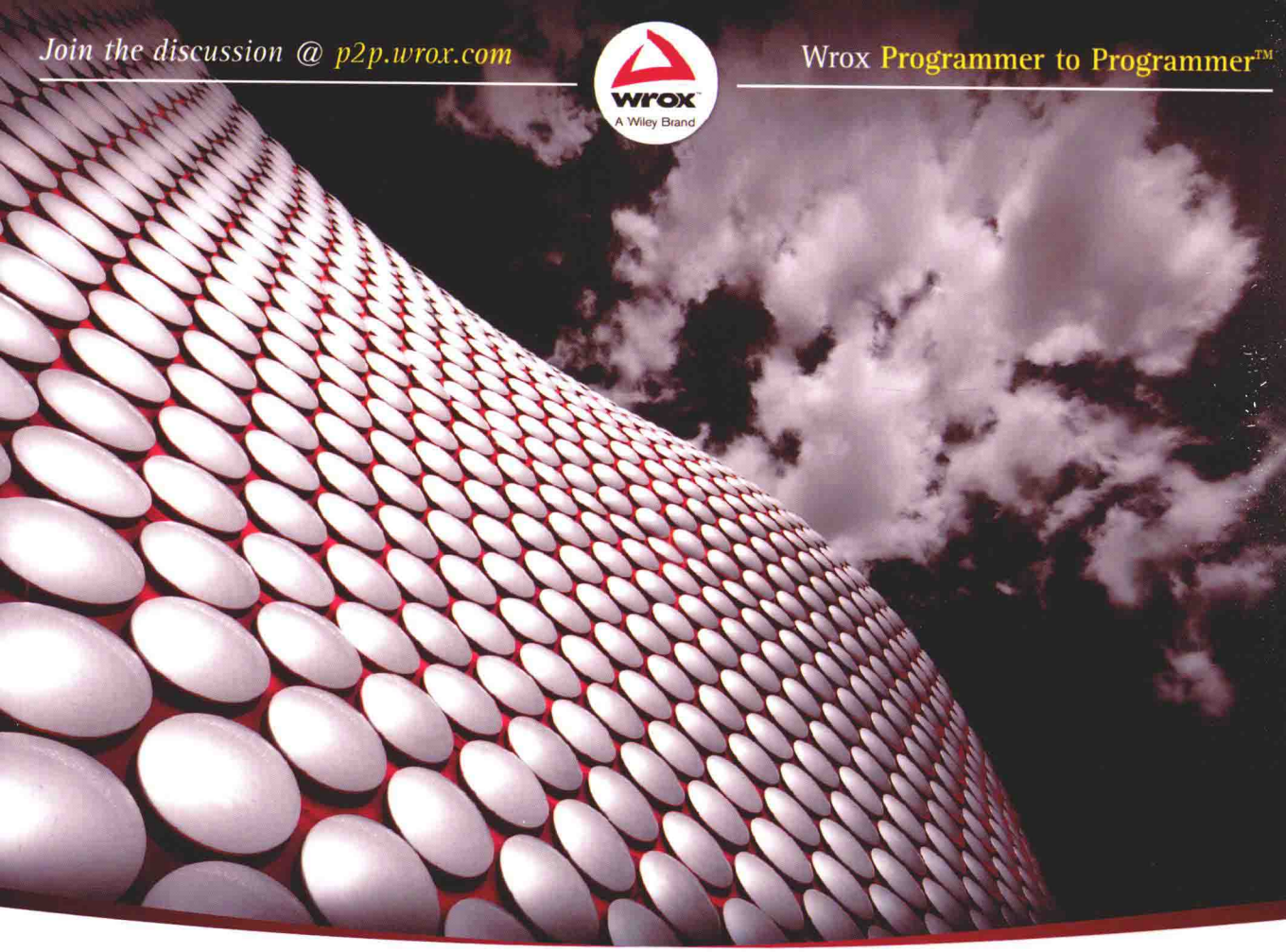

Beginning Software Engineering

软件工程入门经典

[美] Rod Stephens 著
明道洋 曾庆红 译

清华大学出版社

软件工程入门经典

[美] Rod Stephens 著
明道洋 曾庆红 译

清华大学出版社

北 京

Rod Stephens
Beginning Software Engineering
EISBN: 978-1-118-96914-4

图书在版编目(CIP)数据

软件工程入门经典/(美)罗德·斯蒂芬森(Rod Stephens)著；明道洋，曾庆红 译. —北京：清华大学出版社，2016
书名原文: Beginning Software Engineering
ISBN 978-7-302-43926-4

Ⅰ. ①软…　Ⅱ. ①斯…　②明…　③曾…　Ⅲ. ①软件工程—基本知识　Ⅳ. ①TP311

中国版本图书馆 CIP 数据核字(2016)第 111162 号

责任编辑：王　军　韩宏志
装帧设计：孔祥峰
责任校对：曹　阳
责任印制：沈　露

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：清华大学印刷厂
装 订 者：北京市密云县京文制本装订厂
经　　销：全国新华书店
开　　本：185mm×260mm　　印　　张：23.5　　字　　数：602 千字
版　　次：2016 年 7 月第 1 版　　印　　次：2016 年 7 月第 1 次印刷
印　　数：1~4000
定　　价：59.80 元

产品编号：064484-01

目　录

第 I 部分

进　阶

软件和教堂大致一样，我们首先建造它们，然后祈祷。

——Samuel Redwine

软件工程大体上是一个简单的两步过程：编写最畅销的程序，然后通过所得购买一些昂贵玩具。遗憾的是第一步可能相当困难。说“编写最畅销的程序”有些像告诉一个作家“撰写最畅销的书”，或是告诉一个垒球运动员“triple to left”。这是一个不错的想法，但知道目标并不一定就真正有助于实现它。

开发优秀的软件需要处理大量的复杂任务，其中的任何一个环节都可能失败，从而葬送掉整个项目。为让软件项目有规可循，多年来人们发明了大量的方法和技术。其中的一些，如瀑布和V模型法，在开发前使用了详细的需求规范来精确定义想要的结果。其他一些，如Scrum和敏捷方法，要依赖频繁反馈的快节奏增量开发，以确保项目的有序进行(还有一些方法，如牛仔编码(cowboy coding)和极限编程(extreme programming)，听起来与其说是软件开发方法，不如说更像动作冒险片)。

不同的开发方法学使用不同的方法，但它们都执行几乎相同的任务。它们都要确定软件要做什么以及如何做。它们要生成软件，消除代码中的bug(至少代码中的部分bug)，确保软件或多或少应该做该做的事情，并且部署最终的结果。

注意：

我把这些基本项称为任务，而不是“阶段”或“步骤”，是因为不同的软件工程方法对它们的处理方式、处理时间都不尽相同。把它们称为“阶段”或“步骤”容易让人误解，因为这暗示所有项目都以相同的可预见顺序通过这些阶段。

第Ⅰ部分介绍了任何成功的软件项目在处理这些任务时，都必须采用的一些方法。这部分内容主要介绍软件开发中的一些主要步骤，以及某个项目未能成功处理这些任务的种种表现方式。本书的第Ⅱ部分主要探讨不同的方法(如瀑布法和敏捷法)如何处理这些任务。

本书第Ⅰ部分的第1章主要从一个较高的层面对软件开发进行概述。后续的一些章节主要和开发过程的一些具体环节有关。

第 1 章 软件工程概览

设计软件有两种方法：一种是简单到明显没有缺陷，另一种复杂到缺陷不那么明显。第一种方法相当困难。

——C.A.R. Hoare

本章主要内容

- 成功的软件工程需要的一些基本步骤
- 软件工程方法和其他工程方法的异同
- 修复一个 bug 如何导致引入其他 bug
- 尽早侦测错误为何如此重要

软件工程和其他类型的工程在很多方面有共同之处。无论是建造桥梁、飞机场、核电站还是新改进的九宫格游戏，都需要完成某些任务。例如，需要制定一个计划，按计划行事，勇敢地克服各种意想不到的困难，最后雇佣一个很棒的乐队在剪彩仪式上演奏。

以下将介绍的这些步骤，可用来确保某个软件项目的正常进行。对于一些大型项目而言，虽然这些步骤还是有很多差异，但或多或少还是存在一些共同之处。后续章节将对这些任务进行深入探讨。

1.1 需求收集

没有计划，大型项目就不可能成功。有时，某个项目并未严格按照计划执行，但是每个大型的项目必须有计划。这样的计划用于告诉项目成员要做什么，什么时候开始做，需要多长时间，其中最重要的莫过于项目的目标是什么。

软件项目的第一个步骤是确定需求。首先需要确定用户的所想和所需，这取决于用户的需求，可能非常费时。

客户是谁

有时，很容易确定客户是谁。如果是为自己公司的另外一个部分编写软件，则客户是谁就非常明显。这种情况下，就可以和他们坐下来共同探讨一下软件要做什么。

其他一些情况下，至于谁将使用最终的软件，还模糊不清。例如，如果是创建一个新的在线纸牌游戏，则直到游戏开始推广之后，要弄清楚客户是谁都很难。

有时，甚至连自己都是客户。我总是在为自己编写软件。这有很多有利之处，例如，我很清楚自己想要什么，同时也或多或少地知道一些软件功能的开发难度(遗憾的是，有时我难以对自己说“不”，所以这些项目总是一再被拖延)。

对于任何项目而言，首先应该设法确定我们的客户，尽可能与他们多接触，只有这样才能设计出最有用的应用程序。

确定客户的所想和所需之后(并非总是相同)，就可以将它们转变为需求文档。这些文档告诉客户它们将得到什么，同时也可以告诉项目成员他们将创建什么。

在整个项目中，为确保项目朝着正确的方向推进，无论客户还是项目成员都可以查阅这些文档。如果有人建议在项目中包含一个视频教程，那么就可以查看一下需求文档，看看是否有这样的需求。如果它是新的功能，而且确实有用，就可以允许这样的修改，这样就不至于打乱其他计划。

如果上述这个新需求没有任何意义(可能对于项目毫无价值可言，也可能是时间并不允许)，那么就可以将其推延至以后的版本中。

发生变更

尽管软件工程和其他类型的工程有一些相似之处，但由于软件并非以物理的形式存在这一事实，它们之间也还是存在着较大的差异。由于软件的可塑性很强，因此直到最终发行时，用户都将频繁地要求新增一些新功能。他们可能要求开发人员缩短日程计划，无休止地进行请求变更，如改变数据平台，甚至是硬件平台(的确，这些都在我身上发生过)。他们坚持认为：“程序就是‘0’和‘1’，‘0’和‘1’并不关心它们是运行在Android平板电脑上还是Windows Phone上，是不是？”

相反的是，某个公司不会要求一个建筑公司，在最后一刻搬迁到街对面的一个新会展中心；城市的主管部门也不会让施工者，在一个高速大桥正开放时，再去增加一个额外的车道；在一栋90层的大楼竣工之时，没有人愿意尝试在底楼再插入一个中庭层。

1.2 概要设计

知道项目的需求后，就可以开始概要设计。概要设计主要包含和决策有关的一些事项，例如，使用何种平台(如桌面电脑、笔记本、平板电脑、手机)、何种数据设计(如直接访问、1层或2层)以及与其他系统的接口(如外部采购系统)。

概要设计也应包含和项目架构有关的一些概要信息。应该根据我们所处理的项目的主要功能区域，把项目分解成一些较大的块。根据我们所使用的方法，可能还要包含一个需要创建的模块清单或者是类系列清单。

例如，要开发一个用于管理鸵鸟赛跑结果的系统，我们可能决定使用下列一些主要的功能块：

- 数据库(保存数据)。
- 类(如 Race、Ostrich 以及 Jockey 类)。
- 用户接口 (输入鸵鸟和骑师数据、输入比赛结果、生成成绩报告、创建新的比赛)。
- 外部接口(发送信息、给游戏的参与者和爱好者群发邮件)。

概要设计一定要覆盖项目需求的每个方面。概要设计应该明确规定这些部分做什么以及它们之间的交互方式，但要尽可能少包含如何实现的一些具体细节。

设计还是不设计，这是一个问题

此时，极限编程、Scrum以及其他增量开发方法的爱好者们，可能会不耐烦地抱怨这些方法为什么不需要概要设计。第5章“概要设计”将更深入地探讨这个问题。至于现在，我只能声明每种开发方法都离不开设计，即使是在没有明确规定的前提下也是如此。

1.3　详细设计

经过概要设计以后，项目已经被划分成了不同的部分，可将它们分配给项目组进行详细设计。详细设计包含有关该项目各部分如何工作的一些信息，无须给出它们的具体实现细节，但应该给实现它们的开发人员提供足够的指导性说明。

例如，鸵鸟赛跑程序数据库部分还应包含一个数据库的初步设计。可以简要地说明一下用于保存比赛、鸵鸟以及骑师信息的一些数据表。

此时，我们也将发现项目不同部分之间存在一些交互，可能还需要进行很多调整。鸵鸟赛跑程序的外部接口可能需要一张新的表格来保存电子邮件、发送短消息以及和其他爱好者有关的一些信息。

1.4　开发

经过概要设计和详细设计后，程序员们就可以开始工作了(事实上，程序员们一直在努力收集需求，进行概要设计，并把概要设计的结果细化成详细设计，但开发才是程序员最热衷的部分)。程序员们继续对详细设计的结果进行细化，直到知道如何通过代码实现这些设计。

(事实上，在我最喜欢的一种开发技术中，基本上只需要对设计进行不断的细化，给出越来越详细的信息，直到它更易于编写代码就行了，然后就可以准确地编码了)。

程序员编写代码，对其进行测试，以确保它不包含任何错误。

此时，任何有经验的程序员都应该窃笑，如果不能放声大笑。

编程公理：重要的程序不可能完全没有bug(这里换一种措辞)。

程序员编写代码，对其进行测试，找出并移除尽可能多的bug。

1.5 测试

有效地对代码进行测试极其辛苦。如果只是编写代码，则显然不会故意插入bug。如果知道代码中有bug，则编写前就移除了。这样的思想导致程序员假定他们的代码都正确无误(我想他们只是天生的乐观主义者)，因此他们不会进行彻底的测试。

即使是已经过彻底测试的一段不包含(或几乎没有)bug的特定代码，也不能确保它就能和系统的其他部分一起正常工作。

解决这些问题(开发人员不测试他们的代码和代码各部分之间不能在一起工作)的一种方法是执行不同类型的测试。开发人员首先测试他们自己的代码，然后让没有编写这些代码的测试人员进行测试。当某个代码段看似能够正常工作之后，就把它整合到剩下的项目中。事情的整个过程就是测试新代码是否影响代码的其他部分。

无论何时测试失败，这些程序员都必须返回到代码中，弄清楚问题出在哪里以及如何修复。这些代码在经过修复之后，还需要返回测试队列，重新测试。

一群 bug

此时，我们应该想知道为什么要重新测试这些代码。毕竟，刚刚修复过，对吗？

遗憾的是，修复一个bug有时还可能产生一个新的bug。对bug的修复有时是不正确的。有时它破坏依赖原始bug行为的另一部分代码。已知的bug中隐藏着未知的bug。

还可能是这些程序员把一些正确的行为改成了方式不同的正确行为，而并未意识到其他的一些代码也将依赖这个正确的行为(想象一下，如果有人改变了你家冷热水管的排列，尽管每一种排列都能正常工作，但下一次洗澡时你肯定相当诧异)。

每当对代码进行了修改时(无论是添加新代码还是修复老代码)，都需要进行测试，以确保一切正常。

遗憾的是，永远无法确定已经抓住了每一个bug。如果测试过程中没有发现有错误的地方，那么这并不意味着没有bug，只是你并未发现它们。正如前辈程序员Edsger W. Dijkstra所言“测试显示的只是表面，并非没有错误”(这个问题可以成为哲学，如果某个bug未被发现，那么它还是bug吗？)

最好是测试并修复bug，直至bug出现在一个可接受的低概率范围。如果确实出现了bug，但并没有过于频繁和严重地影响到用户，那么就可以转向部署阶段。

示例：bug 计数

假定需求收集、概要设计、详细设计以及开发工作都是像这样：每次决策时，序列中的下一个任务中的两个以上的决策都要依赖上一个决策。例如，当制定一项需求决策时，概要设计中的两个决策都要依赖它(这并非它真正的工作方式，但并非像你可能期望的那样荒谬)。

现在假定我们在需求收集中犯了一个错误(客户说：应用程序必须支持30用户和5秒以内的响应时间，但我们听到的却是5用户和30秒以内的响应时间)。

如果在需求收集阶段就检测到了这个错误，则只须修复这一错误。如果直到部署完成以后都未发现此问题，那么有多少个不正确的决策将依赖上述的这个错误！

需求收集中的一个错误将导致概要设计中的两项决策都不正确。

概要设计中的两个可能的错误中的每一个，都将导致详细设计阶段两个新的决策错误，详细设计中总共将产生2×2＝4个可能的错误。

详细设计中4个可疑的错误中的每一个，都可能导致两个以上的开发错误，共产生4×2=8个开发错误。

加上需求收集、概要设计、详细设计以及开发阶段的所有错误，共计达1+2+4+8=15个可能的错误。图1-1显示了可能的错误传播情况。

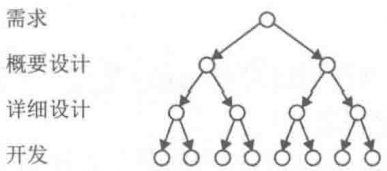

图1-1　圆圈表示不同开发阶段可能出现的错误。一个前期错误将导致许多后期错误

在上述示例中，如果已经及时发现需求收集中产生的决策错误，则找出、调试以及最终可能解决这些错误的决策是前者的15倍。这将引入软件工程的一个重要准则。这个准则非常重要，以至于本书的后续章节中还将出现：

错误仍未被发现的时间越长，就越难修复。

有人认为测试是在事后才去验证所编写的代码是否正确无误。事实上，为确保最终的应用程序可用，对于开发过程的每个阶段而言，测试都是至关重要的。

1.6　部署

理想情况下，客户们无不对即将推出的软件望眼欲穿。如果是开发一个改进版的“俄罗斯方块”游戏，并且需要在互联网上发行，那么其部署就可能非常简单。

然而，事情并非总是一帆风顺。部署可能会很困难，很费时，并且代价不菲。例如，假定已经编写了一个新的付费系统，用于跟踪某公司数百万个用户的付费情况，则这样的部署可能牵涉到以下情况：

- 用于后端数据库的一些新电脑。
- 一个新的网络。
- 用户的新电脑。
- 用户培训。
- 用户刚开始了解新系统时的现场支持。
- 并行操作(在部分用户开始了解新系统，而部分老用户仍在使用旧系统的过程中)。
- 专门的数据库维护工作(用于保持新旧数据库同步)。
- 大量的 bug 修复(当 250 个用户发现了几十或几百个测试阶段并未发现的 bug)。
- 未能预料的其他情况。

谁能预测？

我从事过一个电话公司调配维修人员解决客户问题的项目。在分配现场系统测试任务时，某人两次都被分配到了他前妻的家里。幸运的是，这位维修人员认出了这个地址，并且让他的主管重新分配了任务。

如果心理学能够应用到每一个软件项目的开发过程中，就有可能预料这些类型的古怪问题。如果做不到这一点或是不能预料，那么就应该在项目计划日程中预留一些额外的时间，用来处理此类完全不可预测的疑难杂症。

1.7 维护

如果用户开始使用软件，则他们将发现bug(这是另一个公理。测试者们未发现的bug将出现在现场用户开始使用应用程序之时)。

当然，只要用户发现了bug，就需要进行修复。正如前文所述，修复一个bug有时将引入另一个bug，因此现在同样需要修复它。

如果是一个用户经常使用的成功应用程序，则他们将更有可能发现程序中的bug。他们同样想对程序进行改进和提高，并且立即增加一些新的功能。

每一个开发者都希望遇到这样的情况：用户非常喜欢某个程序，他们的需求越来越多。这是每个软件工程项目的共同目标，但这意味着更多的工作量。

1.8 总结和反思

此时，可能要停下来休息一下。我们已经在计划、设计、开发以及测试上花费了太多时间。一些始料不及的bug已经被发现，用户的bug报告和变更请求让人应接不暇。我们最需要的莫过于一个愉快的长假。

在开始一次说走就走的旅行之前，还有一件很重要的事情要做——执行项目总结。我们需要对项目进行评估，看看哪些事情做对了，哪些做错了，从中找出那些成功的因素；相反，也要防止一些不利的因素继续恶化。

项目一旦完成，很多开发人员就不打算再进行这样的总结了。当所有人把和该项目有关的一些经验教训抛至脑后时，难能可贵的是你却能从中获益良多。

1.9 一次性处理所有事项

有人说“时间是一种让万物瞬间停止的自然力量”。遗憾的是，时间并没有以这种方式对待软件工程。

根据项目的规模和任务的分配方式，一些基本的任务将会重叠——有时将大规模重叠。

试想正在开发一个对国家安全利益至关重要的“航母式”应用程序。例如，完善全国能量饮料的订货、配送以及消费——这是一个很大的问题(的确如此)。可能你会有一些有关如何开始的想法，但为了构建可能的最佳解决方案，需要弄清很多细节。为了收集用户需求，可

能要花费相当长的一段时间来研究现有业务。

你可以自己花费数周的时间去调查用户需求，而让其他团队成员去玩《马里奥赛车》游戏，享用你正在研究的饮料，但这些都是非常低效的做法。

充分地利用每个人的时间，有利于发挥团队的整体优势，使项目更快地准备就绪。可以找几个人与客户一起进行需求分析，虽然这将花费更多的协调时间，但如果是大型项目，那么将可以节省很多时间。

如果认为已经理解了用户的很多需求，那么可以让其他团队成员开始概要设计。与等到需求分析完成之后才开始概要设计相比，他们可能出现更多的错误，但可以尽快把事情做完。

随着项目的不断推进，工作重心将逐渐转移到这些重点任务上。例如，当需求收集接近尾声，概要设计应该已经完成时，团队成员就可以转向详细设计，甚至是一些开发工作。

同时，在整个项目中，测试者们总是试图找出更多bug。当部分程序已经完成，就可以进行测试，以确保应用程序能处理各种不同的状况。

根据测试者的技术，甚至可以测试这些设计和需求。当然，不可能通过运行某个编译器来调试它们，但可以确定这些需求中未涉及的一些情况：假设一船能量饮料迟迟未到，但客户都在游轮上，刚刚跨过国际日期变更线。货物仍被认为是迟到吗？

有时，任务也将回滚。例如，开发过程中的错误可能回溯出一个设计甚至是需求问题。修正错误时回溯得越远，则带来的影响就越大。还记得前面那个每一个错误都将带来两个以上错误的示例吗？开发中发现的那些需求问题将带来大量其他未发现的bug。在最糟糕的情况下，对“已完成”代码的测试，可能会暴露出前期设计(甚至是需求收集阶段)中存在的一些根本性缺陷。

需求修复

我从事的第一个项目是一个海军特种部队(基本上就是海豹突击队)的库存系统。该系统可用来为各种活动定义作战装备包，然后让队员检查什么是必需的(有些类似童子军军需官所做的那种事情——对于此次露营而言，将需要帐篷、铺盖卷、水壶、炊具箱以及M79枪榴弹发射器)。

在开发该项目某个部分的过程中，我意识到，这些需求说明和概要设计，并未包含返回队员装备的任何方法(检查完装备后)。军需官的仓库在几周内都将是空的，而兵营里则到处可见堆积如山的各种军需装备。

这是一个规模相当小的项目，因此很容易进行修复。我告诉了项目经理，他仓促地设计一个库存返回窗口，然后我创建了它。这种类型的快速修复对于每一个项目并非都是可能的，尤其是一些大型项目，即使是在这种情况下，整个修复过程也花费了近一个小时的时间。

除了重叠和回退外，这些基本任务的处理方式也大相径庭。一些开发模型依赖极其详尽、严格的规范；而其他开发模型使用的规范则变幻莫测，有时让人感觉根本就没有使用任何规范。为使最终应用程序趋于完善，很多迭代方法甚至多次重复相同的任务。本书第II部分将对这些最常见的开发方法进行探讨。

1.10　本章小结

所有软件工程项目都必须处理一些相同的基本任务，但不同的开发模型处理它们的方式