**实验四 XP开发方法，DevOps，活动图**

实验目的：

1. 了解XP开发方法
2. 了解DevOps
3. 理解项目活动图

实验内容：

1. 阅读XP开发方法文档，理解XP过程工作模型
2. 阅读DevOps文档，了解DevOps
3. 活动图练习

（预习）书上练习题2,3（p97-98) 的软件开发项目活动图，找出关键路径。

小组讨论，针对自己项目中的工作进行工作活动分解，分工进行各自合理的工作进度估算，最后汇总绘出项目活动图，找出关键路径。

**XP开发方法**

极限编程（ExtremeProgramming，简称XP）是由KentBeck在1996年提出的。极限编程是一个轻量级的、灵巧的软件开发方法；同时它也是一个非常严谨和周密的方法。

XP是一种近螺旋式的开发方法，它将复杂的开发过程分解为一个个相对比较简单的小周期；通过积极的交流、反馈以及其它一系列的方法，开发人员和客户可以非常清楚开发进度、变化、待解决的问题和潜在的困难等，并根据实际情况及时地调整开发过程。

“Extreme”（极限）是指，对比传统的项目开发方式，XP强调把它列出的每个方法和思想做到极限、做到最好；其它所不提倡的，XP则一概忽略（如开发前期的整体设计等）。一个严格实施XP的项目，其开发过程应该是平稳的、高效的和快速的，能够做到一周40小时工作制而不拖延项目进度。

XP的核心是其总结的沟通（Communication）、简单（Simplicity）、反馈（Feedback）、勇气（Courage）四大价值观，它们是XP的基础，也是XP的灵魂。  
此外还扩展了第五个价值观：谦逊（Modesty）。 XP用“沟通、简单、反馈、勇气和谦逊”来减轻开发压力和包袱；XP精神可以启发我们如何学习和对待快速变化、多样的开发技术。成功学习XP的关键，是用“沟通、简单、反馈、勇气和谦逊”的态度来对待XP；轻松愉快地来感受XP的实践思想；自己认真实践后，通过对真实反馈的分析，来决定XP对自己的价值；有勇气接受它，或改进它。

**价值观**

**1. 沟通**

通常程序员给人留下的印象就是“内向、不善言谈”，然后项目中的许多问题就出在这些缺乏沟通的开发人员身上。经常由于某个程序员做出了一个设计决定，但是却不能及时地通知大家，结果使得大家在协作与配合上出现了很多的麻烦，而在传统的方法论中，并不在意这种口头沟通不畅的问题，而是希望借助于完善的流程和面面俱到的文档、报表、计划来替代，但是这同时又引入了效率不高的新问题。

XP方法论认为，如果小组成员之间无法做到持续的、无间断的交流，那么协作就无从谈起，从这个角度能够发现，通过文档、报表等人工制品进行交流面临巨大的局限性。因此，XP组合了诸如对编程这样的最佳实践，鼓励大家进行口头交流，通过交流解决问题，提高效率。

**2. 简单**

XP方法论提倡在工作中秉承“够用就好”的思路，也就是尽量地简单化，只要今天够用就行，不考虑明天会发现的新问题。这一点看上去十分容易，但是要真正做到保持简单的工作其实很难的。因为在传统的开发方法中，都要求大家对未来做一些预先规划，以便对今后可能发生的变化预留一些扩展的空间。

正如对传统开发方法的认识一样，许多开发人员也会质疑XP，保持系统的扩展性很重要，如果都保持简单，那么如何使得系统能够有良好的扩展性呢？其实不然，保持简单的理由有两个：

* 开发小组在开发时所做的规划，并无法保证其符合客户需要的，因此做的大部分工作都将落空，使得开发过程中重复的、没有必要的工作增加，导致整体效率降低。
* 另外，在XP中提倡时刻对代码进行重构，一直保持其良好的结构与可扩展性。也就是说，可扩展性和为明天设计并不是同一个概念，XP是反对为明天考虑而工作，并不是说代码要失去扩展性

而且简单和沟通之间还有一种相对微妙的相互支持关系。当一个团队之间，沟通的越多，那么就越容易明白哪些工作需要做，哪些工作不需要做。另一方面，系统越简单，需要沟通的内容也就越少，沟通也将更加全面。

**3. 反馈**

是什么原因使得我们的客户、管理层这么不理解开发团队？为什么客户、管理层总是喜欢给我们一个死亡之旅？究其症结，就是开发的过程中缺乏必要的反馈。在许许多多项目中，当开发团队经历过了需求分析阶段之后，在相当长的一段时间内，是没有任何反馈信息的。整个开发过程对于客户和管理层而言就像一个黑盒子，进度完全是可见的。

而且在项目的过程中，这样的现象不仅出现在开发团队与客户、管理层之间，还包括在开发团队内部。这一切问题都需要我们更加注重反馈。，反馈对于任何软件项目的成功都是至关重要的，而在XP方法论中则更进一步，通过持续、明确的反馈来暴露软件状态的问题。具体而言就是：

* 在开发团队内部，通过提前编写单元测试代码，时时反馈代码的问题与进展。
* 在开发过程中，还应该加强集成工作，做到持续集成，使得每一次增量都是一个可执行的工作版本，也就是逐渐是软件长大，整个过程中，应该通过向客户和管理层演示这些可运行的版本，以便及早地反馈，及早地发现问题。

同时，我们也会发现反馈与沟通也有着良好的配合，及时和良好的反馈有助于沟通。而简单的系统更有利于测试和反馈。

**4. 勇气**

在应用XP方法论时，我们每时每刻都在应对变化：由于沟通良好，因此会有更多需求变更的机会；由于时刻保持系统的简单，因此新的变化会带来一些重新开发的需要；由于反馈及时，因此会有更多中间打断你的思路的新需求。

总之这一切，使得你立刻处于变化之中，因此这时就需要你有勇气来面对快速开发，面对可能的重新开发。也许你会觉得，为什么要让我们的开发变得如此零乱，但是其实这些变化若你不让它早暴露，那么它就会迟一些出现，并不会因此消亡，因此，XP方法论让它们早出现、早解决，是实现“小步快走”开发节奏的好办法。

也就是XP方法论要求开发人员穿上强大、自动测试的盔甲，勇往直前，在重构、编码规范的支持下，有目的地快速开发。

勇气可以来源于沟通，因为它使得高风险、高回报的试验成为可能；勇气可以来源于简单，因为面对简单的系统，更容易鼓起勇气；勇气可以来源于反馈，因为你可以及时获得每一步前进的状态（自动测试），会使得你更勇于重构代码。

**5. 四大价值观之外**

在这四大价值观之下，隐藏着一个更深刻的东西，那就是尊重。因为这一切都建立在团队成员之间的相互关心、相互理解的基础之上。

**原则**

**1. 快速反馈**

及时地、快速地获取反馈，并将所学到的知识尽快地投入到系统中去。也就是指开发人员应该通过较短的反馈循环迅速地了解现在的产品是否满足了客户的需求。这也是对反馈这一价值观的进一步补充。

**2. 简单性假设**

类似地，简单性假设原则是对简单这一价值观的进一步补充。这一原则要求开发人员将每个问题都看得十分容易解决，也就是说只为本次迭代考虑，不去想未来可能需要什么，相信具有将来必要时增加系统复杂性的能力，也就是号召大家出色地完成今天的任务。

**3. 逐步修改**

就像开车打方向盘一样，不要一次做出很大的改变，那样将会使得可控性变差，更适合的方法是进行微调。而在软件开发中，这样的道理同样适用，任何问题都应该通过一系列能够带来差异的微小改动来解决。

**4. 提倡更改**

在软件开发过程中，最好的办法是在解决最重要问题时，保留最多选项的那个。也就是说，尽量为下一次修改做好准备。

**5. 优质工作**

在实践中，经常看到许多开发人员喜欢将一些细小的问题留待后面解决。例如，界面的按钮有一些不平整，由于不影响使用就先不管；某一两个成员函数暂时没用就不先写等。这就是一种工作拖泥带水的现象，这样的坏习惯一旦养成，必然使得代码质量大打折扣。

而在XP方法论中，贯彻的是“小步快走”的开发原则，因此工作质量决不可打折扣，通常采用测试先行的编码方式来提供支持。

**13个最佳实践**

在XP中，集成了13个最佳实践，有趣的是，它们没有一个是创新的概念，大多数概念和编程一样老。其主要创新点在于提供一种良好的思路，将这些最佳实践结合在一起，并且确保尽可能彻底地执行它们，使得它们能够在最大程度上相互支持，紧接下来，我们就对每一种最佳实践进行一番了解。

**1. 计划游戏**

计划游戏的主要思想就是先快速地制定一份概要的计划，然后随着项目细节的不断清晰，再逐步完善这份计划。计划游戏产生的结果是一套用户故事及后续的一两次迭代的概要计划。

“客户负责业务决策，开发团队负责技术决策”是计划游戏获得成功的前提条件。也就是说，系统的范围、下一次迭代的发布时间、用户故事的优先级应该由客户决定；而每个用户故事所需的开发时间、不同技术的成本、如何组建团队、每个用户故事的风险，以及具体的开发顺序应该由开发团队决定。

首先客户和开发人员坐在同一间屋子里，每个人都准备一支笔、一些用于记录用户故事的纸片，最好再准备一个白板，就可以开始了。

* 客户编写故事：由客户谈论系统应该完成什么功能，然后用通俗的自然语言，使用自己的语汇，将其写在卡片上，这也就是用户故事。
* 开发人员进行估算：首先客户按优先级将用户故事分成必须要有、希望有、如果有更好三类，然后开发人员对每个用户故事进行估算，先从高优先级开始估算。如果在估算的时候，感到有一些故事太大，不容易进行估算，或者是估算的结果超过2人/周，那么就应该对其进行分解，拆成2个或者多个小故事。
* 确定迭代的周期：接下来就是确定本次迭代的时间周期，这可以根据实际的情况进行确定，不过最佳的迭代周期是2~3周。有了迭代的时间之后，再结合参与的开发人数，算出可以完成的工作量总数。然后根据估算的结果，与客户协商，挑出时间上够、优先级合适的用户故事组合，形成计划。

**2. 小型发布**

XP方法论秉承的是“持续集成，小步快走”的哲学，也就是说每一次发布的版本应该尽可能的小，当然前提条件是每个版本有足够的商业价值，值得发布。

由于小型发布可以使得集成更频繁，客户获得的中间结果也越频繁，反馈也就越频繁，客户就能够实时地了解项目的进展情况，从而提出更多的意见，以便在下一次迭代中计划进去。以实现更高的客户满意度。

**3. 隐喻**

相对而言，隐喻这一个最佳实践是最令人费解的。什么是隐喻呢？根据词典中的解释是：“一种语言的表达手段，它用来暗示字面意义不相似的事物之间的相似之处”。那么这在软件开发中又有什么用呢？总结而言，常常用于四个方面。

* 寻求共识：也就是鼓励开发人员在寻求问题共识时，可以借用一些沟通双方都比较熟悉的事物来做类比，从而帮助大家更好地理解解决方案的关键结构，也就是更好地理解系统是什么、能做什么。
* 发明共享词汇：通过隐喻，有助于提出一个用来表示对象、对象间的关系通用名称。例如，策略模式（用来表示可以实现多种不同策略的设计模式）、工厂模式（用来表示可以按需“生产”出所需类得设计模式）等。
* 创新的武器：有的时候，可以借助其他东西来找到解决问题的新途径。例如：“我们可以将工作流看做一个生产线”。
* 描述体系结构：体系结构是比较抽象的，引入隐喻能够大大减轻理解的复杂度。例如管道体系结构就是指两个构件之间通过一条传递消息的“管道”进行通信。

当然，如果能够找到合适的隐喻是十分快乐的，但并不是每种情况都可以找到恰当的隐喻，你也没有必要强求。

**4. 简单设计**

强调简单设计的价值观，引出了简单性假设原则，落到实处就是“简单设计”实践。这个实践看上去似乎很容易理解，但却又经常被误解，许多批评者就指责XP忽略设计是不正确的。其实，XP的简单设计实践并不是要忽略设计，而且认为设计不应该在编码之前一次性完成，因为那样只能建立在“情况不会发生变化”或者“我们可以预见所有的变化”之类的谎言的基础上的。

Kent Beck概念中简单设计是这样的：

* 能够通过所有的测试程序。
* 没有包括任何重复的代码。
* 清楚地表现了程序员赋予的所有意图。
* 包括尽可能少的类和方法
* 他认为要想保持设计简单的系统，需要具备简单思考的能力，拥有理解代码和修改的勇气，以及为了消除代码的“坏味道”而定期重构的习惯。
* 那么如何开始进行简单的设计呢？XP实践者们也总结也一些具体的、可操作的思考方法。
* 首先写测试代码：具体将在后面详细描述。
* 保持每个类只负责一件事：SRP（单一职责原则）是面向对象设计的基础原则之一。
* 使用Demeter（迪米特）法则：迪米特法则，也称为LoD法则、最少知识原则。也就是指一个对象应当对其他对象尽可能少地了解。用隐喻的方法来解释的话就是“只与你直接的朋友通信”、“不要和陌生人说话”。
* 使用CRC卡片进行探索。

**5. 测试先行／测试驱动开发**

首先，我们来看看两个不同的工匠是如何工作的吧。

* 工匠一：先拉上一根水平线，砌每一块砖时，都与这跟水平线进行比较，使得每一块砖都保持水平。
* 工匠二：先将一排砖都砌完，然后再拉上一根水平线，看看哪些砖有问题，对有问题的砖进行适当的调整。

你会选择哪种工作方法呢？但我们平时在编写程序的时候又是怎么做的呢？我们就是按工匠二的方法在工作呀！甚至有时候比工匠二还笨，是整面墙都砌完了，直接进行“集成测试”，经常让整面的墙倒塌。

不仅我们没有采用工匠一的工作方法，甚至有的时候程序员会以“开发工作太紧张”为理由，而忽略测试工作。但这样却导致了一个恶性循环，越是没有空编写测试程序，代码的效率与质量越差，花在找Bug、解决Bug的时间也越来越多，实际产能大打降低。由于产能降低了，因此时间更紧张，压力更大。你想想，为什么不拉上一根水平线呢？难道，我们不能够将后面浪费的时间花在单元测试上，使得我们的程序一开始就更健壮，更加易于修改吗？不过，编写测试程序当然要比拉一条水平线难道多，所以我们需要引入“自动化测试工具”，免费的xUnit测试框架就是你最佳的选择。

为了鼓励程序员原意甚至喜欢在编写程序之前编写测试代码，XP方法论还提供了许多有说服力的理由。

* 如果你已经保持了简单的设计，那么编写测试代码根本不难。
* 如果你在结对编程，那么如果你想出一个好的测试代码，那么你的伙伴一定行。
* 当所有的测试都通过的时候，你再也不会担心所写的代码今后会“暗箭伤人”，那种感觉是相当棒的。
* 当你的客户看到所有的测试都通过的时候，会对程序充满前所未有的信心。
* 当你需要进行重构时，测试代码会给你带来更大的勇气，因为你要测试是否重构成功只需要一个按钮。

测试先行是XP方法论中一个十分重要的最佳实践，并且其中所蕴含的知识与方法也十分丰富。

**6. 重构**

重构时一种对代码进行改进而不影响功能实现的技术，XP需要开发人员在闻到代码的坏味道时，有重构代码的勇气。重构的目的是降低变化引发的风险，使得代码优化更加容易。通常重构发生在两种情况之下。

* 实现某个特性之前：尝试改变现有的代码结构，以使得实现新的特性更加容易。
* 实现某个特性之后：检查刚刚写完的代码后，认真检查一下，看是否能够进行简化。

在《重构》一书中，作者Martin Fowler提示我们：在考虑重构时，应该要养成编写并经常运行测试代码的习惯；要先编写代码，再进行重构；把每一次增加功能都当做一次重构的好时机；将每一个纠正错误当做一次重构的重要时机。同时，该书中也列出大量需要重构的情况和重构方法。

* XP提倡集体代码所有制，因此你可以大胆地在任何需要修改的地方做改动。
* 由于在XP项目组中有完整的编码标准，因此在重构前无须重新定义格式。
* 在重构中遇到困难，和你结对编程的伙伴能够为你提供有效的帮助。
* 简单的设计，会给重构带来很大的帮助。
* 测试先行让你拥有了一个有效的检验器，随时运行一下就知道你重构的工作是否带来了影响。
* 由于XP在持续集成，因此你重构所带来的破坏很快就能够暴露，并且得以解决。

重构技术是对简单性设计的一个良好的补充，也是XP中重视“优质工作”的体现，这也是优秀的程序员必备的一项技能。

**7. 结对编程**

自从20世纪60年代，就有类似的实践在进行，长期以来的研究结果告诉我们结对编程的效率反而比单独编程更高。一开始虽然会牺牲一些速度，但慢慢的，开发速度会逐渐加快，究其原因，主要是结对编程大大降低了沟通的成本，提高了工作的质量，具体表现在：

* 所有的设计决策确保不是由一个人做出的。
* 系统的任何一个部分都肯定至少有2个人以上熟悉。
* 几乎不可能有2个人都忽略的测试项或者其他任务
* 结对组合的动态性，是一个企业知识管理的好途径。
* 代码总是能够保证被评审过。
* 而且XP方法论集成的其他最佳实践也能够使得结对编程更加容易进行：
* 编码标准可以消除一些无谓的分歧。
* 隐喻可以帮助结对伙伴更好地沟通。
* 简单设计可以使得结对伙伴更了解他们所从事的工作。

结对编程技术被誉为XP保持工作质量、强调人文主义的一个典型的实践，应用得当还能够使得开发团队之间的协作更加流畅、知识交流与共享更加频繁，团队的稳定性也会更加稳固。

**8. 集体代码所有制**

由于XP方法论鼓励团队进行结对编程，而且认为结对编程的组合应该动态地搭配，根据任务的不同、专业技能的不同进行最优组合。由于每个人都肯定会遇到不同的代码，所以代码的所有制就不再适合于私有，因为那样会给修改工作带来巨大的不便。

也就是说，团队中的每个成员都拥有对代码进行改进的权利，每个人都拥有全部代码，也都需要对全部代码负责。同时，XP强调代码是谁破坏的（也就是修改后发生问题），就应该由谁来修复。

由于在XP中，有一些与之匹配的最佳实践，因此你并无须担心采用集体代码所有制会让你的代码变得越来越乱：

* 由于在XP项目中，集成工作是一件经常性得工作，因此当有人修改代码而带来了集成的问题，会在很快的时间内被发现。
* 由于每一个类都会有一个测试代码，因此不论谁修改了代码，都需要运行这个测试代码，这样偶然性的破坏发生的概率将很小。
* 由于每一个代码的修改就是通过了结对的两个程序员共同思考，因此通常做出的修改都是对系统有益的。
* 由于大家都坚持了相同的编码标准，因此代码的可读性、可修改性都会比较好，而且还能够避免由于命名法、缩进等小问题引发经常性得代码修改。

集成代码所有制是XP与其他敏捷方法的一个较大不同，也是从另一个侧面体现了XP中蕴含的很深厚的编码情节。

**9. 持续集成**

在前面谈到小型发布、重构、结对编程、集体代码所有制等最佳实践的时候，我们多次看到“持续集成”的身影，可以说持续集成是对这些最佳实践的基本支撑条件。

小型发布是指在开发周期经常发布中间版本，而持续集成的含义则是要求XP团队每天尽可能多次地做代码集成，每次都在确保系统运行的单元测试通过之后进行。这样，就可以及早地暴露、消除由于重构、集体代码所有制所引入的错误，从而减少解决问题的痛苦

要在开发过程中做到持续集成并不容易，首先需要养成这个习惯。而且集成工作往往是十分枯燥、烦琐的，因此适当地引入每日集成工具是十分必要的。XP建议大家首先使用配置管理服务器将代码管理起来，然后使用Ant或Nant等XP工具，编写集成脚本，调用xUint等测试框架，这样就可以实现每当程序员将代码Check in到配置服务器上时，Ant就会自动完成编译和集成，并调用测试代码完成相应的测试工作。

**10. 每周工作40小时／可持续的速度**

这是最让开发人员开心的、管理者反对的一个最佳实践了，加班、再加班早已成为开发人员的家常便饭，也是管理者最常使用的一种策略，而XP方法论认为，加班最终会扼杀团队的积极性，最终导致项目失败，这也充分体现了XP方法关注人的因素比关注过程的因素更多一些。

Kent Beck认为开发人员即使能够工作更长的时间，他们也不该这样做，因为这样做会使他们更容易厌倦编程工作，从而产生一些影响他们效能的其他问题。因此，每周工作40小时是一种顺势行为，是一种规律。其实对于开发人员和管理者来说，违反这种规律是不值得的。

* 开发人员：如果不懂得休息，那么就无法将自己的节奏调整到最佳状态，那么就会带来很大的负面影响。而且在精神不集中的状态下，开发质量也得不到保证。
* 管理者：也许这可以称得上“第二种人月神话”，那就是你不得不通过延长每天的工作时间来获得更多的人月。这是因为，每个开发人员的工作精力是有限的，不可能无限增长，在精力不足的时候，不仅写出来的代码质量没有保障，而且还可能为项目带来退步的效果。因此采用加班的方式并不是一个理性的方式，是得不偿失的。

不过有一点是需要解释的，“每周工作40小时”中的40不是一个绝对数，它所代表的意思是团队应该保证按照“正常的时间”进行工作。那么如何做到这一点呢？

首先，定义符合你团队情况的“正常工作时间”。

其次，逐步将工作时间调整到“正常工作时间”。

再次，除非你的时间计划一团糟，否则不应该在时间妥协。

最后，鼓起勇气，制定一个合情合理的时间表。

正如米卢说过的“享受足球”一样，同样地，每一个开发人员应该做到“享受编程”，那么“每周工作40小时”就是你的起点。

团队只有持久才有获胜的希望。他们以能够长期维持的速度努力工作，他们保存精力，他们把项目看作是马拉松长跑，而不是全速短跑。

**11. 现场客户**

为了保证开发出来的结果与客户的预想接近，XP方法论认为最重要的需要将客户请到开发现场。就像计划游戏中提到过的，在XP项目中，应该时刻保证客户负责业务决策，开发团队负责技术决策。因此，在项目中有客户在现场明确用户故事，并做出相应的业务决策，对于XP项目而言有着十分重要的意义。

也许有人会问，客户提交了用户故事之后不就完成工作了吗？其实很多尝试过用户故事的团队都会发现其太过简单，包含的信息量极少，XP方法论不会不了解，因此，不会把用户故事当做开发人员交付代码的唯一指示。用户故事只是一个起点，后面的细节还需要开发人员与客户之间建立起来的良好沟通来补充。

作为一名有经验的开发人员，绝对不会对现场客户的价值产生任何怀疑，但是都会觉得想要实现现场客户十分困难。要实现这一点，需要对客户进行沟通，让其明白，相对于开发团队，项目成功对于客户而言更为重要。而现场客户则是保障项目成功的一个重要措施，想想在你装修房子的时候，你是不是常常在充当现场客户的角色呢？其实这隐喻就是让客户理解现场客户重要性最好的突破口。

其实现场客户在具体实施时，也不是一定需要客户一直和开发团队在一起，而是在开发团队应该和客户能够随时沟通，可以是面谈，可以是在线聊天，可以是电话，当然面谈是必不可少的。其中的关键是当开发人员需要客户做出业务决策时，需要进一步了解业务细节时能够随时找到相应的客户。

不过，也有一些项目是可以不要现场客户参与的：

* 当开发组织中已经有相关的领域专家时。
* 当做一些探索性工作，而且客户也不知道他想要什么时（例如新产品、新解决方案的研究与开发）。

**12. 编码标准**

编码标准是一个“雅俗共享”的最佳实践，不管是代表重型方法论的RUP，PSP，还是代表敏捷方法论的XP，都认为开发团队应该拥有一个编码标准。XP方法论认为拥有编码标准可以避免团队在一些与开发进度无关的细节问题上发生争论，而且会给重构、结对编程带来很大麻烦。试想如果有人将你上次写的代码的变量命名法做了修改，下次你需要再改这部分代码时，会是一种什么感觉呢？

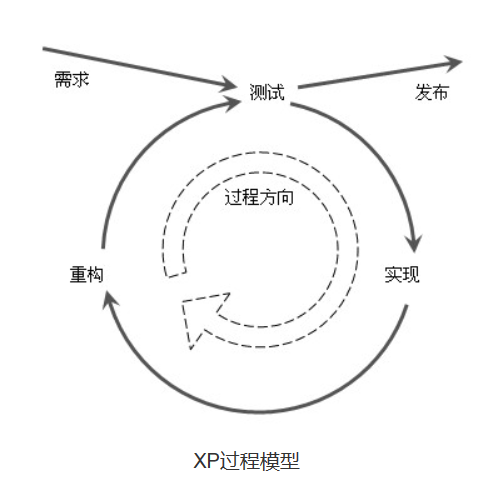
不过，XP方法论的编码标准的目的不是创建一个事无巨细的规则表，而是只要能够提供一个确保代码清晰，便于交流的指导方针。

如果你的团队已经拥有编码标准，就可以直接使用它，并在过程中进行完善。如果还没有，那么大家可以先进行编码，然后在过程中逐步总结出编码规则，边做边形成。当然除了这种文字规范以外，还可以采用一些如自动格式化代码工具之类的方法进行代码规范。，事实上，你只需要很好地贯彻执行其他的实践并且进行沟通，编码标准会很容易地浮现出来。

**13. 配合是关键**

有句经典名言“1+1>2”最适合表达XP的观点，Kent Beck认为XP方法论的最大价值在于在项目中融会贯通地运用12个最佳实践，而非单独地使用。你当然可以使用其中的一些实践，但这并不意味着你就运用了XP方法论。XP方法论真正能够发挥其效能，就必须完整地运用12个实践。

**XP过程模型**



对于XP过程，虽然每次迭代的都会按照传统瀑布的五个阶段有序进行，但每个阶段的产品都是在逐渐迭代的过程中产生（当然时机可能并不相同，或者有些根本不需要）。

此方式的迭代同时包含了上述的五个阶段，因此可以称为“纵向迭代”。

XP过程模型对各阶段资源配置水平以及执行者间的沟通有较高的要求，适合用户需求不明确、变更难以预测、探索预研、业务变动频繁的项目。

**DevOps**

DevOps的文化和哲学强调团队之间的协作、透明度和信任。它旨在打破开发和运维之间的隔阂和沟通障碍，以便在整个软件开发生命周期中，从需求分析和设计到编码、测试、部署和运维，都能够实现无缝的协作和交流。在这种文化中，重视持续改进和自动化，以便更快地响应客户需求和市场变化，并提高软件质量和可靠性。

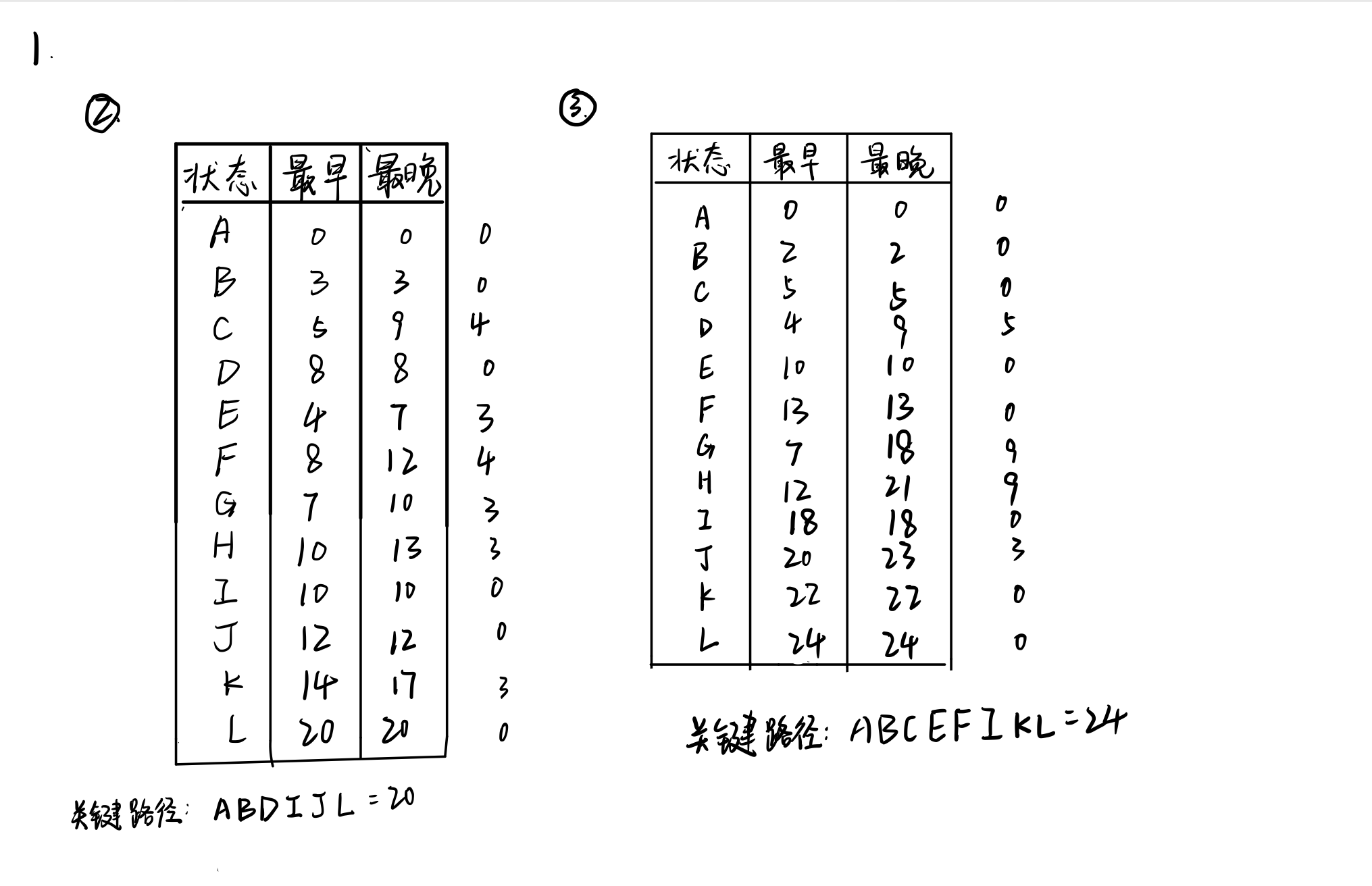
在DevOps的技术和工具方面，它主要关注以下几个方面：

1. 自动化 - DevOps鼓励使用自动化工具来减少手动操作的需要，以降低出错率，并加快软件开发和部署的速度。
2. 持续集成和持续交付 - DevOps通过持续集成和持续交付实现更快的软件交付。这个过程包括将代码的更改持续集成到主干代码库中，并自动构建、测试和部署应用程序。
3. 容器化 - DevOps使用容器技术，如Docker，来封装应用程序和依赖关系，从而使应用程序在不同环境中运行时更加一致和可靠。
4. 监控和反馈 - DevOps关注应用程序的监控和反馈。通过使用日志、度量和其他监控工具，DevOps可以及时发现和解决潜在的问题，并优化应用程序的性能和可靠性。

总之，DevOps是一种旨在改善软件开发和运维之间协作和交流的文化和哲学，并且它使用一组技术和工具来支持这种文化和哲学。它可以帮助软件开发团队更快、更频繁地部署和交付软件，同时提高软件质量和可靠性。

活动图练习：

练习题：



我们的项目：

