
敏捷过程

第 3 章 敏捷软件过程

许多人都经历过由于缺乏实践经验而导致的项目失败。缺乏有效的实践会导致不可预测或重复的错误以及白白浪费努力。延期的进度、增加的预算和低劣的质量致使客户对软件丧失信心。更长时间的工作却生产出更加低劣的软件产品，也使得开发人员感到沮丧。

一旦经历了这样的惨败，就会害怕重蹈覆辙。这种恐惧激发开发人员创建一个过程来约束软件工程活动、要求有某些人工制品输出。开发人员根据过去的经验来规定这些约束和输出，挑选那些在以前的项目中看起来好像工作得不错的方法。开发人员希望这些方法这次还会有效，从而消除开发人员的恐惧。

然而，项目并没有简单到使用一些约束和人工制品就能够可靠地防止错误的地步。当连续地犯错时，开发人员会对错误进行诊断，并在过程中增加更多的约束和人工制品来防止以后重犯这样的错误。经过多次这样的增加以后，开发人员就会不堪巨大、笨重的过程的重负，极大地削弱开发人员完成工作的能力。

一个大而笨重的过程会产生它本来企图去解决的问题。它降低了团队的开发效率，使得进度延期，预算超支。它降低了团队的响应能力，使得团队经常创建错误的产品。遗憾的是，许多团队认为，这种结果是因为没有采用更多的过程方法引起的。因此，在这种失控的过程膨胀中，过程会变得越来越庞大。

用失控的过程膨胀来描述 2000 年前后的许多软件公司中的情形是很合适的。虽然有许多团队在工作中并没有使用过程方法，但是采用庞大、重型的过程方法的趋势却在快速地增长，在大公司中尤其如此。本章主要介绍在此背景下诞生的，反对过度使用重型过程方法的软件开发哲学，即敏捷软件过程。本章的学习要点如下。

- 理解敏捷联盟诞生的背景原因及其宣言。
- 理解敏捷原则，掌握极限编程重要实践。
- 了解极限编程、Scrum 等敏捷过程。
- 理解敏捷统一过程的含义和适用场景。

3.1 敏捷联盟

2001 年年初，由于看到许多公司的软件团队陷入了不断增长的过程的泥潭，一批业界专家聚集在一起概括出了一些可以让软件开发团队具有快速工作、应变能力的价值观和原则。其称自己为敏捷(agile)联盟。在随后的几个月中，其发布了一份价值观声明，也就是**敏捷联盟宣言**(the manifesto of the agile alliance)，见图 3.1。

我们正在通过亲身实践以及帮助他人实践，揭示更好的软件开发方法，通过这项工作，我们认为：

（1）个体及其交互胜过过程和工具。

（2）可以工作的软件胜过面面俱到的文档。

（3）客户合作胜过合同谈判。

（4）响应变化胜过遵循计划。

在这些对比中，虽然右项也有价值，但是我们认为左项具有更大的价值。

宣言人：

Kent Beck	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekom	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

图 3.1　敏捷联盟宣言

1. 个体及其交互胜过过程和工具

人是获得成功的最为重要的因素。如果团队中没有优秀的成员，那么即使使用好的过程也不能从失败中挽救项目，但是，不好的过程却可以使最优秀的团队成员失去效用。如果不能作为一个团队进行工作，那么即使拥有一批优秀的成员也一样会惨败。

一个优秀的团队成员未必就是一个一流的程序员。一个优秀的团队成员可能是一个平均水平的程序员，但是却能够很好地和他人合作，合作、沟通以及交互能力要比单纯的编程能力更为重要。

合适的工具对于成功来说是非常重要的，像编译器、集成开发环境 IDE、源代码控制系统等，对于团队的开发者正确地完成工作是至关重要的。然而，工具的作用可能会被过分地夸大。使用过多的庞大、笨重的工具就像缺少工具一样，都是不好的。

记住，团队的构建要比环境的构建重要得多。许多团队和管理者就犯了先构建环境，然后期望团队自动凝聚在一起的错误。相反，应该首先致力于构建团队，然后再让团队基于需要来配置环境。

2. 可以工作的软件胜过面面俱到的文档

没有文档的软件是一种灾难。代码不是传达系统原理和结构的理想媒介。团队更需要编制易于阅读的文档，来对系统及其设计决策的依据进行描述。

然而，过多的文档比过少的文档更糟。编制众多的文档需要花费大量的时间，并且要使这些文档和代码保持同步，就要花费更多的时间。如果文档和代码之间失去同步，那么文档就会变成庞大的、复杂的谎言，会造成重大的误导。

对于团队来说，编写并维护一份系统原理和结构方面的文档总是一个好主意，但是那份文档应该是短小并且主题突出的。“短小”的意思是说，文档最多有一二十页。“主题突出”则是指文档应该仅论述系统的高层结构和概括的设计原理。

许多团队因为注重文档而非软件，导致进度拖延，这常常是一个致命的缺陷。有一个称为“Martin 文档第一定律（Martin’s first law of document）”的简单规则可以预防该缺陷的发生：**直到迫切需要且意义重大时，才来编制文档。**

3. 客户合作胜过合同谈判

客户不能像订购日用品一样来订购软件。客户不能够仅仅写下一份关于自己想要的软

件的名称或概要说明，就让人在固定的时间内以固定的价格去开发它。所有用这种方式来对待软件项目的尝试都以失败而告终。有时，失败的代价是惨重的。

告诉开发团队想要的东西，然后期望开发团队消失一段时间后就能够交付一个满足需要的系统来，这对于公司的管理者来说是具有诱惑力的。然而，这种操作模式将导致低劣的质量和失败。

成功的项目需要有序、频繁的客户反馈。不是依赖于合同或者关于工作的陈述，而是让软件的客户和开发团队密切地在一起工作，并尽量经常地提供反馈。

一个指明了需求、进度以及项目成本的合同存在着根本上的缺陷。在大多数的情况下，合同中指明的条款远在项目完成之前就变得没有意义。那些为开发团队和客户的协同工作方式提供指导的合同才是最好的合同。

4. 响应变化胜过遵循计划

响应变化的能力常常决定着一个软件项目的成败。当制订计划时，应该确保计划是灵活的并且易于适应商务和技术方面的变化。计划不能考虑得过远。首先，商务环境很可能会变化，这会引起需求的变动。其次，一旦客户看到系统开始运作，客户很可能会改变需求。最后，即使开发人员熟悉需求，并且确信它们不会改变，但仍然不能很好地估算出开发它们需要的时间。

一个缺乏经验的管理者往往希望创建一张优美的甘特(Gantt)图并贴到墙上，也许觉得这张图赋予了控制整个项目的权力、能够跟踪单个人的任务并在任务完成时将任务从图上去除。可以对实际完成日期和计划完成日期进行比较，并对出现的任何偏差做出反应。

实际上发生的是：这张图的组织结构可能会变化，又或者，当团队增加了对于系统的认识，当客户增加了对于需求的认识，图中的某些任务会变得可有可无，另外一些任务会被发现并增加到图中，简而言之，计划将会遭受形态上的改变，而不仅仅是日期上的改变。

较好的做计划的策略是：为下两周做详细的计划，为下三个月做粗略的计划，再以后就做极为粗糙的计划。开发人员应该清楚地知道下两周要完成的任务，粗略地了解一下以后三个月要实现的需求至于系统一年后将要做什么，有一个模糊的想法即可。

计划中这种逐渐降低的细致度，意味着开发人员仅仅对于迫切的任务才花费时间进行详细的计划。一旦制订了这个详细的计划，就很难进行改变，因为团队会根据这个计划启动工作并有了相应的投入。然而，由于计划仅仅涵盖了几周的时间，计划的其余部分仍然保持着灵活性。

3.2 敏捷原则

从上述的价值观中引出了下面的 12 条原则，它们是敏捷实践区别于重型过程的特征所在。

1. 最优先要做的是通过尽早的、持续的交付有价值的软件来使客户满意

曾有研究分析了对公司构建高质量产品方面有帮助的软件开发实践，发现了很多对于最终系统质量有重要影响的实践。其中一个实践表明，尽早地交付具有部分功能的系统和系统质量之间具有很强的相关性；初期交付的系统中所包含的功能越少，最终交付的系统的质量就越高。该项研究的另一项发现是，以逐渐增加功能的方式经常性地交付系统和最终

质量之间有非常强的相关性；交付得越频繁，最终产品的质量就越高。

敏捷实践会尽早地、经常地进行交付。努力在项目刚开始的几周内就交付一个具有基本功能的系统。然后，努力坚持每两周就交付一个功能渐增的系统。

如果客户认为目前的功能已经足够了，客户可以选择把这些系统加入产品中或者可以简单地选择再检查一遍已有的功能，并指出想要做的改变。

2. 即使到了开发的后期，也欢迎改变需求；敏捷过程利用变化来为客户创造竞争优势

这是一个关于态度的声明。敏捷过程的活动者不惧怕变化，一般认为改变需求是好的事情，因为那些改变意味着团队已经学到了很多如何满足市场需要的知识。

敏捷团队会非常努力地保持软件结构的灵活性，这样当需求变化时，对于系统造成的影响是最小的。在本书的后面部分，会学习一些面向对象设计的原则和模式，这些内容会帮助开发人员维持这种灵活性。

3. 经常性地交付可以工作的软件，交付的间隔可以从几周到几个月，交付的时间越短越好

交付可以工作的软件，并且尽早地（项目刚开始很少的几周后）、经常性地（此后每隔很少的几周）交付它。不赞成交付大量的文档或者计划。开发人员认为那些不是真正要交付的东西，其关注的目标是交付满足客户需要的软件。

4. 在整个项目开发期间，业务人员和开发人员必须天天都在一起工作

为了能够以敏捷的方式进行项目的开发，客户、开发人员以及涉众之间就必须要进行有意义的频繁的交互。软件项目不像发射出去就能自动导航的武器，必须要对软件项目进行持续不断地引导。

5. 围绕被激励起来的个人来构建项目，给其提供所需要的环境和支持，并且信任其能够完成工作

在敏捷项目中，人被认为是项目取得成功的最重要的因素。而所有其他的因素，比如过程、环境、管理等都被认为是次要的，并且当这些因素对于人有负面的影响时，就要对它们进行改变。

例如，如果办公环境对团队的工作造成阻碍，就必须对办公环境进行改变。如果某些过程步骤对团队的工作造成阻碍，就必须对那些过程步骤进行改变。

6. 在团队内部，最具有效果并且富有效率的传递信息的方法，就是面对面的交谈

在敏捷项目中，人们之间相互进行交谈。首要的沟通方式就是交谈。也许会编写文档，但是不会企图在文档中包含所有的项目信息。敏捷团队不需要书面的规范、计划或者设计。团队成员也可以去编写文档，如果对于这些文档的需求是迫切并且意义重大的，文档不是默认的沟通方式。默认的沟通方式是交谈。

7. 工作的软件是首要的进度度量标准

敏捷项目通过度量当前软件满足客户需求的数量来度量开发进度。它们不是根据所处的开发阶段、已经编写的文档的多少或者已经创建的基础结构代码的数量来度量开发进度的只有当 30％的必须功能可以工作时，才可以确定进度完成了 30％。

8. 敏捷过程提倡可持续的开发速度，责任人、开发者和用户应该能够保持一个长期的、恒定的开发速度

敏捷项目不是 50 米短跑，而是马拉松长跑。团队不是以全速启动并试图在项目开发期间维持那个速度；相反，团队以快速但是可持续的速度行进。

跑得过快会导致团队精力耗尽、出现短期行为以至于崩溃。敏捷团队会测量自己的速度，不允许自己过于疲惫。不会借用明天精力来在今天多完成一点工作。工作在一个可以使在整个项目开发期间保持最高质量标准的速度上。

9. 不断地关注优秀的技能和好的设计会增强敏捷能力

高的产品质量是获取高的开发速度的关键。保持软件尽可能的简洁、健壮是快速开发软件的途径。因而，所有的敏捷团队成员都致力于只编写其能够编写的最高质量的代码。不要制造混乱，不再告诉自己等有更多的时间时再来清理。如果今天制造了混乱，务必在今天把混乱清理干净。

10. 简单是根本，是允许未完成的工作最大化的艺术

敏捷团队不会试图去构建那些华而不实的系统，总是更愿意采用和目标一致的最简单的方法。并不看重对于明天会出现的问题的预测，也不会在今天就对那些问题进行防卫。相反，在今天以最高的质量完成最简单的工作，深信如果在明天发生了问题，也会很容易进行处理。

11. 最好的构架、需求和设计出自于自组织的团队

敏捷团队是自组织的团队。任务不是从外部分配给单个团队成员，而是分配给整个团队，然后来确定完成任务的最好方法。

敏捷团队的成员共同来解决项目中所有方面的问题。每个成员都具有项目中所有方面的参与权力，不存在单一的团队成员对系统构架、需求或者测试负责的情况。整个团队共同承担那些责任，每个团队成员都能够影响它们。

12. 每隔一定时间，团队会在如何才能更有效地工作方面进行反省，然后相应地对自己的行为进行调整

敏捷团队会不断地对团队的组织方式、规则、规范、关系等进行调整。敏捷团队知道团队所处的环境在不断地变化，并且知道为了保持团队的敏捷性，就必须要随环境一起变化。

3.3 极限编程实践

极限编程(eXtreme Programming，XP)是敏捷方法中最著名的一个。它由一系列简单却互相依赖的实践组成，这些实践结合在一起成了一个胜于部分结合的整体。本节将简要地探讨一下这个整体以及 XP 涉及的一些关键实践。

3.3.1 客户作为团队成员

客户需要和开发人员在一起紧密地工作，以便彼此知晓对方所面临的问题，并共同去解决这些问题。

谁是客户？XP 团队中的客户是指定义产品的特性并排列这些特性优先级的人或者团体。有时，客户是和开发人员同属一家公司的一组业务分析或者市场专家。有时，客户是用户团体委派的用户代表。有时，客户事实上是支付开发费用的人。但是在 XP 项目中，无论谁是客户，都是能够和团队一起工作的团队成员。

最好的情况是客户和开发人员在同一个房间中工作，次一点的情况是客户和开发人员之间的工作距离在 100 米以内。距离越大，客户就越难成为真正的团队成员。如果客户工

作在另外一幢建筑或另外一个省市，那么他将会很难融合到团队中来。

如果确实无法和客户在一起工作，该怎么办呢？建议是去寻找能够在一起工作、愿意并能够代替真正客户的人。

3.3.2　用户故事

为了进行项目计划，必须要知道和项目需求有关的内容，但是却不用知道得太多。对于做计划而言，了解需求只需要做到能够估算它的程度就足够了。开发人员可能认为，为了对需求进行估算，就必须要了解需求的所有细节，其实并非如此，开发人员必须要知道存在很多细节，也必须要知道细节的大致分类，但是开发人员不必知道特定的细节。

需求的特定细节很可能会随时间而改变，一旦客户开始看到集成到一起的系统，就更会如此。看到新系统的问世是关注需求的最好时刻。因此，在离真正实现需求还很早时就去捕获需求的特定细节，就很可能会导致做无用功以及对需求不成熟的关注。

在 XP 中，需要和客户反复讨论，以获取对于需求细节的理解，但是不去捕获那些细节。开发人员更愿意客户在索引卡片上写下一些自己认可的少量词语，这些只言片语可以提醒其记起这次交谈的内容。基本上在和客户进行书写的同一时刻，开发人员在该卡片上写下对应于卡片上需求的估算。估算是基于和客户进行交谈期间所得到的对于细节的理解进行的。

用户故事(user stories)就是正在进行的关于需求谈话的助记符，它是一个计划工具，客户可以使用它并根据它的优先级和估算代价来安排实现该需求的时间。

3.3.3　短交付周期

XP 项目每两周交付一次可以工作的软件，每两周的迭代(也可称为重复周期或循环周期)都实现了利益相关者的一些需求。在每次迭代结束时，会给利益相关者演示迭代生成的系统，以得到客户的反馈。每次迭代通常耗时两周。这是一次较小的交付，可能会被加入到产品中，也可能不会。它由客户根据开发人员确定的预算而选择的一些用户故事组成。一旦迭代开始，客户就同意不再修改当次迭代中用户故事的定义和优先级别。迭代期间，开发人员可以自由地将用户故事分解成任务，并依据最具技术和商务意义的顺序来开发这些任务。

XP 团队通常会创建一个计划来规划随后大约 6 次迭代的内容，这就是所谓的发布计划。一次发布通常需要“2 周/次×6 次＝12 周”，大概 3 个月的工作。它表示了一次较大的交付，通常此次交付会被加入到产品中。发布计划是由一组客户根据开发人员给出的预算所选择的排好优先级别的用户故事组成。

开发人员通过度量在以前的发布中所完成的工作量来设定本次发布预算。只要估算成本的总量不超过预算，客户就可以为本次发布选择任意数目的用户故事。客户同样可以决定在本次发布中用户故事的实现顺序。如果开发人员强烈要求的话，客户可通过指明哪些用户故事应该在哪次迭代中完成的方式，制定出发布中最初几次迭代的内容。

发布计划不是一成不变的，客户可以随时改变计划的内容。客户可以取消用户故事，编写新的用户故事，或者改变用户故事的优先级别。

3.3.4 结对编程

所有的产品代码都是由结对的程序员使用同一台计算机共同完成的。结对人员中的一位控制键盘并输入代码，另一位观察输入的代码并寻找着代码中的错误和可以改进的地方。两个人强烈地进行着交互，都全身心地投入到软件的编写中。

两人频繁互换角色。控制键盘的可能累了或者遇到了困难，其同伴会取得键盘的控制权。在一个小时内，键盘可能在两人之间来回传递好几次。最终生成的代码是由两人共同设计、共同编写的，两人功劳均等。

结对的关系每天至少要改变一次，以便每个程序员在一天中可以在两个不同的结对中工作在一次迭代期间，每个团队成员应该和所有其他的团队成员在一起工作过，并且其应该参与了本次迭代中所涉及的每项工作。

这将极大地促进知识在团队中的传播。但仍然会需要一些专业知识，并且那些需要一定专业知识的任务通常需要合适的专家去完成，那些专家几乎将会和团队中的所有其他人结对，这将加快专业知识在团队中的传播。这样，在紧要关头，其他团队成员就能够代替所需要的专家。有研究表明，结对非但不会降低编程人员的效率，反而会大大减少缺陷率。

3.3.5 持续集成与可持续开发

程序员每天会多次拆入其代码并进行集成，规则很简单。第一个拆入的只要完成拆入就可以了，其他人负责代码的合并工作。

XP 团队使用非阻塞的源代码控制工具。这意味着程序员可以在任何时候拆卸任何模块，而不管是否有其他人已经拆卸出这个模块，当程序员完成对模块的修改并把该模块加载回去时，必须要把自己所做的改动和在其前面加载该模块的程序员做的任何改动进行合并。为了避免合并的时间过长，团队的成员会非常频繁地加载其模块。

结对人员会在一项任务上工作 1～2 小时，创建测试用例和产品代码。在某个适当的间歇点，也许远远在这项任务完成之前，决定把代码加载回去。最重要的是要确保所有的测试都能够通过。把新的代码集成进代码库中，如果需要，会对代码进行合并。如果有必要，会和先予加载的程序员协商。一旦集成进了更改，就构建新的系统。运行系统中的每个测试，包括当前所有运行着的验收测试。如果破坏了原先可以工作的部分，会进行修正。一旦所有的测试都通过了，就算完成了此次拆入工作。

因而，XP 团队每天会进行多次系统构建，会重新创建整个系统。如果系统的最终结果是 CD，就录制该 CD。如果系统的最终结果是一个可以访问的 Web 站点，就安装该 Web 站点，或许会把它安装在一个测试服务器上。

软件项目不是全速的短跑，它是马拉松长跑。那些一过起跑线就开始尽力狂奔的团队在远离终点前就会筋疲力尽。为了快速完成开发，团队必须要以一种可持续的速度前进。团队必须保持旺盛的精力和敏锐的警觉。团队必须要有意识地保持稳定、适中的速度。

XP 的规则是不允许团队加班工作。在版本发布前的一个星期是该规则的唯一例外。如果发布目标就在眼前并且能够一蹴而就，则允许加班。

3.3.6　开放的工作空间

团队在一个开放的房间中一起工作，房间中有一些桌子每张桌子上摆放了 2～3 台工作站，每台工作站前有给结对编程的人员预备的两把椅子，墙壁上挂满了状态图表、任务明细表、UML 图等。

房间里充满了交谈的声音，结对编程的两人坐在互相能够听得到的距离内，每个人都可以得知另一人何时遇到了麻烦，每个人都了解对方的工作状态，程序员们都处在适合于激烈地进行讨论的位置上。

可能有人认为这种环境会分散人的注意力，很容易会让人担心由于持续的噪声造成干扰。事实上并非如此。而且，密歇根大学的一项研究表明，在“充满积极讨论的屋子”(war room，也称为战室)里工作，生产率非但不会降低，反而会成倍地提高。

3.3.7　简单的设计

XP 团队的设计尽可能地简单、具有表现和表达力。此外，仅仅关注于计划在本次迭代中要完成的用户故事，不会考虑那些未来的用户故事。相反，在一次次的迭代中，不断变迁系统设计，使之对正在实现的用户故事而言始终保持在最优状态。

这意味着 XP 团队的工作可能不会从基础结构开始，能并不先去选择使用数据库或者中间件。团队最开始的工作是以尽可能最简单的方式实现第一批用户故事。只有当出现一个用户故事迫切需要基础结构时，才会引入该基础结构。

下面 3 条 XP 指导原则可以对开发人员进行指导。

1. 考虑能够工作的最简单的事情

XP 团队总是尽可能寻找能实现当前用户故事的最简单的设计。在实现当前的用户故事时，如果能够使用平面文件，就不去使用数据库或者 EB 企业级 Java Bean；如果能够使用简单的 Socket 连接，就不去使用 ORB(对象请求代理)或者 RM(远程方法调用)；如果能够不使用多线程，就别去用它。尽量考虑用最简单的方法来实现当前的用户故事。然后，选择一种能够实际得到的和该简单性最接近的解决方案。

2. 你将不需要它

是的，但是要知道总有一天会需要数据库，会需要 ORB，也总有一天得去支持多用户。所以，现在就需要为那些东西做好准备，不是吗？

如果在确实需要基础结构前拒绝引入它，那么会发生什么呢？XP 团队会对此进行认真的考虑。开始时假设将不需要那些基础结构。只有在有证据，或者至少有十分明显的迹象表明现在引入这些基础结构比继续等待更加合算时，团队才会引入这些基础结构。

3. 一次且只有一次

极限编程者不能容忍重复的代码。无论在哪里发现重复的代码，都会消除这些重复代码。导致代码重复的因素有许多，最明显的是用鼠标选中一段代码后四处粘贴。当发现那些重复的代码时，会通过定义一个函数或基类的方法来消除它们。有时两个或多个算法非常相似，但是它们之间又存在着微妙的差别，就要把它们变成函数，或者使用**模板方法**(template method)模式。无论重复代码源于何处，一旦发现，就必须被消除。

消除重复最好的方法就是抽象。毕竟，如果两种事物相似的话，必定存在某种抽象能够

统一它们。这样，消除重复的行为会迫使团队提炼出许多的抽象，并进一步减少代码间的耦合。

3.3.8 重构

下面是对**重构**(refactoring)的一个简单介绍。

代码往往会腐化。随着添加一个又一个的特性，处理一个又一个的错误，代码的结构会逐渐退化。如果对此置之不理，这种退化最终会导致纠结不清，难以维护的混乱代码。

XP 团队通过经常性的代码重构来扭转这种退化。重构就是在不改变代码行为的前提下，对其进行一系列小的改造，以改进系统结构的实践活动。每个改造都是微不足道的，几乎不值得去做。但是所有的这些改造叠加在一起，就形成了对系统设计和构架显著的改进。

在每次细微改造之后，运行单元测试以确保改造没有造成任何破坏，然后再去做下一次改造。如此往复，周而复始，每次改造之后都要运行测试。通过这种方式，可以在改造系统设计的同时，保持系统可以正常工作。

重构是持续进行的，而不是在项目结束时、发布版本时、选代结束时甚至每天快下班时才进行的。重构是每隔一小时或者半小时就要去做的事情。通过重构，可以持续地保持代码尽可能干净、简单并且具有表现力。

3.3.9 隐喻

隐喻(metaphor)是唯一一个不具体、不直接的 XP 实践，也是所有 XP 实践中最难理解的一个。极限编程者在本质上都是务实主义者。隐喻这个缺乏具体定义的概念使其觉得很不舒服。的确，一些 XP 的支持者经常讨论把隐喻从 XP 的实践中去除。然而，在某种意义上，隐喻却是所有实践中最重要的实践之一。

比如智力拼图玩具，怎样才能知道如何把各个小块拼在一起？显然，每一块都与其他块相邻，并且它的形状必须与相邻的块完美地吻合。如果无法看到但是具有很好的触觉，那么通过锲而不舍地筛选每个小块，不断地尝试它们的位置，也能够拼出整个图形。

但是，相对于各个小块的形状而言，还有一种更为强大的力量把这些复杂的小块拼装在一起。这就是整张拼图的图案。图案是真正的向导。它的力量是如此之大，以至于如果图案中相邻的两块不具有互相吻合的形状，就能断定拼图玩具的制作者把玩具做错了。

这就是隐喻，它是将整个系统联系在一起的全局视图。它是对系统未来图景的展望，它使所有单独模块的位置和形状变得明显直观。如果模块的外观与整个系统的隐喻不符，那么就知道该模块是错误的。

隐喻通常可以归结为一个名字系统。这些名字提供了一个系统组成元素的词汇表，并且有助于定义它们之间的关系。

例如，开发一个以每秒 60 个字符的速度将文本输出到屏幕的系统。以这样的速度，字符充满整个屏幕需要一段时间。所以让产生文本的程序把产生的文本放到一个缓冲区中。当缓冲区满了的时候，把该程序交换到磁盘上。当缓冲区快要变空时，把该程序交换回来并让它继续运行。

用装卸卡车拖运垃圾来比喻整个系统。缓冲区是小车，屏幕是垃圾场，程序是垃圾制造者。所有的名字相互吻合，这有助于从整体上去考虑系统。

另举一例，开发一个分析网络流量的系统。每 30 分钟，系统会轮询许多的网络适配器并从中获取监控数据。每个网络适配器提供一小块由几个单独变量组成的数据，将这些数据块称为“面包切片”。这些面包切片是待分析的原始数据。分析程序“烤制”这些切片，因而被称为“烤面包机”。把数据块中的单个变量称为“面包屑”。总之，它是一个有用并且有趣的隐喻。

3.4　敏捷过程模型

敏捷开发以用户的需求进化为核心，采用迭代、循序渐进的方法进行软件开发。在敏捷开发中，软件项目在构建初期被切分成多个子项目，各个子项目的成果都经过测试，具备可视、可集成和可运行使用的特征。换言之，就是把一个大项目分为多个相互联系、可独立运行的小项目，并分别完成，在此过程中软件一直处于可使用状态。任何一个敏捷过程都可以由以下 3 个关键假设识别出来，这 3 个假设可适用于大多数软件项目。

(1) 提前预测哪些需求是稳定的、哪些需求会变化非常困难。同样地，预测项目进行中客户优先级的变化也很困难。

(2) 对很多软件，设计和构建是交错进行的。事实上，两种活动应当顺序开展以保证通过构建实施来验证设计模型，而在通过构建验证之前很难估计应该设计到什么程度。

(3) 从制订计划的角度来看，分析、设计、构建和测试并不像我们所设想的那么容易预测。

3.4.1　XP 过程

XP 包含 4 个框架活动：策划、设计、编码、测试。

1. 策划

(1) 开始于建立“**用户故事**”。

(2) 敏捷团队评估每个故事并给出**成本**。

(3) 故事被分组用于**可交付增量**。

(4) 对发布日期做出**承诺**。

(5) 在第一个发行版本(软件增量)之后，“**项目速度**”用于帮助建立后续发行版本(软件增量)的发布日期。

2. 设计

XP 设计严格遵循 KIS (keep it simple，保持简洁)原则，通常更愿意使用简单设计而不是更为复杂的表述。

(1) 严格遵守 KIS 原则。

(2) 鼓励使用 CRC(类名，类的职责，类的协作关系)卡片。

(3) 在设计中遇到困难，XP 推荐立即建立这部分设计的可执行原型，实现并评估设计原型。

(4) 鼓励“重构”(一种迭代式改进内部程序设计的方法)。

3. 编码

(1) 建议在开始编码之前为每个故事开发一系列单元测试。

(2) 鼓励“结对编程”。

4. 测试

(1) 所有的单元测试每天都要执行。

(2) “验收测试”由客户定义,将着眼于客户可见的、可评审的系统级的特征和功能。

3.4.2 Scrum

Scrum 的标准释义为:Scrum 是一个框架,在这个框架中人们可以解决复杂的自适应问题,同时也能高效并有创造性地交付尽可能高价值的产品。换言之,Scrum 其实就是一种团队管理工作的方式,其将工作分解为较小的工作单元,并在周期性固定的时间段内持续地交付工作单元。其中,周期性固定的时间段称为迭代(iteration)或者冲刺(sprint),较小的工作单元称为用户故事(user story)。用户故事可以使用特定的格式来描述,其描述了一个对于客户有价值的工作,而且可以在一个迭代周期内完成。

1. Scrum 核心价值

Scrum 核心价值包括以下内容。

(1) **公开**(openness):团队通过自己的方式共同完成工作,每个成员都对进展和问题了如指掌。

(2) **勇气**(courage):每个人不是一个人在战斗,就有了整个团队的支持,就有了更大的勇气来进行挑战。

(3) **承诺**(commitment):每个人对团队承担的工作有了更大的掌控,更加坚定了对成功的承诺。

(4) **尊重**(respect):团队中的每个人都有其特定的背景和经验,互相尊重,谦虚学习。

(5) **专注**(focus):每个人将全部精力和技能都聚焦在所承诺的工作上,团队同心协力来促使更快交付。

2. Scrum 框架的结构

Scrum 框架的结构包括 3 种角色、5 种事件和 3 种工件。

3 种角色如下。

(1) **产品负责人**(product owner):产品负责人是产品最终用户的代表,负责确定产品的方向和愿景,定义产品发布的计划、内容和优先级。产品负责人要不断地与开发团队沟通,保证团队在做业务角度来说最正确的事情。产品负责人是产品待办列表的唯一负责人。

(2) **Scrum 教练**(scrum master):Scrum 教练负责确保团队合理地运作 Scrum,帮助团队移除实施中的障碍。

(3) **开发团队**(development team):一个自组织的跨技能的小团队,承担实际开发工作,负责在周期性的迭代中不断的交付有价值的工作。开发团队通过集体共同交付价值,而不是通过个体。

5 种事件如下。

(1) **sprint**:sprint 本身也是一种事件,其包含下面 4 种事件。

(2) **迭代计划会议**(sprint planning meeting):在每个迭代之初,产品负责人和开发团队共同来计划在迭代周期内要完成的工作。产品负责人负责向团队讲解要完成的工作,开发团队负责对工作进行估计。

(3) **每日站立会议**(daily standup meeting)：每天，产品负责人和开发团队都要进行一个短暂的沟通。在会议期间，每个团队成员都要回答 3 个问题："昨天做了什么？""今天准备做什么？""遇到了什么问题？"。

(4) **迭代评审会议**(sprint review meeting)：在迭代周期结束时，开发团队向产品负责人及所有相关人员进行演示，并接受反馈。

(5) **迭代回顾会议**(sprint retrospective meeting)：Scrum 团队在迭代结束之后会进行一次迭代回顾会议，通过这次会议对迭代的过程进行总结，以促使团队自我持续改进。

3 种工件如下。

(1) **产品待办列表**(product backlog)：这是一个产品负责人想要交付的产品功能列表。产品负责人负责维护该列表，并且将列表项按照交付优先级进行排序。

(2) **冲刺待办列表**(sprint backlog)：这是一个迭代计划会议的输出、包含开发团队在迭代周期内所要完成的工作列表。

(3) **产品增量**(product increment)：每个迭代周期都需要交付高质量的产品增量。产品增量必须满足 Scrum 团队对完成标准(definition of done)的定义。

3. Scrum 的工作流程

Scrum 的工作流程如图 3.2 所示。

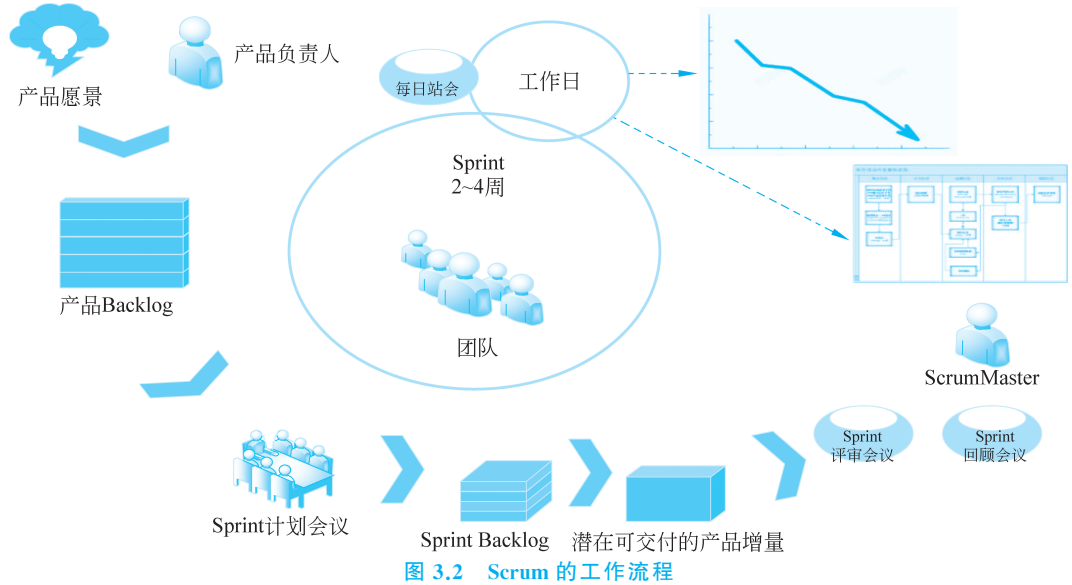

图 3.2　Scrum 的工作流程

(1) **冲刺周期的开始**。从产品待办列表中选取一些条目，放入到冲刺计划中。在冲刺计划中，先开冲刺计划会议，会议主要确定"这些条目怎么做"以及"如何去实现这些条目"两个问题。在会议结束会有当前冲刺的目标、冲刺待办列表两个输出。

(2) **冲刺周期的进行**。在冲刺周期中每天都会进行每日站立会议，每个人都需要总结"昨天做了什么""今天准备做什么""遇到了什么问题"。

(3) **冲刺周期的结束**。当 Sprint 周期结束，得到了产品增量，接着进行迭代评审会议。主要对产品进行反馈。在迭代评审会议之后，Scrum 团队会进行一次迭代回顾会议，通过这次会议对迭代的过程进行总结，以促使团队自我持续改进。

3.5 敏捷统一过程

敏捷统一过程(AUP)是 Rational 统一过程(RUP)的简化版本。它描述了一种使用敏捷技术和概念开发业务应用软件的简单、易于理解的方法,但仍然保持 RUP 的真实性。人们一直试图让敏捷尽可能简单,无论是在方法上还是在描述上。这些描述很简单,切中要害,如果需要,可以在网上链接到细节。该方法应用了敏捷技术,包括测试驱动开发(TDD)、敏捷模型驱动开发(AMDD)、敏捷变更管理和数据库重构,以提高生产率。

图 3.3 描述了 AUP 的生命周期。需要注意到的第一件事是活动与动作的边界已经改变了。首先,建模包括了 RUP 的业务建模、需求以及分析和设计过程。正如所看到的,建模是 AUP 的一个重要部分,但它并没有主导整个过程——应该通过创建“几乎不够好”的模型和文档来保持敏捷。其次,配置和变更管理现在变为了配置管理。在敏捷开发中,变更管理活动通常是需求管理工作的一部分,也成为建模活动的一部分。

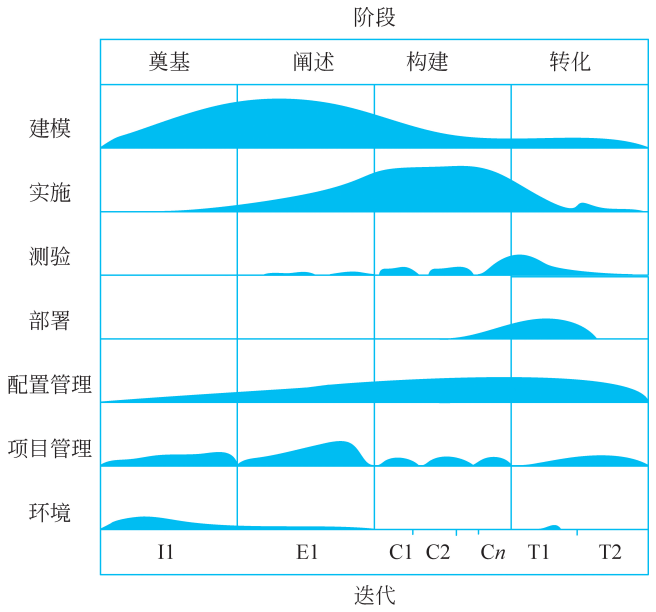

图 3.3 敏捷统一过程的生命周期

3.5.1 宏观上连续

敏捷统一过程的连续性体现在如下 4 个阶段。

(1) **奠基**:目标是确定项目的初始范围、系统的潜在体系结构,并获得初始项目资金和利益相关者的认可。

(2) **阐述**:目标是证明系统的体系结构。

(3) **构建**:目标是定期、增量地构建工作软件,以满足项目干系人的最高优先级需求。

(4) **转化**:目标是验证系统并将其部署到生产环境中。

3.5.2　微观上迭代

微观上，活动以迭代的方式执行，定义开发团队成员为构建、验证和交付满足其利益相关者需求的工作软件而执行的活动。这些活动包括如下。

（1）**建模**：本活动的目标是了解组织的业务、项目正在解决的问题域，并确定解决问题域的可行解决方案。

（2）**实施**：本活动的目标是将模型转换为可执行代码，并执行基本级别的测试，特别是单元测试。

（3）**测验**：本活动的目标是进行客观评估，以确保质量。这包括发现缺陷，验证系统是否按设计工作，以及验证是否满足要求。

（4）**部署**：本活动的目标是计划系统的交付，并执行计划，使系统可供最终用户使用。

（5）**配置管理**：本活动的目标是管理对项目文件的访问。这不仅包括随着时间的推移跟踪文件版本，还包括控制和管理对它们的更改。

（6）**项目管理**：本活动的目标是指导项目中发生的活动。这包括管理风险、指导人员（分配任务、跟踪进度等），以及与项目范围外的人员和系统进行协调，以确保项目按时、在预算内交付。

（7）**环境**：本活动的目标是通过确保团队根据需要提供适当的流程、指导（标准和指南）和工具（硬件、软件等）来支持其余工作。

3.5.3　持续增量发布

与一次交付所有软件的“大爆炸”方法不同，将软件分部分发布到生产中（例如，版本 1，然后是版本 2，等等）。AUP 团队通常在每次迭代结束时将开发版本交付到预生产阶段区域。如果应用程序的开发版本要经过预生产质量保证（QA）、测试和部署过程，则可能会发布到生产中。在图 3.4 中，可以看到第一个生产版本通常比后续版本花费更长的时间来交付；在系统的第一个版本中，可能需要将许多“管道”安装到位，而团队可能还没有“凝聚”起来，使成员能够高效地协作。第一个生产版本可能需要 12 个月才能交付，第二个版本需要 9 个月，然后其他版本每 6 个月交付一次。早期关注部署问题不仅可以避免问题，还可以在开发过程中利用已有的经验。例如，当将软件部署到暂存区域时，应该记录哪些可用，哪些不可用，这些记录可以作为安装脚本的主干。

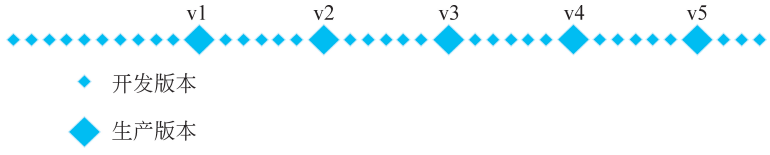

图 3.4　持续增量交付过程

3.5.4　AUP 的原则

敏捷统一过程基于以下原则。

（1）**员工知道自己在做什么**：人们不会阅读详细的流程文档，但其会不时需要一些高

级指导和培训。如果感兴趣，AUP产品提供了许多详细信息的链接，但不会强迫任何人使用这些链接。

(2) **简单**：每件事都是用几页而不是几千页来简洁地进行描述。

(3) **敏捷性**：敏捷性符合敏捷联盟的价值观和原则。

(4) **关注高价值活动**：重点是那些真正重要的活动，而不是项目中可能发生的每件事。

(5) **工具独立性**：可以使用任何自己想要的敏捷开发工具集。建议是使用最适合工作的工具，通常是简单的工具，甚至是开源工具。

(6) **根据需要定制**：AUP产品可以通过任何常见的HTML编辑工具轻松定制。不需要购买特殊工具或参加课程来定制AUP。

3.5.5 何时采用AUP

如果想要介于XP和传统RUP之间的东西，一个敏捷但明确包含你习惯的活动和工作的过程，那么AUP很可能适合你。许多组织对XP持谨慎态度，因为它似乎太轻了：XP没有明确显示如何创建管理层希望看到的一些工作。另外是RUP，管理层似乎很喜欢它，但是开发人员由于大量的工作而对它持怀疑态度。这也是不幸的，因为RUP提供了很多东西，并且可以简化为一些非常有用的东西(这正是IBM Rational建议做的)。AUP介于两者之间，采用了XP的许多敏捷技术和其他敏捷过程，但保留了RUP的一些形式。

AUP并不适合所有人。AUP要么是两个世界中最好的，要么是两个世界中最坏的，由程序员自己来判断。极端程序员可能会发现AUP相当繁重，“传统RUP”用户可能会发现它过于精简。如果想要更轻量级的方法，则建议使用XP。

3.6 本章小结

每位软件开发人员、每个开发团队的职业目标，都是给其雇主和客户交付最大可能的价值。可是，有时项目以令人沮丧的速度失败，或者未能交付任何价值。虽然在项目中采用过程方法是出于好意的，但是膨胀的过程方法对于失败至少是应该负一些责任的。敏捷软件开发的原则和价值观构成了一个可以帮助团队打破过程膨胀循环的方法，这个方法关注的是可以达到团队目标的一些简单的技术。

目前，已经有许多的敏捷过程可供选择，包括Scrum、Crystal特征驱动软件开发(feature driven development，FDD)、自适软件开发(adaptive software develop，ADP)以及最重要的极限编程XP。

极限编程是一组简单、具体的实践，这些实践结合在一起形成了一个敏捷开发过程。该过程已经被许多团队使用过，并且取得了好的效果。极限编程是一种优良的、通用的软件开发方法。项目团队可以拿来直接采用，也可以增加一些实践，或者对其中的一些实践进行修改后再采用。

XP与Scrum是敏捷方法中被业界采用最为广泛采用的两种实践。Scrum注重的是管理和组织实践，而XP关注的是实际的编程实践，两者都聚焦于信息价值流和信息沟通，除了迭代长度稍有差别外，大多数Scrum实践与XP是兼容且相互补充。组合使用Scrum和XP会有显著收获。XP的结对编程、测试驱动开发(TDD)、持续集成等最佳实践仍然可以

在 Scrum 中使用。

敏捷过程还可与统一过程结合起来，形成敏捷统一过程，供给那些同时看重 XP 的精简和 RUP 的稳健特性的开发团队。

习　题　3

1. 为什么会产生传统过程模型和敏捷过程模型这两种看似对立的模型？它们之间可以区分对错吗？如果可以，谁对谁错？如果不可以，为什么？

2. 敏捷的原则包含哪些？试列举 3 条，并结合实践讨论它们可以如何在你过往的项目中实施。

3. 如何理解敏捷统一过程的连续性和迭代性？