

## 第八章 维护

在软件产品被开发出来并交付用户使用之后，就进入了软件的运行维护阶段。这个阶段是软件生命周期的最后一个阶段，其基本任务是保证软件在一个相当长的时期能够正常运行。

软件维护需要的工作量很大，平均说来，大型软件的维护成本高达开发成本的4倍左右。目前国外许多软件开发组织把60%以上的人力用于维护已有的软件，而且随着软件数量增多和使用寿命延长，这个百分比还在持续上升。将来维护工作甚至可能会束缚住软件开发组织的手脚，使他们没有余力开发新的软件。

软件工程的主要目的就是要提高软件的可维护性，减少软件维护所需要的工作量，降低软件系统的总成本。

### 8.1 软件维护的定义

所谓软件维护就是在软件已经交付使用之后，为了改正错误或满足新的需要而修改软件的过程。可以通过描述软件交付使用后可能进行的4项活动，具体地定义软件维护。因为软件测试不可能暴露出一个大型软件系统中所有潜藏的错误，所以必然会有第一项维护活动：在任何大型程序的使用期间，用户必然会发现程序错误，并且把他们遇到的问题报告给维护人员。把诊断和改正错误的过程称为改正性维护。

计算机科学技术领域的各个方面都在迅速进步，大约每过36个月就有新一代的硬件宣告出现，经常推出新操作系统或旧系统的修改版本，时常增加或修改外部设备和其他系统部件；另一方面，应用软件的使用寿命却很容易超过10年，远远长于最初开发这个软件时的运行环境的寿命。

因此，适应性维护，也就是为了和变化了的环境适当地配合而进行的修改软件的活动，是既必要又经常的维护活动。

当一个软件系统顺利地运行时，常常出现第三项维护活动：在使用软件的过程中用户往往提出增加新功能或修改已有功能的建议，还可能提出一般性的改进意见。为了满足这类要求，需要进行完善性维护。这项维护活动通常占软件维护工作的大部分。

当为了改进未来的可维护性或可靠性，或为了给未来的改进奠定更好的基础而修改软件时，出现了第四项维护活动。这项维护活动通常称为预防性维护，目前这项维护活动相对比较少。

从上述关于软件维护的定义不难看出，软件维护绝不仅限于纠正使用中发现的错误事实上在全部维护活动中一半以上是完善性维护。国外的统计数字表明，完善性维护占全部维护活动的50%~66%，改正性维护占17%~21%，适应性维护占18%—25%，其他维护活动只占4%左右。

应该注意，上述4类维护活动都必须应用于整个软件配置，维护软件文档和维护软件的可执行代码是同样重要的。

## 8.2 软件维护的特点

### 8.2.1 结构化维护与非结构化维护差别巨大

#### 1. 非结构化维护

如果软件配置的惟一成分是程序代码，那么维护活动从艰苦地评价程序代码开始，而且常常由于程序内部文档不足而使评价更困难，对于软件结构、全程数据结构、系统接口、性能和(或)设计约束等经常会产生误解，而且对程序代码所做的改动的后果也是难于估量的：因为没有测试方面的文档，所以不可能进行回归测试(即指为了保证所做的修改没有在以前可以正常使用的软件功能中引入错误而重复过去做过的测试)。非结构化维护需要付出很大代价(浪费精力并且遭受挫折的打击)，这种维护方式是没有使用良好定义的方法学开发出来的软件的必然结果。

#### 2. 结构化维护

如果有一个完整的软件配置存在，那么维护工作从评价设计文档开始，确定软件重要的结构特点、性能特点以及接口特点；估量要求的改动将带来的影响，并且计划实施途径。然后首先修改设计并且对所做的修改进行仔细复查。接下来编写相应的源程序代码；使用在测试说明书中包含的信息进行回归测试；最后，把修改后的软件再次交付使用。

刚才描述的事件构成结构化维护，它是在软件开发的早期应用软件工程方法学的结果。虽然有了软件的完整配置并不能保证维护中没有问题，但是确实能减少精力的浪费并且能提高维护的总体质量。

### 8.2.2 维护的代价高昂

在过去的几十年中，软件维护的费用稳步上升。1970年用于维护已有软件的费用只占软件总预算的35%~40%，1980年上升为40%~60%，1990年上升为70%~90%。

维护费用只不过是软件维护的最明显的代价，其他一些现在还不明显的代价将来可能更为人们所关注。因为可用的资源必须供维护任务使用，以致耽误甚至丧失了开发的良机，这是软件维护的一个无形的代价。其他无形的代价还有：

- 当看来合理的有关改错或修改的要求不能及时满足时将引起用户不满；
- 由于维护时的改动，在软件中引入了潜伏的错误，从而降低了软件的质量；
- 当必须把软件工程师调去从事维护工作时，将在开发过程中造成混乱。

软件维护的最后一个代价是生产率的大幅度下降，这种情况在维护旧程序时常常遇到。例如，据 Gausler 在 1976 年的报道，美国空军的飞行控制软件每条指令的开发成本是 75 美元，然而维护成本大约是每条指令 4 000 美元，也就是说，生产率下降了 50 倍以上。

用于维护工作的劳动可以分成生产性活动(例如, 分析评价, 修改设计和编写程序代码等)和非生产性活动(例如, 理解程序代码的功能, 解释数据结构、接口特点和性能限度等)。下述表达式给出维护工作量的一个模型:

$$M = P + K \times \exp(c - d)$$

其中:  $M$  是维护用的总工作量,  $P$  是生产性工作量,  $K$  是经验常数,  $c$  是复杂程度(非结构化设计和缺少文档都会增加软件的复杂程度),  $d$  是维护人员对软件的熟悉程度。

上面的模型表明, 如果软件的开发途径不好(即, 没有使用软件工程方法学), 而且原来的开发人员不能参加维护工作, 那么维护工作量和费用将指数地增加。

## 8. 2. 3 维护的问题很多

与软件维护有关的绝大多数问题, 都可归因于软件定义和软件开发的方法有缺点。在软件生命周期的头两个时期没有严格而又科学的管理和规划, 几乎必然会导致在最后阶段出现问题。下面列出和软件维护有关的部分问题:

(1)理解别人写的程序通常非常困难, 而且困难程度随着软件配置成分的减少而迅速增加。如果仅有程序代码没有说明文档, 则会出现严重的问题。

(2)需要维护的软件往往没有合格的文档, 或者文档资料显著不足。认识到软件必须有文档仅仅是第一步, 容易理解的并且和程序代码完全一致的文档才真正有价值。

(3)当要求对软件进行维护时, 不能指望由开发人员给我们仔细说明软件。由于维护阶段持续的时间很长, 因此, 当需要解释软件时, 往往原来写程序的人已经不在附近了。

(4)绝大多数软件在设计时没有考虑将来的修改。除非使用强调模块独立原理的设计方法学, 否则修改软件既困难又容易发生差错。

(5)软件维护不是一项吸引人的工作。形成这种观念很大程度上是因为维护工作经常遭受挫折。

上述种种问题在现有的没采用软件工程思想开发出来的软件中, 都或多或少地存在着。不应该把一种科学的方法学看做万应灵药, 但是, 软件工程至少部分地解决了与维护有关的每一个问题。

## 8. 3 软件维护过程

维护过程本质上是修改和压缩了的软件定义和开发过程, 而且事实上远在提出一项维护要求之前, 与软件维护有关的工作已经开始了。首先必须建立一个维护组织, 随后必须确定报告和评价的过程, 而且必须为每个维护要求规定一个标准化的事件序列。此外, 还应该建立一个适用于维护活动的记录保管过程, 并且规定复审标准。

### 1. 维护组织

虽然通常并不需要建立正式的维护组织, 但是, 即使对于一个小的软件开发团体而言, 非正式地委托责任也是绝对必要的。每个维护要求都通过维护管理员转交给相应的系统管理员去评价。系统管理员是被指定去熟悉一小部分产品程序的技术人员。系统管理员对维护任务做出评价之后, 由变化授权人决定应该进行的活动。图 8. 1 描绘了上述组织

方式。

在维护活动开始之前就明确维护责任是十分必要的，这样做可以大大减少维护过程中可能出现的混乱。

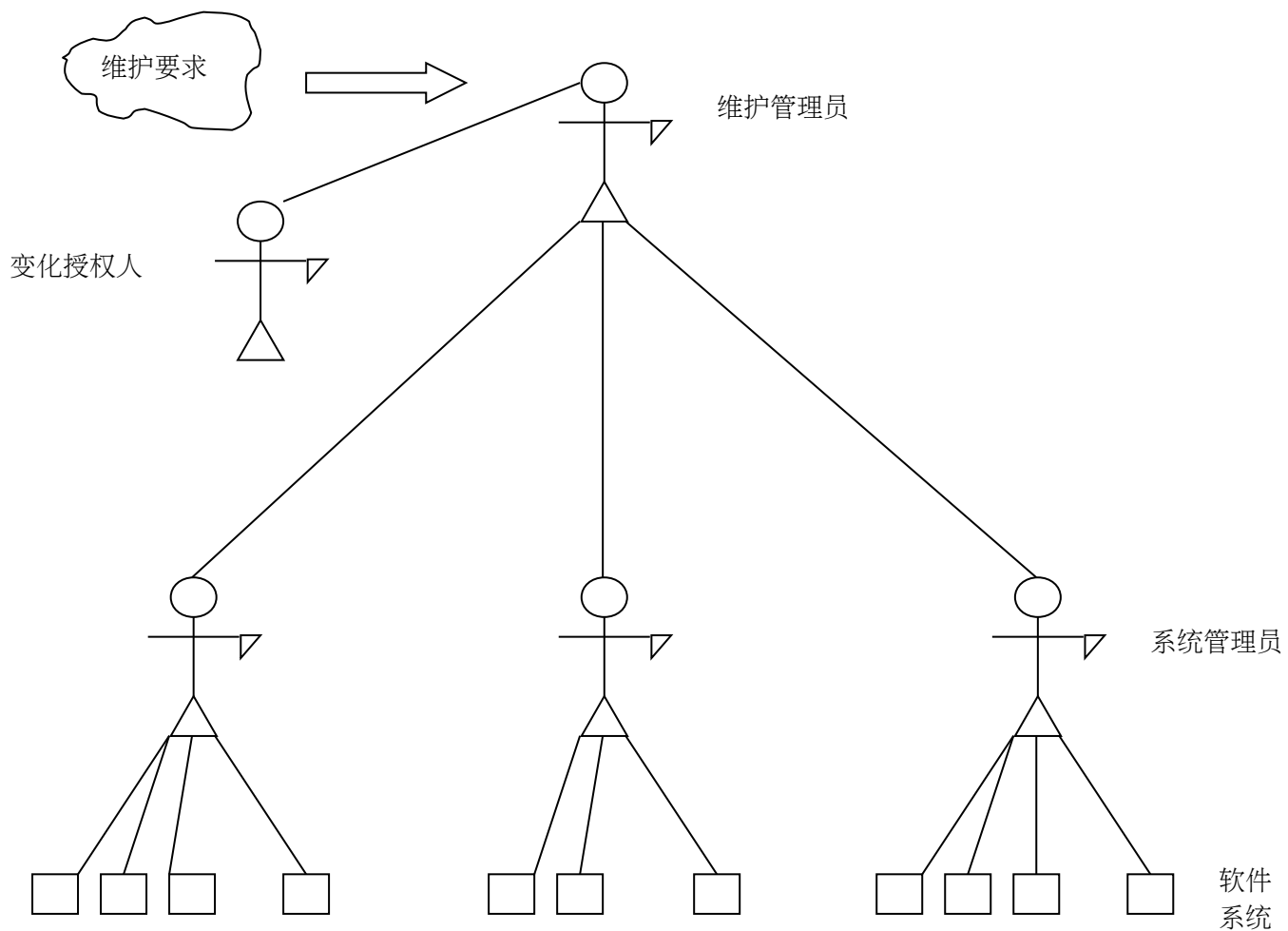


图 8.1 维护组织

## 2. 维护报告

应该用标准化的格式表达所有软件维护要求。软件维护人员通常给用户提供空白的维护要求表——有时称为软件问题报告表，这个表格由要求一项维护活动的用户填写。如果遇到一个错误，那么必须完整描述导致出现错误的环境(包括输入数据、全部输出数据以及其他有关信息)。对于适应性或完善性的维护要求，应该提出一个简短的需求说明书。如前所述，由维护管理员和系统管理员评价用户提交的维护要求表。

维护要求表是一个外部产生的文件，它是计划维护活动的基础。软件组织内部应该制定出一个软件修改报告，它给出下述信息：

- (1)满足维护要求表中提出的要求所需要的工作量；
- (2)维护要求的性质；
- (3)这项要求的优先次序；
- (4)与修改有关的事后数据。

在拟定进一步的维护计划之前，把软件修改报告提交给变化授权人审查批准。

## 3. 维护的事件流

图 8. 2 描绘了由一项维护要求而引起的一串事件。首先应该确定要求进行的维护的类型。用户常常把一项要求看作是为了改正软件的错误(改正性维护)，而开发人员可能把同一项要求看作是适应性或完善性维护。当存在不同意见时必须协商解决。

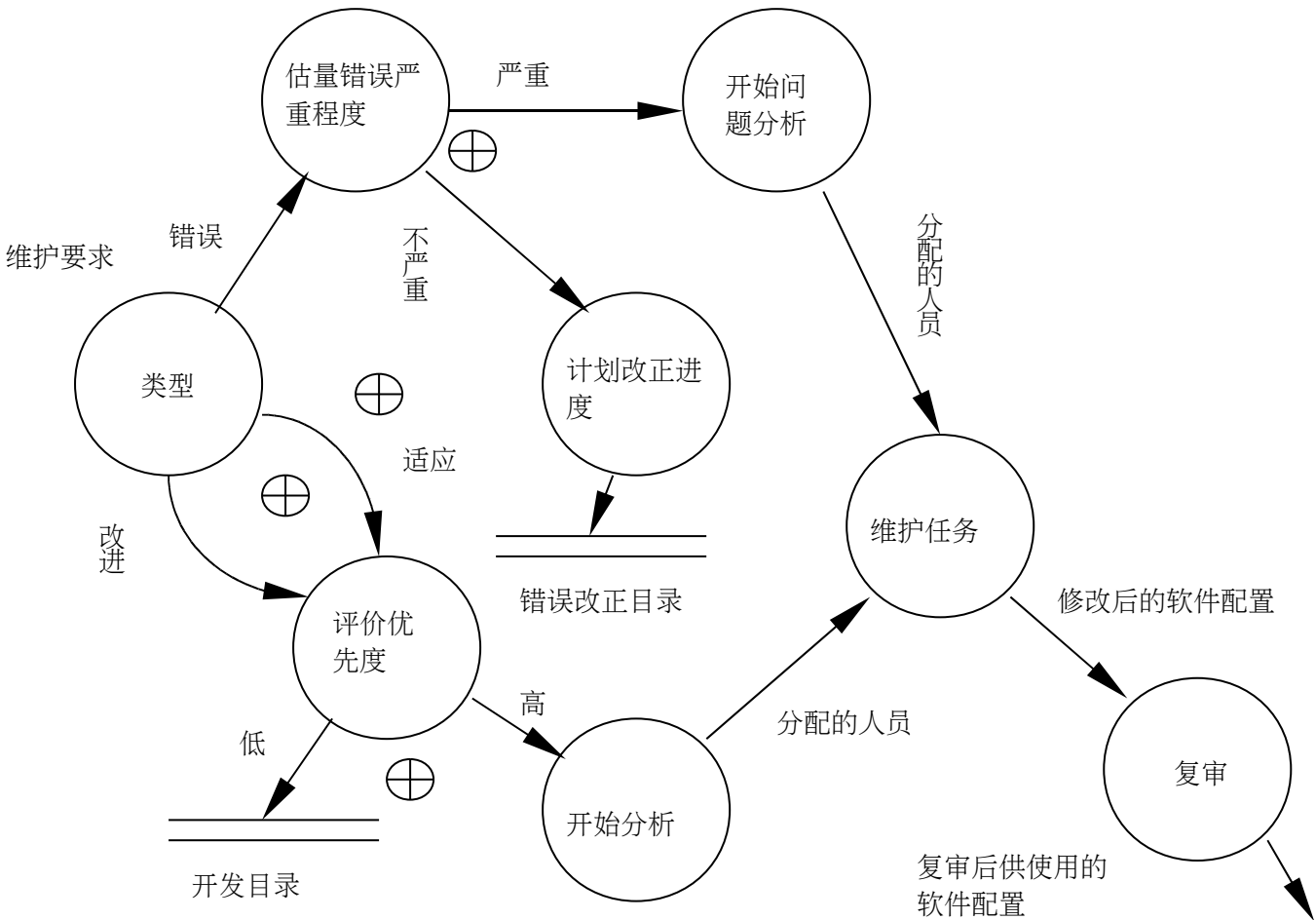


图 8.2 维护阶段的事件流

从图 8.2 描绘的事件流看到,对一项改正性维护要求(图中“错误”通路)的处理,从估量错误的严重程度开始。如果是一个严重的错误(例如,一个关键性的系统不能正常运行),则在系统管理员的指导下分派人员,并且立即开始问题分析过程。如果错误并不严重,那么改正性的维护和其他要求软件开发资源的任务一起统筹安排。

适应性维护和完善性维护的要求沿着相同的事件流通路前进。应该确定每个维护要求的优先次序,并且安排要求的工作时间,就好像它是另一个开发任务一样(从所有意图和目标来看,它都属于开发工作)。如果一项维护要求的优先次序非常高,可能立即开始维护工作。

不管维护类型如何,都需要进行同样的技术工作。这些工作包括修改软件设计、复查、必要的代码修改、单元测试和集成测试(包括使用以前的测试方案的回归测试)、验收测试和复审。不同类型的维护强调的重点不同,但是基本途径是相同的。维护事件流中最后一个事件是复审,它再次检验软件配置的所有成分的有效性,并且保证事实上满足了维护要求表中的要求。

当然,也有并不完全符合上述事件流的维护要求。当发生恶性的软件问题时,就出现所谓的“救火”维护要求,这种情况需要立即把资源用来解决问题。如果对一个组织来说,“救火”是常见的过程,那么必须怀疑它的管理能力和技术能力。

在完成软件维护任务之后,进行处境复查常常是有好处的。一般说来,这种复查试图回答下述问题:

- 在当前处境下设计、编码或测试的哪些方面能用不同方法进行?
- 哪些维护资源是应该有而事实上却没有的?
- 对于这项维护工作什么是主要的(以及次要的)障碍?
- 要求的维护类型中有预防性维护吗?

处境复查对将来维护工作的进行有重要影响,而且所提供的反馈信息对有效地管理软件组织十分重要。

## 4. 保存维护记录

对于软件生命周期的所有阶段而言,以前记录保存都是不充分的,而软件维护则根本没有记录保存下来。由于这个原因,往往不能估价维护技术的有效性,不能确定一个产品程序的“优良”程度,而且很难确定维护的实际代价是什么。

保存维护记录遇到的第一个问题就是,哪些数据是值得记录的?Swanson 提出了下述内容:①程序标识;②源语句数;③机器指令条数;④使用的程序设计语言;⑤程序安装的日期;⑥自从安装以来程序运行的次数;⑦自从安装以来程序失效的次数;⑧程序变动的层次和标识;⑨因程序变动而增加的源语句数;⑩因程序变动而删除的源语句数;⑪每个改动耗费的人时数;⑫程序改动的日期;⑬软件工程师的名字;⑭维护要求表的标识;⑮维护类型;⑯维护开始和完成的日期;⑰累计用于维护的人时数;⑱与完成的维护相联系的纯效益。

应该为每项维护工作都收集上述数据。可以利用这些数据构成一个维护数据库的基础，并且像下面介绍的那样对它们进行评价。

## 5. 评价维护活动

缺乏有效的数据就无法评价维护活动。如果已经开始保存维护记录了，则可以对维护工作做一些定量度量。至少可以从下述 7 个方面度量维护工作：

- (1)每次程序运行平均失效的次数；
- (2)用于每一类维护活动的总人时数；
- (3)平均每个程序、每种语言、每种维护类型所做的程序变动数；
- (4)维护过程中增加或删除一个源语句平均花费的人时数；
- (5)维护每种语言平均花费的人时数；
- (6)一张维护要求表的平均周转时间；
- (7)不同维护类型所占的百分比。

根据对维护工作定量度量的结果，可以做出关于开发技术、语言选择、维护工作量规划、资源分配及其他许多方面的决定，而且可以利用这样的数据去分析评价维护任务。

## 8.4 软件的可维护性

可以把软件的可维护性定性地定义为：维护人员理解、改正、改动或改进这个软件的难易程度。在前面的章节中曾经多次强调，提高可维护性是支配软件工程方法学所有步骤的关键目标。

### 8.4.1 决定软件可维护性的因素

维护就是在软件交付使用后进行的修改，修改之前必须理解待修改的对象，修改之后应该进行必要的测试，以保证所做的修改是正确的。如果是改正性维护，还必须预先进行调试以确定错误的具体位置。因此，决定软件可维护性的因素主要有下述 5 个：

#### 1. 可理解性

软件可理解性表现为外来读者理解软件的结构、功能、接口和内部处理过程的难易程度。模块化(模块结构良好，高内聚，松耦合)、详细的设计文档、结构化设计、程序内部的文档和良好的高级程序设计语言等等，都对提高软件的可理解性有重要贡献。

#### 2. 可测试性

诊断和测试的容易程度取决于软件容易理解的程度。良好的文档对诊断和测试是至关



重要的，此外，软件结构、可用的测试工具和调试工具，以及以前设计的测试过程也都是非常重要的。维护人员应该能够得到在开发阶段用过的测试方案，以便进行回归测试。在设计阶段应该尽力把软件设计成容易测试和容易诊断的。

对于程序模块来说，可以用程序复杂度来度量它的可测试性。模块的环形复杂度越大，可执行的路径就越多，因此，全面测试它的难度就越高。

### 3. 可修改性

软件容易修改的程度和本书第5章讲过的设计原理和启发规则直接有关。耦合、内聚、信息隐藏、局部化、控制域与作用域的关系等等，都影响软件的可修改性。

### 4. 可移植性

软件可移植性指的是，把程序从一种计算环境(硬件配置和操作系统)转移到另一种计算环境的难易程度。把与硬件、操作系统以及其他外部设备有关的程序代码集中放到特定的程序模块中，可以把因环境变化而必须修改的程序局限在少数程序模块中，从而降低修改的难度。

### 5. 可重用性

所谓重用(reuse)是指同一事物不做修改或稍加改动就在不同环境中多次重复使用。大量使用可重用的软件构件来开发软件，可以从下述两个方面提高软件的可维护性：

(1)通常，可重用的软件构件在开发时经过很严格的测试，可靠性比较高，且在每次重用过程中都会发现并清除一些错误，随着时间推移，这样的构件将变成实质上无错误的。因此，软件中使用的可重用构件越多，软件的可靠性越高，改正性维护需求越少。

(2)很容易修改可重用的软件构件使之再次应用在新环境中，因此，软件中使用的可重用构件越多，适应性和完善性维护也就越容易。

## 8. 4. 2 文档

文档是影响软件可维护性的决定因素。由于长期使用的大型软件系统在使用过程中必然会经受多次修改，所以文档比程序代码更重要。

软件系统的文档可以分为用户文档和系统文档两类。用户文档主要描述系统功能和使用方法，并不关心这些功能是怎样实现的；系统文档描述系统设计、实现和测试等各方面的内容。

总的说来，软件文档应该满足下述要求：

- (1)必须描述如何使用这个系统，没有这种描述时即使是最简单的系统也无法使用；
- (2)必须描述怎样安装和管理这个系统；
- (3)必须描述系统需求和设计；
- (4)必须描述系统的实现和测试，以便使系统成为可维护的。

下面分别讨论用户文档和系统文档。

## 1. 用户文档

用户文档是用户了解系统的第一步，它应该能使用户获得对系统的准确的初步印象。文档的结构方式应该使用户能够方便地根据需要阅读有关的内容。

用户文档至少应该包括下述 5 方面的内容：

- (1)功能描述，说明系统能做什么；
- (2)安装文档，说明怎样安装这个系统以及怎样使系统适应特定的硬件配置；
- (3)使用手册，简要说明如何着手使用这个系统(应该通过丰富例子说明怎样使用常用的系统功能，还应该说明用户操作错误时怎样恢复和重新启动)；
- (4)参考手册，详尽描述用户可以使用的所有系统设施以及它们的使用方法，还应该解释系统可能产生的各种出错信息的含义(对参考手册最主要的要求是完整，因此通常使用形式化的描述技术)；
- (5)操作员指南(如果有系统操作员的话)，说明操作员应该如何处理使用中出现的各种情况。

上述内容可以分别作为独立的文档，也可以作为一个文档的不同分册，具体做法应该由系统规模决定。

## 2. 系统文档

所谓系统文档指从问题定义、需求说明到验收测试计划这样一系列和系统实现有关的文档。描述系统设计、实现和测试的文档对于理解程序和维护程序来说是极端重要的。和用户文档类似，系统文档的结构也应该能把读者从对系统概貌的了解，引导到对系统每个方面每个特点的更形式化更具体的认识。本书前面各章已经较详细地介绍了各个阶段应该产生的文档，此处不再重复。

### 8. 4. 3 可维护性复审

可维护性是所有软件都应该具备的基本特点，必须在开发阶段保证软件具有 8. 4. 1 节中提到的那些可维护因素。在软件工程过程的每一个阶段都应该考虑并努力提高软件的可维护性，在每个阶段结束前的技术审查和管理复审中，应该着重对可维护性进行复审。

在需求分析阶段的复审过程中，应该对将来要改进的部分和可能会修改的部分加以注意并指明；应该讨论软件的可移植性问题，并且考虑可能影响软件维护的系统界面。

在正式的和非正式的设计复审期间，应该从容易修改、模块化和功能独立的目标出发，评价软件的结构和过程；设计中应该对将来可能修改的部分预作准备。

代码复审应该强调编码风格和内部说明文档这两个影响可维护性的因素。

在设计和编码过程中应该尽量使用可重用的软件构件，如果需要开发新的构件，也应该注意提高构件的可重用性。

每个测试步骤都可以暗示在软件正式交付使用前，程序中可能需要做预防性维护的部分。在测试结束时进行最正式的可维护性复审，这个复审称为配置复审。配置复审的目

的是保证软件配置的所有成分是完整的、一致的和可理解的，而且为了便于修改和管理已经编目归档了。

在完成了每项维护工作之后，都应该对软件维护本身进行仔细认真的复审。

维护应该针对整个软件配置，不应该只修改源程序代码。当对源程序代码的修改没有反映在设计文档或用户手册中时，就会产生严重的后果。

每当对数据、软件结构、模块过程或任何其他有关的软件特点做了改动时，必须立即修改相应的技术文档。不能准确反映软件当前状态的设计文档可能比完全没有文档更坏。在以后的维护工作中很可能因文档不完全符合实际而不能正确理解软件，从而在维护中引入过多的错误。

用户通常根据描述软件特点和使用方法的文档来使用、评价软件。如果对软件的可执行部分的修改没有及时反映在用户文档中，则必然会使用户因为受挫折而产生不满。

如果在软件再次交付使用之前，对软件配置进行严格的复审，则可大大减少文档的问题。事实上，某些维护要求可能并不需要修改软件设计或源程序代码，只是表明用户文档不清楚或不准确，因此只需要对文档做必要的维护。

## 8.5 预防性维护

几乎所有历史比较悠久的软件开发组织，都有一些十几年前开发出的“老”程序。目前，某些老程序仍然在为产品服务，但是，当初开发这些程序时并没有使用软件工程方法来指导，因此，这些程序的体系结构和数据结构都很差，文档不全甚至完全没有文档，对曾经做过的修改也没有完整的记录。

怎样满足用户对上述这类老程序的维护要求呢？为了修改这类程序以适应用户新的或变更的需求，有以下几种做法可供选择：

(1)反复多次地做修改程序的尝试，与不可见的设计及源代码“顽强战斗”，以实现所要求的修改；

(2)通过仔细分析程序尽可能多地掌握程序的内部工作细节，以便更有效地修改它；

(3)在深入理解原有设计的基础上，用软件工程方法重新设计、重新编码和测试那些需要变更的软件部分；

(4)以软件工程方法学为指导，对程序全部重新设计、重新编码和测试，为此可以使用CASE工具(逆向工程和再工程工具)来帮助理解原有的设计。

第一种做法很盲目，通常人们采用后3种做法。其中第4种做法称为软件再工程，这样的维护活动也就是本章8.1节中所说的预防性维护，而第3种做法实质上是局部的再工程。

预防性维护方法是由Miller提出来的，他把这种方法定义为：“把今天的方法学应用到昨天的系统上，以支持明天的需求。”粗看起来，在一个正在工作的程序版本已经存在的情况下重新开发一个大型程序，似乎是一种浪费。其实不然，下述事实很能说明问题：

(1)维护一行源代码的代价可能是最初开发该行源代码代价的14~40倍；

(2)重新设计软件体系结构(程序及数据结构)时使用了现代设计概念，它对将来的维护可能有很大的帮助；

(3)由于现有的程序版本可作为软件原型使用，开发生产率可大大高于平均水平；

(4)用户具有较多使用该软件的经验，因此，能够很容易地搞清新的变更需求和变更的范围；

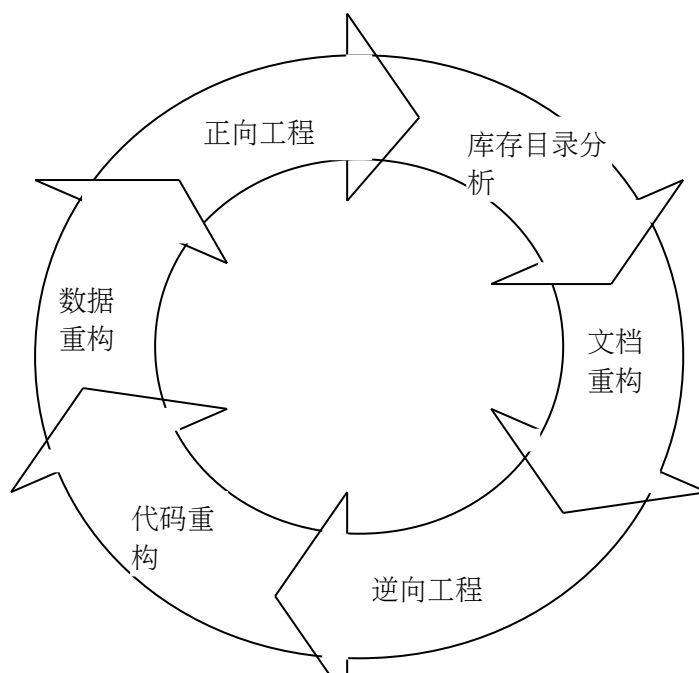
(5)利用逆向工程和再工程的工具，可以使一部分工作自动化；

(6)在完成预防性维护的过程中可以建立起完整的软件配置。

虽然由于条件所限，目前预防性维护在全部维护活动中仅占很小比例，但是，我们不应该忽视这类维护，在条件具备时应该主动地进行预防性维护。

## 8.6 软件再工程过程

典型的软件再工程过程模型如图 8.3 所示，该模型定义了 6 类活动。在某些情况下这些活动以线性顺序发生，但也并非总是这样，例如，为了理解某个程序的内部工作原理，可能在文档重构开始之前必须先进行逆向工程。



在图 8.3 中显示的再工程范型是一个循环模型。这意味着作为该范型的组成部分的每个活动都可能被重复，而日对于任意一个特定的循环来说，过程可以在完成任意一个活动之后终止。下面简要

图 8.3 软件再工程过程模型

### 1. 库存目录分析

每个软件组织都应该保存其拥有的所有应用系统的库存目录。该目录包含关于每个应用系统的基本信息(例如，应用系统的名字，最初构建它的日期，已做过的实质性修改次数，过去 18 个月报告的错误，用户数量，安装它的机器数量，它的复杂程度，文档质量，整体可维护性等级，预期寿命，在未来 36 个月内的预期修改次数，业务重要程度等)。

每一个大的软件开发机构都拥有上百万行老代码，它们都可能是逆向工程或再工程的对象。但是，某些程序并不频繁使用而且不需要改变，此外，逆向工程和再工程工具尚不成熟，目前仅能对有限种类的应用系统执行逆向工程或再工程，代价又十分高昂，因此对库中每个程序都做逆向工程或再工程是不现实的。下述 3 类程序有可能成为预防性维护的对象：

- (1) 预定将使用多年的程序；
- (2) 当前正在成功地使用着的程序；
- (3) 在最近的将来可能要做重大修改或增强的程序。

应该仔细分析库存目录，按照业务重要程度、寿命、当前可维护性、预期的修改次数等标准，把库中的应用系统排序，从中选出再工程的候选者，然后明智地分配再工程所需要的资源。

## 2. 文档重构

老程序固有的特点是缺乏文档。具体情况不同，处理这个问题的方法也不同：

(1)建立文档非常耗费时间，不可能为数百个程序都重新建立文档。如果一个程序是相对稳定的，正在走向其有用生命的终点，而且可能不会再经历什么变化，那么，让它保持现状是一个明智的选择。

(2)为了便于今后的维护，必须更新文档，但是由于资源有限，应采用“使用时建文档”的方法，也就是说，不是一下子把某应用系统的文档全部都重建起来，而是只针对系统中当前正在修改的那些部分建立完整文档。随着时间流逝，将得到一组有用的和相关的文档。

(3)如果某应用系统是完成业务工作的关键，而且必须重构全部文档，则仍然应该设法把文档工作减少到必需的最小量。

## 3. 逆向工程

软件的逆向工程是分析程序以便在比源代码更高的抽象层次上创建出程序的某种表示的过程，也就是说，逆向工程是一个恢复设计结果的过程，逆向工程工具从现存的程序代码中抽取有关数据、体系结构和处理过程的设计信息。

## 4. 代码重构

代码重构是最常见的再工程活动。某些老程序具有比较完整、合理的体系结构，但是，个体模块的编码方式却是难于理解、测试和维护的。在这种情况下，可以重构可疑模块的代码。

为了完成代码重构活动，首先用重构工具分析源代码，标注出和结构化程序设计概念相违背的部分。然后重构有问题的代码(此项工作可自动进行)。最后，复审和测试生成的重构代码(以保证没有引入异常)并更新代码文档。

通常，重构并不修改整体的程序体系结构，它仅关注个体模块的设计细节以及在模块中定义的局部数据结构。如果重构扩展到模块边界之外并涉及软件体系结构，则重构变成了正向工程。

## 5. 数据重构

对数据体系结构差的程序很难进行适应性修改和增强，事实上，对许多应用系统来说，数据体系结构比源代码本身对程序的长期生存力有更大影响。

与代码重构不同，数据重构发生在相当低的抽象层次上，它是一种全范围的再工程

活动。在大多数情况下，数据重构始于逆向工程活动，分解当前使用的数据体系结构，必要时定义数据模型，标识数据对象和属性，并从软件质量的角度复审现存的数据结构。

当数据结构较差时(例如，在关系型方法可大大简化处理的情况下却使用平坦文件实现)，应该对数据进行再工程。

由于数据体系结构对程序体系结构及程序中的算法有很大影响，对数据的修改必然会导致体系结构或代码层的改变。

## 6. 正向工程

正向工程也称为革新或改造，这项活动不仅从现有程序中恢复设计信息，而且使用该信息去改变或重构现有系统，以提高其整体质量。

正向工程过程应用软件工程的原理、概念、技术和方法来重新开发某个现有的应用系统。在大多数情况下，被再工程的软件不仅重新实现现有系统的功能，而且加入了新功能和提高了整体性能。

## 8.7 小结

维护是软件生命周期的最后一个阶段，也是持续时间最长代价最大的一个阶段。软件工程学的主要目的就是提高软件的可维护性，降低维护的代价。

软件维护通常包括 4 类活动：为了纠正在使用过程中暴露出来的错误而进行的改正性维护；为了适应外部环境的变化而进行的适应性维护；为了改进原有的软件而进行的完善性维护；以及为了改进将来的可维护性和可靠性而进行的预防性维护。

软件的可理解性、可测试性、可修改性、可移植性和可重用性，是决定软件可维护性的基本因素，软件重用技术是能从根本上提高软件可维护性的重要技术，而本书第 9 章至第 12 章将要讲述的面向对象的软件技术是目前最成功的软件重用技术。

软件生命周期每个阶段的工作都和软件可维护性有密切关系。良好的设计，完整准确易读易理解的文档资料，以及一系列严格的复审和测试，使得一旦发现错误时比较容易诊断和纠正，当用户有新要求或外部环境变化时软件能较容易地适应，并且能够减少维护引入的错误。因此，在软件生命周期的每个阶段都必须充分考虑维护问题，并且为软件维护预做准备。

文档是影响软件可维护性的决定因素，因此，文档甚至比可执行的程序代码更重要。文档可分为用户文档和系统文档两大类。不管是哪一类文档都必须和程序代码同时维护，只有和程序代码完全一致的文档才是真正有价值的文档。

虽然由于维护资源有限，目前预防性维护在全部维护活动中仅占很小比例，但是不应该忽视这类维护活动，在条件具备时应该主动地进行预防性维护。预防性维护实质上是软件再工程。典型的软件再工程过程模型定义了库存目录分析、文档重构、逆向工程、代码重构、数据重构和正向工程等 6 类活动。在某些情况下，以线性顺序完成这些活动，但也并不总是这样。上述模型是一个循环模型，这意味着每项活动都可能被重复，而且对于任意一个特定的循环来说，再工程过程可以在完成任意一个活动之后终止。