第6章　充分利用软件工程方法

刚开始学习如何编写程序时，你可能有自己的安排。也许随意性很大，把工作放在最后一刻才做，也可能在实现过程中完全改变主意。不过，如果是专业性地编写代码，程序员很少会有这种灵活性。即使是思想最活跃的工程管理人员也承认，一定的过程管理很有必要。如今，了解软件工程过程与了解如何编写代码同样重要。

这一章会分析多种软件工程方法。在此不会对任何一种方法做非常深入的介绍，有关软件工程过程已经有许多非常好的书了。我们的想法是拓宽思路，介绍多种不同类型的过程，使你能够有所比较、有所对照。我们不会说哪一种方法特别好，也不会说哪一种方法特别不好。相反，希望通过不同方法之间的权衡，自己做出决定，建立一个适合自己和小组中其他成员的软件工程过程。

不论你是独立地开发项目，还是参与一个由来自各大洲数百位工程人员组成的团队，如果能理解软件开发的不同方法，将有助于日常的工作。

6.1　为什么需要过程

在软件开发的历史进程中，不乏项目惨败的事件发生。从预算超支且销售业绩很差的客户应用，到超大规模的操作系统，几乎没有哪个软件开发领域能逃过此劫。

即使软件确实到了用户手中，但由于 bug 太多，以至于最终用户不得不忍受不断的更新和补丁。有时软件并没有完成它本应完成的任务，或者并没有按用户期望的方式来完成。由这些问题可以清楚地指出软件界众所周知的一点：编写软件确实很难。

有人可能会感到奇怪，同样是工程，与其他形式的工程相比，为什么软件工程会这么频繁地失败呢？虽然汽车也存在问题，但你很少看到汽车会因为缓冲区溢出突然停下来，而要求重启动（不过，随着越来越多的汽车组件都由软件驱动，这种情况也是有可能的！）。举例来说，电视可能并非完美无缺，但是起码不必升级到 2.3 版本才能收到 6 频道。

这是不是说其他工程领域比软件工程更高级呢？城建工程人员能够从修建大桥的源远历史中汲取经验来建造一座合用的大桥。化学工程人员能够成功地合成一种化合物，因为已经通过前几代的试验解决了其中存在的问题。是不是这样呢？

> 软件（领域）是不是太新了，或者是不是它本身存在一些特性确实会导致 bug 的出现、无法使用的结果，以及招致项目失败？

看上去似乎与软件确实有所不同。一方面，在软件中，技术会迅速改变，这就带来软件开发过程的不确定性。即使在开发项目过程中没有遇到地震量级的突破，业界飞快前进的步伐还是会带来巨大的冲击。通常需要快速地开发软件，因为竞争太过激烈了。

软件开发也可能是不可预料的。要想做出精确的进度安排几乎是不可能的，因为仅仅一处小 bug 就可能需要花费数天甚至几个星期才能解决。即使一切似乎都按进度进行，但产品定义（产品需求）极有

可能改变，这就是功能扩张（feature creep），而这会严重影响甚至改变进程。

软件很复杂。没有一种容易而且准确的方法能够证明一个程序没有 bug。如果软件经过维护，发展了多个版本，倘若代码存在 bug 或者很混乱，就会对软件造成长时间（几年）的影响。软件系统通常非常复杂，这样当有人员调动时，对于粗心大意的人所留下的混乱代码，没有人乐意去靠近（接手）。而复杂的软件会导致一个无休止的循环往复：打补丁、修改，再解决问题。

当然，软件中还存在一般的企业风险。市场压力和交流不畅仍然存在。许多程序员想要理清公司的行政事务，但是由于市场压力的存在和交流不畅，开发小组与产品营销小组之间可能会出现争议，这种情况绝非少见。

软件工程产品所存在的这些负面因素都说明需要某种过程管理。软件项目规模很大、很复杂，而且前进的步伐很快。为了避免失败，工程小组需要采用一个系统来控制这种难处理的过程。

6.2 软件生命期模型

软件中的复杂性并不是一个新内容。早在数十年前人们就已经意识到形式化过程的必要性。已经提出了许多软件生命期（software life cycle）的建模方法，力图在软件开发的混乱局面中引入一些秩序，即按步骤来定义从最初提出软件想法到形成最终产品之间的软件过程。这些模型经过多年的改进，正指导着当前的许多软件开发。

6.2.1 分阶段模型和瀑布模型

软件的经典生命期模型一般称为分阶段模型或分步模型（Stagewise Model）。这个模型的基本思想是软件可以像照着菜谱一样来构建。相应地有一组步骤，如果严格地按照菜谱，就能得到一个很美味的巧克力蛋糕，类似地，如果确实按步骤进行，就能得到所期望的程序。下一阶段开始之前，前一个阶段必须先完成，如图 6-1 所示。

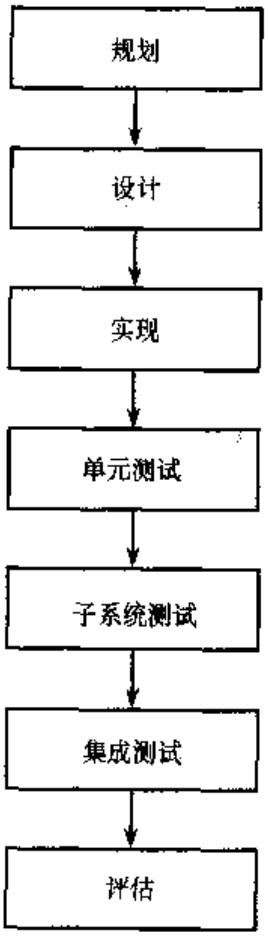

图 6-1

这个过程先从正式的规划开始，包括收集大量需求。这个需求表可以决定产品的功能是否完备。需求越具体，项目就越有可能成功。接下来，要设计并明确地指定软件。设计步骤与需求步骤类似，也要尽可能地具体，这样才更有可能成功。所有设计决策都在这一步做出，通常要提供伪代码，对于所需编写的特定子系统，还要给出相应的定义。编写子系统的人要明确其代码如何交互，开发小组要协商好体系结构的特定细节。接下来是对设计加以实现。由于已经明确地指定了设计，代码必须严格地遵循设计，否则各部分代码将无法协作。最后 4 个阶段则分别完成单元测试、子系统测试、集成测试和评估。

分阶段模型的主要问题在于，在实际中，一般至少需要对下一个阶段做些分析，否则在此之前完成前一个阶段几乎是不可能的。如果不写些代码，设计往往不能确定。另外，如果这个模型没有提供适当的途径返回编码阶段（即再对代码做些调整），那么测试还有什么意义呢？

对分阶段模型提出了许多改进，这些改进在 20 世纪 70 年代初提出的瀑布模型（Waterfall Model）中得到了形式化。这个模型仍然对当前的软件工程组织有很大的影响（甚至是主导性的影响）。瀑布模型所引入的主要改进是提出了阶段之间可以存在反馈。尽管它还是强调规划、设计、编码和测试这样一个严格的过程，

但是后面的阶段可以部分重叠。图 6-2 显示了瀑布模型的一个例子，在此可以看到反馈和重叠改进。基于反馈，在某个阶段获得的教训可以导致前一个阶段的修改。基于重叠，则允许两个阶段的行动可以同时发生。

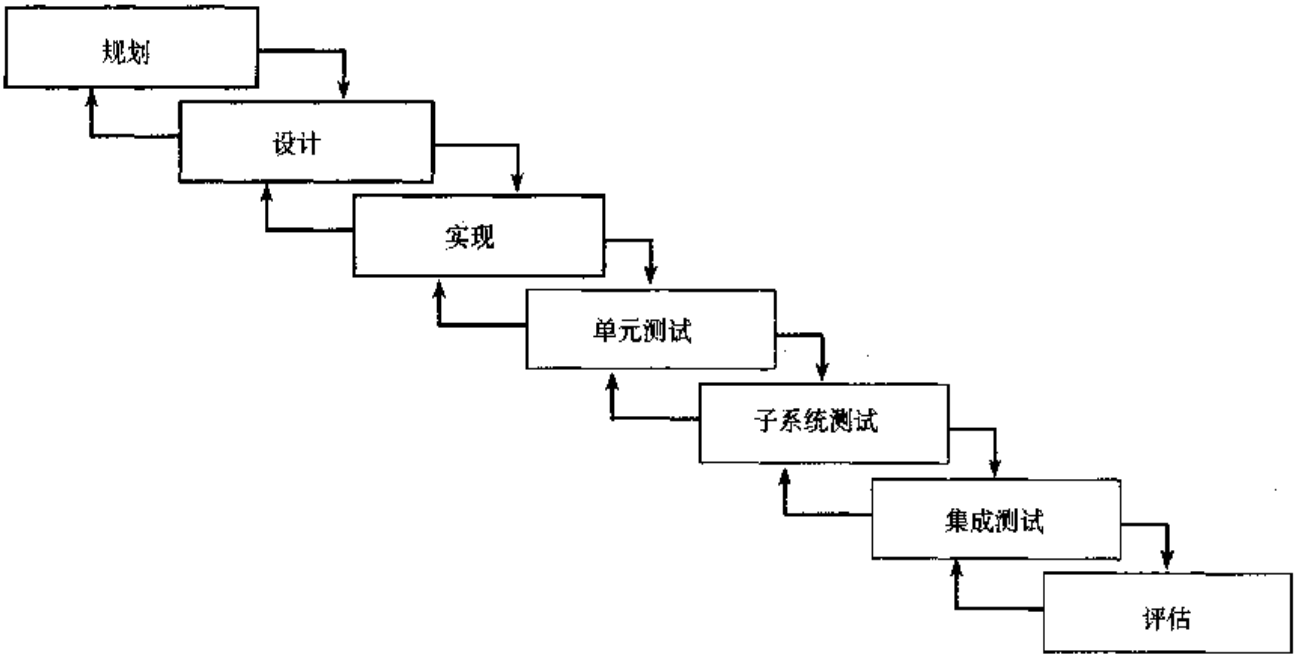

图　6-2

不同的瀑布方法会以不同的方式改进这个过程。例如，有些规划会包括一个“可行性”步骤，即在收集正式的需求之前，先完成一些可行性试验。

瀑布模型的好处

瀑布模型的价值在于它相当简单。实际上，你或管理人员在以往的项目中可能已经采用过这种方法，只不过不是那么正式，或者没有用这个名字而已。分步（分阶段）模型和瀑布模型有一个前提假设：只要每一步尽可能完全、尽可能准确地完成，后面的步骤就可以顺利地进行。只要在第一步仔细地指定了所有需求，而且第 2 步中做出了所有设计决策，并解决了所有问题，那么第 3 步中的实现就只是把设计翻译为代码而已。

由于瀑布模型的简单性，这就使得基于这种系统建立的项目规划很有组织性，而且很容易管理。使用这种方法的管理人员可能会提出要求，例如在设计阶段的最后，负责各子系统的所有工程人员要将其设计作为一份正式的设计文档或功能子系统规范提交上来。对于管理人员来说，其好处在于，由工程人员先行指定并完成设计，就能尽可能地降低风险。

从工程人员的角度看，采用瀑布方法可以提前明确主要问题。所有工程人员在编写大量代码之前都需要了解项目，并设计其子系统。理想情况下，这意味着代码可以只编写一次，而不是等发现各部分代码不能很好地协作时，再把各部分代码纠缠在一起进行修改或完全重写。

对于有特定需求的小型项目，瀑布方法能很好地工作。特别是在顾问领域，它的优点在于可以从项目一开始就明确可以用哪些特定的度量标准来衡量项目成功与否。将需求形式化，这有助于顾问准确地得出客户希望做什么，并要求客户具体地指出项目的目标。

瀑布模型的缺点

在许多组织中，以及在几乎所有软件工程方面的资料中，瀑布方法都不再得宠。瀑布方法认为软件

开发任务总在离散的线性步骤中完成，但有批评指出这个基本前提是不实际的。尽管瀑布方法允许阶段的重叠，但是并不允许较大幅度的后退。在当前的许多项目中，产品的整个开发过程中都可能出现新的需求。通常，顾客可能需要一个新特性，而且只有满足了这个特性，产品才卖得出去，或者是竞争对手的产品提供了一项新功能，所以你的产品中也需要有一个与之相当的功能。

> 提前指定所有需求，这使得瀑布方法对许多组织来说都是不可用的，因为这样就无法做到足够的灵活（动态）。

还有一个缺点是，为了尽可能降低风险，瀑布模型会尽可能正式而且尽早地做出决策，而这实际上可能只是把风险隐藏起来。例如，在设计阶段可能没有发现、遗漏、忘记或有意避开了一个主要的设计问题。到了集成测试阶段才发现存在这个问题，此时要想挽救这个项目可能已经为时过晚了。尽管已经暴露出了一个重大的设计缺陷，但是按照瀑布模型，并没有解决这个问题，而是带着它更进一步！瀑布过程中任何一个错误都很可能导致过程最后的失败。很难做早期检测，也很少这么做。

尽管瀑布模型仍然很常用，而且这仍是过程进行直观分析的一个有效手段，但是通常很有必要采纳其他方法的一些优点，让它更灵活一些。

6.2.2　螺旋方法

螺旋方法（Spiral Method）是 Barry W. Boehm 在 1988 年提出的，他认识到软件开发过程中可能出现未预料到的问题，而且需求可能发生改变，因此提出了这种方法。这个方法只是一组称为迭代过程（iterative process）技术中的一部分。其基本思想是，如果哪里出了问题，也没有关系，因为你会在下一轮里解决这个问题。图 6-3 显示了螺旋方法中的一轮（一次迭代）。

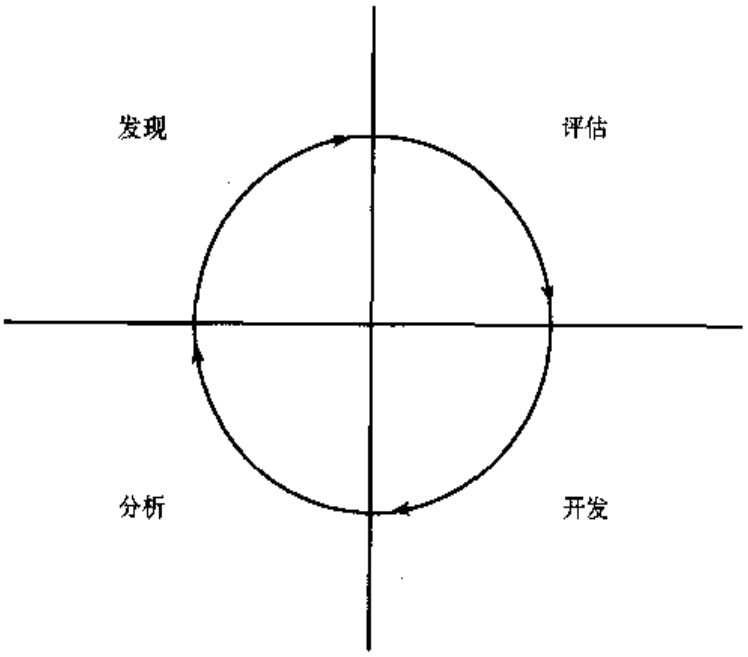

图　6-3

螺旋方法的各个阶段与瀑布方法中的各个步骤很相似。发现阶段包括建立需求，并确定目标。在评估阶段，会考虑多种实现选择，并且可能会建立原型。螺旋方法中，会特别重视评估阶段的风险评估和风险消除。螺旋的当前周期实现的是风险最大的任务。开发阶段的任务就由评估阶段明确的风险来确定。

例如，如果通过评估发现一个有问题的算法可能无法实现，那么当前周期的主要开发任务就是建立该算法模型、构建算法并测试。第 4 阶段用于完成分析和规划。基于当前周期的结果，可以建立后续周期的规划。每次迭代都要在较短的时间内完成，只考虑不多的几个关键特性和风险。

图 6-4 显示了采用螺旋方法开发一个操作系统的三个周期。第一个周期得到了一个包含产品主要需求的规划。第二个周期得到一个反映用户体验的原型。第三个周期建立一个组件，因为经认定这个任务的风险很高。

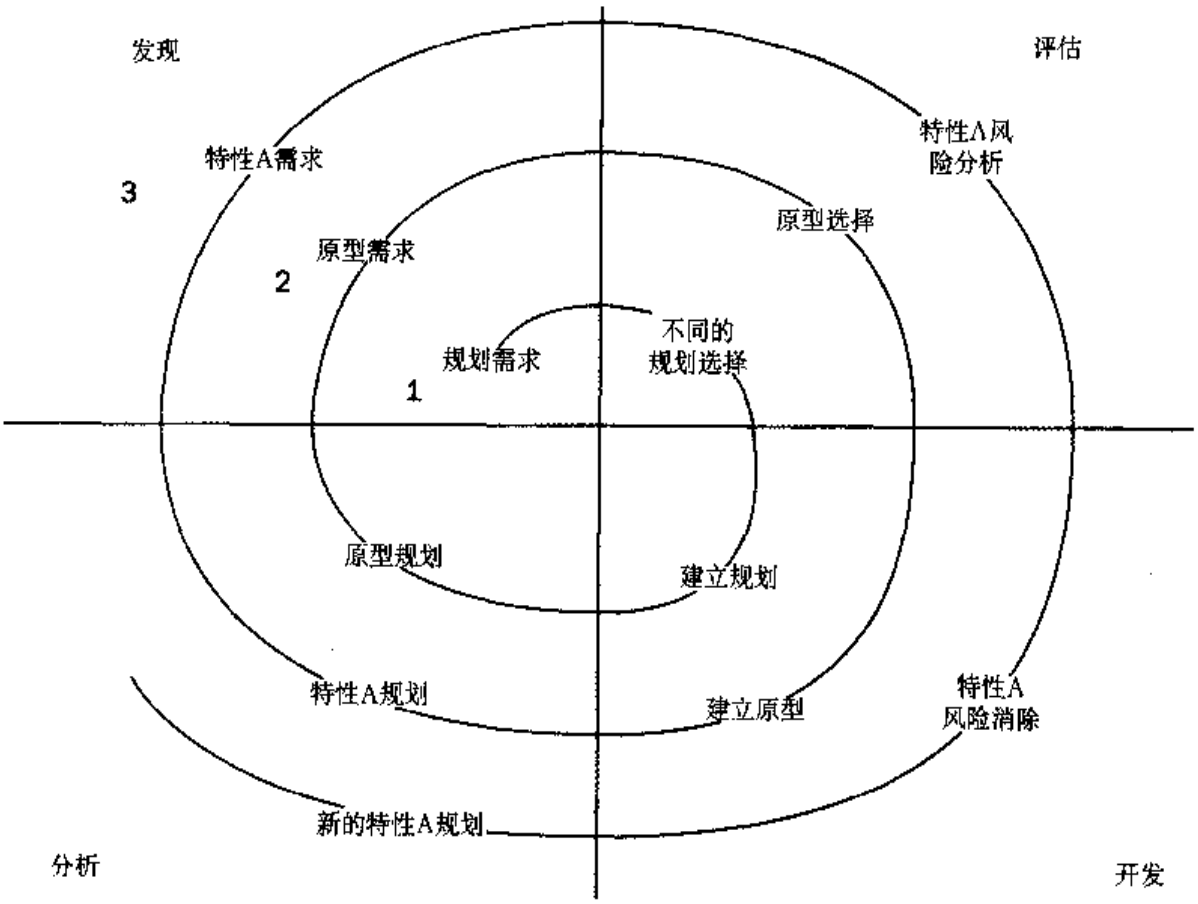

图　6-4

螺旋方法的好处

瀑布方法应当提供迭代，而螺旋方法可以看作是这种迭代方法的极致应用。图 6-5 将螺旋方法显示为一个瀑布过程，这个过程已经得到调整以允许迭代。通过这些为时不长的迭代周期，就可以消除瀑布方法存在的主要缺点（隐藏风险和线性开发路线）。

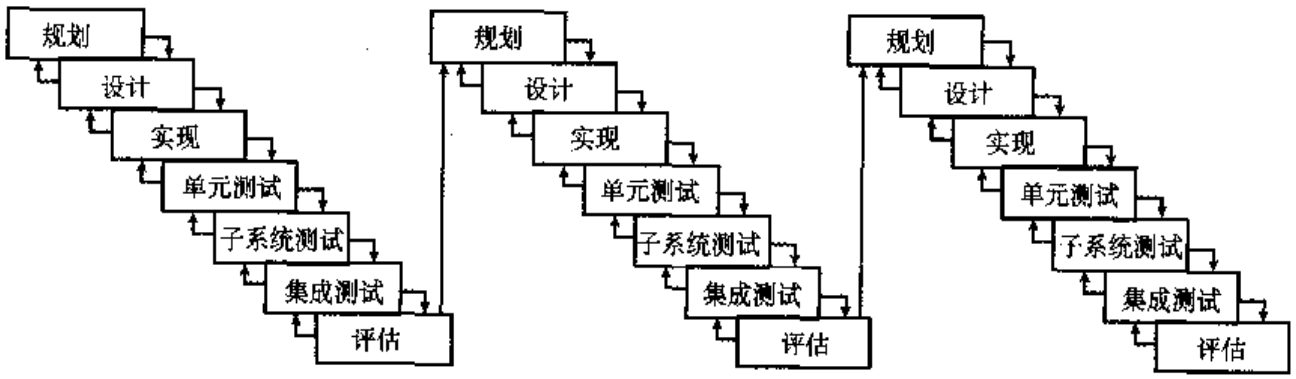

图　6-5

螺旋方法还有一个好处，即风险最大的任务最先完成。通过把风险放到前台，并认识到任何时刻都可能会有新的条件发生，螺旋方法就能避免瀑布模型中可能出现的隐藏的时间炸弹。出现未预料到的问题时，可以使用同样的4步方法来处理，这对余下的过程也同样适用。

最后一点，每个周期之后都进行分析并建立新的设计，通过这种重复做法，基本上可以消除“设计-然后实现”方法所存在的实际困难。通过每个周期，会对系统有更多了解，这些了解会进一步影响设计。

螺旋方法的缺点

螺旋方法的主要缺点在于，要将每次迭代界定得足够小，以便得到真正的收益，这往往很困难。在最坏的情况下，螺旋方法会蜕化为瀑布模型，因为一次迭代可能太长了。遗憾的是，螺旋方法只是对软件生命期建模。它无法指定一种具体的方法将项目分解为单周期的迭代，原因是这种划分会因项目不同而有所不同。

除此以来，螺旋方法还存在其他一些可能的缺点，这包括每个周期都要重复这4个阶段，这就会带来一定的开销，另外协调各个循环周期存在着难度。从逻辑上讲，可能很难在适当的时间将小组成员对设计的有关讨论汇总在一起。如果不同的小组在同时完成产品的不同部分，就有可能在并行的循环周期中工作，而这可能是不同步的。例如，开发用户界面的小组可能准备启动窗口管理器（Window Manager）周期的发现阶段，而核心操作系统（OS）小组还处在内存子系统的开发阶段。

6.2.3 统一开发过程

统一开发过程（Rational Unified Process，RUP）是管理软件开发过程的一种严格的形式化方法。RUP最重要的特征在于，不同于螺旋方法或瀑布模型，RUP不仅仅是一种理论性过程模型。RUP实际上是一个软件产品，由Rational Software（这是IBM的一个部门）推向市场。完全可以把这个过程看作是软件，主要有以下几个原因：

- 过程本身可以得到更新和改进，就像软件产品会周期性地更新一样。
- RUP并不只是推荐了一个开发框架，它还包括有一组软件工具，可与该框架结合使用。
- 作为一个产品，RUP可以推广到整个工程小组，这样所有成员都会采用完全相同的过程和工具。
- 与许多软件产品一样，RUP可以定制，以满足用户的需要。

RUP作为产品

作为一个产品，RUP的形式是一组软件应用，可在软件开发过程中为开发人员提供指导。RUP产品还可为其他Rational产品提供特殊的指导，如Rational Rose可视化建模工具和Rational ClearCase配置管理工具。另外RUP包括大量群件（groupware）通信工具，作为“思想市场”（marketplace of ideas）的一部分，开发人员可以利用这些工具实现知识共享。

RUP的一个基本原则是：开发周期的每次迭代都应当有一些有形的结果。在统一开发过程中，用户会建立许多设计、需求文档、报告和计划。RUP软件为建立这些结果提供了可视化工具和开发工具。

RUP作为过程

定义一个准确的模型，这是RUP的核心原则。按照RUP的说法，模型有助于解释软件开发过程中复杂的结构和关系。在RUP中，模型通常用统一建模语言（Unified Modeling Language，UML）的格式来表示。

RUP将过程的每个部分定义为一个单独的工作流（workflow）。工作流从以下几个方面来表示过程中的每一步，包括谁负责这一步，要完成哪些任务，这些任务的结果是什么，以及驱动任务的事件序列等。RUP的所有一切都是可定制的，不过RUP“预先”定义了一些核心过程工作流（core process workflow）。

核心过程工作流与瀑布模型中的阶段有一定的相似性，但是每个核心过程工作流都是迭代的，而且在定义上更为特定。*业务建模工作流*（business modeling workflow）是对业务过程建模，其目标通常是驱动软件需求向前发展。*需求工作流*（requirements workflow）通过分析系统中的问题，并描述其假设，从而得到需求定义。分析和设计工作流（analysis and design workflow）所处理的是系统体系结构和子系统设计。*实现工作流*（implementation workflow）涵盖了软件子系统的建模、编码和集成。*测试工作流*（test workflow）是对软件质量测试的规划、实现和评估建模。*部署工作流*（deployment workflow）是对整体规划、发布、运行和测试工作流的一个高层视图。*配置管理工作流*（configuration management workflow）涉及从新的项目概念出发，通过反复迭代，最终得到产品的过程。最后，*环境工作流*（environment workflow）通过创建和维护开发工具为工程组织提供支持。

RUP 实践

RUP 主要在较大规模的组织中采用，与传统的生命周期模型相比，RUP 有许多优点。一旦开发小组完成了学习曲线，了解了如何使用这个软件，所有成员都会使用一个共同的平台来设计、交流和实现其思想。这个过程可以进行定制，以满足小组的特定需求，而且每个阶段都能得到大量有价值的结果，这就是开发各个阶段的一系列文档。

对于某些组织来说，RUP 之类的产品可能太过庞大了。如果开发环境多样，或者工程预算很紧张，这样的开发小组可能不想或者无法对基于软件的开发系统实现标准化。另外，学习曲线也可能是一个影响因素，新的工程人员如果对过程软件不熟悉，就必须学习如何使用它，同时还要尽快地了解产品和现有的代码基（这样势必使新加入的工程人员手忙脚乱）。

6.3 软件工程方法论

软件生命期模型提供了一种形式化方法来回答这样一个问题“下一步要做什么”，紧接着我们可能会问“下一步该怎么做呢?”，但是软件生命期模型很少能够回答这个问题，但诸如 RUP 的形式化系统除外。要对“怎么做”之类的问题做出回答，为此已经开发了许多方法论，这些方法论可以为专业的软件开发提供许多实用的经验。有关软件方法论的书和文章数不胜数，不过最近有两个新方法特别值得关注，即*极限编程*（Extreme Programming）和*软件 Triage*（Software Triage）。

6.3.1 极限编程（XP）

几年前，本书作者之一下班回家时，告诉妻子他的公司采用了极限编程的某些原则。她开玩笑说“希望你能系上安全带”。尽管极限编程这个名字有些夸张，实际上它只是将既有的软件开发原则和新的原则捆绑在一起，而形成的一种越来越普及的全新方法论。

XP 因 Kent Beck 所著的《*eXtreme Programming eXplained*》（Addison-Wesley，1999）一书推广开来，极限编程主张采用优秀的软件开发的最佳实践，并将其上升到一个新的高度。例如，测试是很有意义的，这一点大多数程序员都同意。在极限编程中，测试更被看作是一个好东西，以至于甚至会在编写代码之前先编写测试。

XP 理论

极限编程方法论由 12 条主要的指导原则组成。这些原则在软件开发过程的所有阶段都有体现，而且对工程人员每天完成的任务有直接的影响。

根据要求做规划

在瀑布模型中，只会做一次规划，即在过程开始时规划。在螺旋方法中，规划则是每次迭代中的第一个阶段。在 RUP 中，规划是大多数工作流中少不了的一步。在极限编程中，规划不仅仅是一个步骤，

还是一个永不结束的任务。极限编程小组先从一个大致的计划开始，其中只抓住所开发产品的几个重点。在开发过程中，这个计划会根据需要得到改进和修改。这样做的道理是，条件会不断地改变，而且会一直得到新的信息。

在极限编程中，由谁实现特性，就应由他对这个特性做出估计。这有助于避免一些不切实际的情况，否则，实现人员可能被迫遵守一个不可行的安排。最初，估计可能相当粗略，也许只是以周为单位来估计实现一个特性（功能）需要多长时间。随着时间越来越短，估计的粒度也越来越细。特性会分解为小任务，完成这些小任务应不超过5天。

建立小的发布版本

极限编程的一个理论是，如果软件项目一次想要做太多工作，那么风险就会更大，也更为笨重。极限编程不鼓励过大的软件发布，这会涉及核心修改，而且发布说明可能长达数页，XP建议建立较小的发布版本，而且发布周期要接近两个月而不是长达18个月。由于发布周期这么短，只有最重要的特性才会放到产品中。这就要求工程部门和市场部门对于哪些特性确实重要达成一致。

采用共同的“隐喻”

XP使用隐喻（metaphor）一词，而其他方法论可能使用体系架构（architecture）来表示同样的概念。其思想是，小组的所有成员都应当有一个共同的系统高层视图。这不必是对象如何交互或者API如何编写之类的具体问题。相反，隐喻是系统组件的高层模型。小组成员应当使用隐喻，这样在讨论项目时可以采用相同的术语。

简化设计

热衷XP的工程人员常念叨的口头禅是“要避免过分的一般性”。这与许多程序员的一般想法有所背离。如果要完成一个任务，如设计一个基于文件的对象库，可能会从这样一条路着手：即为所有基于文件的存储问题提供一个终极解决方案。设计可能很快会演变为涵盖多种语言和各类对象。XP指出，应当向一般性的反方向倾斜。不要力图建立一个要夺大奖、受称赞的完美对象库，而应当设计最简单的对象库，只要能完成任务就行。你应当理解当前的需求，并编写代码来满足这些要求，而避免过于复杂的代码。

你可能不太习惯做简单的设计。取决于你做的是哪一类的工作，代码可能需要存在数年之久，而且可能会由代码的其他部分使用，对此你也许从未想到过。如第5章所述，如果要加入一些将来可能有用的功能，这就存在一个问题，你并不知道这些假想的用例是什么，而且很难得到一个好的设计。这就是一种过虑的情况。与此不同，XP指出，应当建立对当前有用的东西，但要留有余地以便以后再做修改。

不断地测试

《*eXtreme Programming eXplained*》中指出，“如果没有一个相应的自动化测试，实际上任何程序特性（功能）都不能认为真的存在”。根据这种说法，极限编程把测试摆到了一个很高的位置。作为一个XP工程人员，职责之一就是编写代码的相应单元测试。单元测试往往是一小段代码，用以确定某一段功能能够正常工作。例如，对于一个基于文件的对象库，单元测试可能包括testSaveObject、testLoadObject和testDeleteObject。

XP则将单元测试更向前推进一步，建议应当在编写具体代码之前编写单元测试。当然，由于代码尚未编写，因此不可能通过测试。从理论上讲，如果测试是全面的，就应该能知道什么时候代码才算完成，因为此时所有测试都会顺利通过。可以指出，这就是“极限”。

必要时重构

大多数程序员会不时地重构（refactor）他们的代码。重构是一个重新设计既有工作代码的过程，从而考虑到新得到的知识，或者编写代码后才发现的其他方法。在传统的软件工程进度安排中很难加入重构，因为重构的结果不像实现一个新特性那样具体。不过，好的管理人员都会认识到，重构对于代码长

期的可维护性是相当重要的。

极限重构是指，在开发过程中识别出哪些情况下重构是有用的，并相应地完成重构。并不是在发布之初就确定产品现有的哪些部分需要设计工作，而是由 XP 程序员去学习如何在开发过程中找出可以重构的代码。尽管这种做法几乎总是会带来未预计和未安排的任务，但是在适当的时候重构代码会使特性的开发更为容易。

配对编写代码

配对编程（pair programming）概念作为一种人际互动（touchy-feely）软件过程可以算是极限编程的一个特色。实际上，配对编程往往比想像中还要实用。XP 建议所有成品代码都应当同时由两个人协作编写。显然，只有一个人能操控键盘（即具体编写代码），另一个人则站在更高层上，考虑诸如测试、必要的重构以及项目整体模型等问题。

举例来说，如果你负责为应用中一个特定功能编写用户界面，可能希望请该功能的原作者坐在一旁帮助你。他能建议你如何正确地使用这个功能，警告你要提防哪些“陷阱”，还可以在一个高层次上对你的工作进行监督。即使你得不到原作者的帮助，小组中的另一个成员也能提供帮助。道理很简单，通过配对工作可以共享双方的知识，保证正确的设计，并引入一个非正式的检查和平衡体系。

共享代码

在许多传统的开发环境中，代码所有权往往相当明确，而且会带来一定的限制。本书作者之一就曾经在这样一个环境中工作过，其中管理人员明确地禁止大家修改小组中其他成员编写的代码。XP 则走了另一个极端，声明代码为所有人所共有。程序员之所以会携起手来，向“固步自封”说再见，XP 的这个方面可谓功不可没。实际上，这不太算是“人际互动”。

出于多种原因，共同所有权很实用。从管理的角度看，一个工程人员突然离开的话，危害不会太大，因为还有其他人理解他的那部分代码。从工程人员的角度看，基于共同所有权，可以对系统如何工作建立一个共同观点。这有助于设计任务，而免除了单个程序员的负担，因为不会要求他做出对整个项目都有意义的修改。

对于共同所有权，需要说明很重要的一点，无需每一位程序员都熟悉每一行代码。更正确的理解是，项目是整个团队的共同努力结果，没有必要让某一个人了解全部内容。

不断地集成

所有程序员都不会对“可怕的” 代码集成工作感到陌生。当你发现所看到的对象库并没有按原来编写的那样工作时，就要完成这个集成任务了。子系统聚在一起时，问题就会暴露出来。XP 认识到这种现象的存在，并鼓励在开发代码的过程中经常性地将代码集成到项目中。

XP 推荐了一种用于集成的特定方法。两个程序员（开发代码的配对）在指定的“集成点”完成代码的合并。在代码 100%地成功通过测试之前，此代码不能提交。由于有一个集成点，这可以避免冲突，而且集成可以清楚地定义为一个必须在提交之前出现的步骤。

我们还发现了一个类似的方法，采用这种方法，单个程序员也能完成集成。工程人员在把代码提交到代码库之前，要单独或配对地运行测试。由一台指定机器持续地运行自动化测试。当自动化测试失败时，开发小组会收到一个电子邮件，指出存在的问题，并列出最近提交的代码。

合理的工作时间

XP 对投入的时间也有涉及。据称，如果程序员休息得好，不仅心情愉快，而且效率更高。XP 建议一周工作大约 40 小时，并警告不要过度工作连续两周以上。

当然，不同的人需要多少休息存在差异。不过，重点是，如果你坐下来编写代码，但头脑不清醒，写出来的代码肯定不好，而且也会违反许多 XP 原则。

有一个顾客在场

由于崇尚XP的工程组会不断地改进其产品计划，而且只在产品中加入对当前而言必要的东西，因此如果有一个顾客能够参与会很有意义。尽管往往很难说服一个顾客真正在开发过程中一直在场，但是工程小组与最终用户之间应当保持交流，这一点无疑很重要。除了能够帮助设计各个特性之外，顾客还能根据各自的需要确定哪些任务更加优先。

共享共同的编码标准

由于存在共同所有权原则，而且实践中会采用配对编程方法，如果每个工程人员都有自己的一套命名和标识约定，那么在极限环境中编写代码就会很困难。XP并不特别推崇某一种风格，不过指出了一个指导原则：如果看一段代码的时候能够很容易地发现其作者是谁，开发小组就很有可能需要定义一种更好的编码标准。

XP实践

XP倡导者称，极限编程的12个原则彼此关联，如果采用其中的某些原则而不采用另外的一些原则，这会大大影响这个方法论的实施。例如，配对编程对测试就至关重要，因为如果你不能确定如何测试一段特定代码，你的同伴可以助一臂之力。另外，如果你哪一天太累，想要直接跳过测试，你的同伴则可以在一旁唤醒你的理智。不过，有些XP指导原则可能很难实现。对于某些工程人员来说，在编写代码之前编写测试，这种想法就很抽象。对他们而言，在有代码可供测试之前，只要设计出测试就足够了，此时不用具体编写测试。许多XP原则定义很严格，不过如果你了解基本理论，就能发现可以对这些原则稍做调整，来满足项目的需要。

XP的协作方面也很有难度。配对编程的好处可能很多，但是管理人员也许很难合理地安排一半人具体编写代码。小组中的有些成员可能不喜欢这种密切协作，在自己写代码时，有别人在旁边看着，可能会让他觉得不舒服。如果开发小组中的成员分布得很开，或者需要定期地远程通信，显然很难做到配对编程。

对于有些组织来说，极限编程可能太极端了。一些规模很大的公司对于工程开发往往有正式的条文，这些公司在采纳像XP之类的新方法时可能就很慢。不过，即使你的公司拒绝采用XP，如果能理解其基本原理，也能提高自己的工作效率。

6.3.2 软件triage

在一本名字很恐怖的书《*Death March*》(Prentice Hall, 2003)中，Edward Yourdon介绍了导致软件进度滞后、人员紧缺、预算超支或设计糟糕的一些常见的危险条件。Yourdon的理论称，软件项目如果处在这种状态，即便是当前最好的软件开发方法论也无济于事。在本章中你已经了解到，许多软件开发方法都是建立在形式化文档基础上的，或者采用一种以用户为中心的方法来设计。如果项目已经处在“死亡之旅“中，就没有时间、没有机会来实施这些方法了。

软件triage的基本思想是，如果一个项目已经处在很糟糕的状态下，资源就会紧缩。不仅时间不够，工程人员不够，资金也很短缺。当项目进度滞后时，管理人员和开发人员需要克服的一个最大障碍是：可能无法在预定的时间内满足最初提出的需求。原本要完成的任务只好简单地列在“必须有的功能”、“应当有的功能”和“最好有的功能”列表中，等待以后实现。

软件triage是一个麻烦而且细致的过程。通常要求经历过“死亡之旅”项目的有经验的老手指导，并做出艰难的抉择。对工程人员来说，最重要的一点是，某些情况下可能需要将我们熟悉的一些过程抛在一边（遗憾的是，这样可能也会丢掉一些原有的代码），以便项目如期完工。

6.4 建立自己的过程和方法论

肯定有一种软件开发方法论是我们全心拥护的，而这不一定非得是上述的某一种方法论。不管是哪

一本书或哪一种工程理论，一般不太可能刚好满足你的项目或组织的需要。建议你尽可能多地了解多种方法，并设计自己的过程。结合不同方法中的概念要比完全让自己去想容易得多。例如，RUP 可以支持一种类 XP 的方法。以下是建立自己的软件工程过程的一些提示。

6.4.1　以开放的心态接纳新思想

有些工程技术乍看上去可能很不可思议，或者好像不可行。要把软件工程方法论中的创新看作是改进现有过程的一个途径。在可能的情况下要尽量尝试。如果极限编程听上去很有趣，但不能保证它确实能在你所在组织中奏效，就可以慢慢地看它是否可行，一次采纳几个原则，或者在一个较小的预研项目中先做尝试。

6.4.2　汇总新思想

更常见的情况下，工程小组由来自各个不同背景的人所组成。这些人可能包括资深的老手，经验丰富的顾问，刚刚毕业的研究生以及一些博士。每个人都有不同的经验，而且对于软件项目应当如何运行也都有自己的一套想法。有时，通过将适用于各个不同环境的工作方式加以结合，往往能得到最佳的过程。

6.4.3　明确哪些可行，哪些不可行

在项目最后（或者更好情况下，可以在项目进行过程中），要把小组召集在一起对过程做出评估。有时只有当整个小组停下当前的工作，共同思考一个问题，才会发现重要的问题。也许每个人都知道一个问题的存在，但是从未有人提起过！要考虑哪些是不可行的，并明确这些部分能否修正。有些组织在提交任何源代码之前需要正式的代码审查。如果代码审查需要太长时间，太过枯燥，那么谁的工作也做不好，大家应当在一起讨论代码审查技术。另外，还要考虑哪些方面工作得很好，并了解这些部分如何扩展。例如，可以建立一个网站来维护各种任务（小组能够编辑这些任务），这种想法如果可行，那么可以多花一些时间把这个网站建得更好。

6.4.4　不要做叛逃者

不论过程是由管理人员控制，还是由小组自行建立，总有它存在的原因。如果你的过程涉及编写正式的设计文档，就要确保文档由你编写。如果你认为这个过程很糟糕，或者太过复杂，可以看看是不是能与管理人员做些讨论。不要一味地躲避这个过程，它还是会“找上门来”的。

6.5　小结

本章介绍了软件过程的一些模型和方法论。建立软件还有许多其他的方法，这包括一些正式和非正式的方法。开发软件时，可能没有一种完全正确的方法，而只有最适合你的方法。发现这种方法的最佳途径是自己做研究，尽可能地从多种不同方法中学习，向别人讨教他们在使用各种方法时的经验，并迭代地完成过程。要记住，分析一个过程方法论时，有一个真正有意义的标准，这就是看这种方法论对你的小组编写代码有多大的帮助。

到本章为止，本书第一部分就结束了，这一部分所讨论的都是有关软件设计方面的内容。你了解了如何设计一个程序，如何组织对象关系，如何利用既有的模式和库，如何与别人一同有效地编写代码，如何管理软件开发过程等等。在本书余下的部分中，你学到的这些设计原则将与 C++紧密结合。下一部分将深入到细节当中，介绍如何用 C++编写专业质量的代码。在深入到书中有关编写代码的部分中时，不要忘记了前面几章学到的设计原则。我们之所以把有关设计的章节放在最前面，就是想要突出强调它们的重要性。