

8.XP实践中,哪项技术被认为是降低变更成本的关键；数据库优化；云计算；大模型技术；面向对象技术 √