软件维护

- 软件维护的概念
- 软件维护活动
- 程序修改的步骤及修改的副作用
- 可维护性
- 提高可维护性的方法

软件维护的概念

- 软件维护的定义
- 影响维护工作量的因素
- 软件维护的策略
- 维护成本

软件维护的定义

- 在软件运行／维护阶段对软件产品进行的修改就是所谓的维护。
- 维护的类型有三种：
  - 改正性维护
  - 适应性维护
  - 完善性维护

改正性维护

- 在软件交付使用后，因开发时测试的不彻底、不完全，必然会有部分隐藏的错误遗留到运行阶段。
- 这些隐藏下来的错误在某些特定的使用环境下就会暴露出来。
- 为了识别和纠正软件错误、改正软件性能上的缺陷、排除实施中的误使用，应当进行的诊断和改正错误的过程就叫做改正性维护。

适应性维护

- 在使用过程中，
  - 外部环境（新的硬、软件配置）
  - 数据环境（数据库、数据格式、数据输入/输出方式、数据存储介质）

  可能发生变化。
- 为使软件适应这种变化，而去修改软件的过程就叫做适应性维护。

完善性维护

- 在软件的使用过程中，用户往往会对软件提出新的功能与性能要求。
- 为了满足这些要求，需要修改或再开发软件，以扩充软件功能、增强软件性能、改进加工效率、提高软件的可维护性。
- 这种情况下进行的维护活动叫做完善性维护。

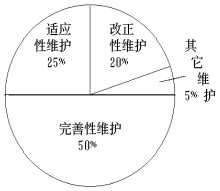
- 实践表明，在几种维护活动中，完善性维护所占的比重最大。即大部分维护工作是改变和加强软件，而不是纠错。
- 完善性维护不一定是救火式的紧急维修，而可以是有计划、有预谋的一种再开发活动。
- 事实证明，来自用户要求扩充、加强软件功能、性能的维护活动约占整个维护工作的50%。

预防性维护

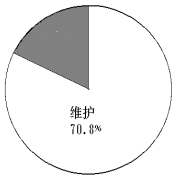
- 预防性维护是为了提高软件的可维护性、可靠性等，为以后进一步改进软件打下良好基础。
- 预防性维护定义为：采用先进的软件工程方法对需要维护的软件或软件中的某一部分（重新）进行设计、编制和测试。

- 在整个软件维护阶段所花费的全部工作量中，完善性维护占了几乎一半的工作量。
- 软件维护活动所花费的工作占整个生存期工作量的70%以上，这是由于在漫长的软件运行过程中需要不断对软件进行修改，以改正新发现的错误、适应新的环境和用户新的要求，这些修改需要花费很多精力和时间，而且有时会引入新的错误。

三类维护占总维护比例



维护在软件生存期所占比例



影响维护工作量的因素

- 在软件的维护过程中，需要花费大量的工作量，从而直接影响了软件维护的成本。
- 应当考虑有哪些因素影响软件维护的工作量，相应应该采取什么维护策略，才能有效地维护软件并控制维护的成本。

- 系统大小：系统越大，理解掌握起来越困难。系统越大，所执行功能越复杂。因而需要更多的维护工作量。
- 程序设计语言：使用强功能的程序设计语言可以控制程序的规模。语言的功能越强，生成程序的模块化和结构化程度越高，所需的指令数就越少，程序的可读性越好。

- 系统年龄：
  - 老系统随着不断的修改，结构越来越乱；
  - 维护人员经常更换，程序又变得越来越难于理解。
  - 许多老系统在当初并未按照软件工程的要求进行开发，因而没有文档，或文档太少。
  - 在长期的维护过程中文档在许多地方与程序实现变得不一致，在维护时就会遇到很大困难。

- 数据库技术的应用：使用数据库，可以简单而有效地管理和存储用户程序中的数据，还可以减少生成用户报表应用软件的维护工作量。
- 先进的软件开发技术：在软件开发时，若使用能使软件结构比较稳定的分析与设计技术，及程序设计技术，如面向对象技术、复用技术等，可减少大量的工作量。

- 其它：
  - 应用的类型
  - 数学模型
  - 任务的难度
  - 开关与标记、IF嵌套深度、索引或下标数等

  对维护工作量都有影响。
- 许多软件在开发时并未考虑将来的修改，为软件的维护带来许多问题。

软件维护的策略

- 改正性维护
  通常要生成100%可靠的软件并不一定合算，成本太高。但通过使用新技术，可大大减少进行改正性维护的需要。

  这些技术包括：数据库管理系统、软件开发环境、程序自动生成系统、较高级(第四代)的语言。以及新的开发方法、软件复用、防错程序设计及周期性维护审查等。

- 适应性维护
  这一类维护不可避免，可以控制。
  (1) 在配置管理时，把硬件、操作系统和其它相关环境因素的可能变化考虑在内。
  (2) 把与硬件、操作系统，以及其它外围设备有关的程序归到特定的程序模块中。
  (3) 使用内部程序列表、外部文件，以及处理的例行程序包，可为维护时修改程序提供方便。

- 完善性维护
  利用前两类维护中列举的方法，也可以减少这一类维护。特别是数据库管理系统、程序生成器、应用软件包，可减少维护工作量。此外，建立软件系统的原型，把它在实际系统开发之前提供给用户。用户通过研究原型，进一步完善他们的功能要求，就可以减少以后完善性维护的需要。

维护成本

- 有形的软件维护成本是花费了多少钱，无形的维护成本有更大的影响。
  - 一些合理的修复或修改请求不能及时安排，使得客户不满意；
  - 变更的结果引入新的故障，使得软件整体质量下降；
  - 把软件人员抽调到维护工作中，干扰了软件开发工作。

- 软件维护的代价是降低了生产率，在做老程序的维护时非常明显。
- 例如，开发每一行源代码耗资25美元，维护每一行源代码需要耗资1000美元。
- 维护工作量包括生产性活动（如分析和评价、设计修改和实现）和“轮转”活动（如力图理解代码在做什么、试图判明数据结构、接口特性、性能界限等）。

维护工作量的模型

$$M = p + Ke^{c-d}$$

- M是维护中消耗的总工作量
- p是上面描述的生产性工作量
- K是一个经验常数
- c是因缺乏好的设计和文档而导致复杂性的度量
- d是对软件熟悉程度的度量。

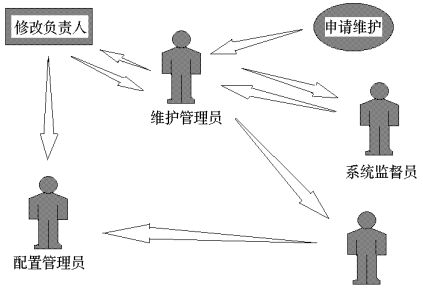
- 模型指明，如果使用了不好的软件开发方法（未按软件工程要求做），原来参加开发的人员或小组不能参加维护，则工作量（及成本）将按指数级增加。

软件维护活动

- 为了有效地进行软件维护，应事先就开始做组织工作。
  - 首先建立维护的机构
  - 申明提出维护申请报告的过程及评价的过程
  - 为每一个维护申请规定标准的处理步骤
  - 建立维护活动的登记制度以及规定评价和评审的标准。

维护机构

- 除了较大的软件开发公司外，通常在软件维护工作方面，并不保持一个正式的组织机构。
- 虽然不要求建立一个正式的维护机构，但是在开发部门确立一个非正式的维护机构则是非常必要的。



软件维护的机构

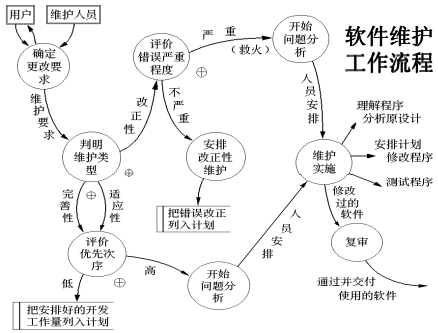
- 维护申请提交给维护管理员，他把申请交给某个系统监督员去评价。
- 一旦做出评价，由修改负责人确定如何进行修改。
- 在修改程序的过程中，由配置管理员严格把关，控制修改的范围，对软件配置进行审计。
- 在维护之前，就把责任明确下来，可以减少维护过程中的混乱。

软件维护申请报告

- 维护申请报告或称软件问题报告，由申请维护的用户填写。
- 用户必须完整地说明产生错误的情况，包括输入数据、错误清单以及其它有关材料。
- 如果申请的是适应性维护或完善性维护，用户必须提出一份修改说明书，列出所有希望的修改。

- 维护申请报告将由维护管理员和系统监督员来研究处理。
- 他们应相应地做出软件修改报告，指明：
  - 所需修改变动的性质；
  - 申请修改的优先级；
  - 为满足某个维护申请报告，所需的工作量；
  - 预计修改后的状况.

- 软件修改报告应提交修改负责人，经批准后才能开始进一步安排维护工作。



软件维护工作流程

- 尽管维护申请的类型不同，但都要进行同样的技术工作。
  - 修改软件需求说明
  - 修改软件设计
  - 设计评审
  - 对源程序做必要的修改
  - 单元测试
  - 集成测试（回归测试）
  - 确认测试
  - 软件配置评审等。

在每次软件维护任务完成后进行情况评审，对以下问题做一总结：
(1) 在目前情况下，设计、编码、测试中的哪一方面可以改进?
(2) 哪些维护资源应该有但没有?
(3) 工作中主要的或次要的障碍是什么?
(4) 从维护申请的类型来看是否应当有预防性维护?
情况评审对将来的维护工作如何进行会产生重要的影响。

维护档案记录

- 程序名称
- 源程序语句条数
- 机器代码指令条数
- 所用的程序设计语言
- 程序安装的日期
- 程序安装后的运行次数
- 与程序安装后运行次数有关的处理故障次数

- 程序改变的层次及名称
- 修改程序增加的源程序语句条数
- 修改程序减少的源程序语句条数
- 每次修改所付出的“人时”数
- 修改程序的日期
- 软件维护人员的姓名
- 维护申请报告的名称、维护类型
- 维护开始时间和维护结束时间、
- 花费在维护上的累计“人时”数
- 维护工作的净收益等。

维护评价

- 评价维护活动比较困难，因为缺乏可靠的数据。
- 如果维护的档案记录做得比较好，可以得出一些维护“性能”方面的度量值。
  - 每次程序运行时的平均出错次数；
  - 花费在每类维护上的总“人时”数；

  - 每个程序、每种语言、每种维护类型的程序平均修改次数；
  - 因为维护，增加或删除每个源程序语句所花费的平均“人时”数；
  - 用于每种语言的平均“人时”数；
  - 维护申请报告的平均处理时间；
  - 各类维护申请的百分比。

据此可对开发技术、语言选择、维护工作计划、资源分配、以及其它许多方面做出判定。

程序修改的步骤及修改的副作用

- 在软件维护时，必然会对源程序进行修改。
- 通常对源程序的修改不能无计划地仓促上阵，为了正确、有效地修改，需要经历以下三个步骤。
  - ☞ 分析和理解程序
  - ☞ 修改程序
  - ☞ 重新验证程序

分析和理解程序

- 经过分析，全面、准确、迅速地理解程序是决定维护成败和质量好坏的关键。在这方面，软件的可理解性和文档的质量非常重要。
  - 理解程序的功能和目标；
  - 掌握程序的结构信息，即从程序中细分出若干结构成分。如程序系统结构、 控制结构、数据结构和输入／输出结构等；

  - 了解数据流信息，即涉及到的数据来源何处，在哪里被使用；
  - 了解控制流信息，即执行每条路径的结果；
  - 理解程序的操作(使用)要求；
- 为了容易地理解程序，要求自顶向下地理解现有源程序的程序结构和数据结构，为此可采用如下几种方法：

1. 分析程序结构图

(1) 搜集所有存储该程序的文件，阅读这些文件，记下它们包含的过程名，建立一个包括这些过程名和文件名的清单；

(2) 分析各个过程的源代码，建立一个直接调用矩阵D或调用树。若过程 i 调用过程 j，则$D[i][j]=1$，否则$D[i][j]=0$。

(3) 建立过程的间接调用矩阵I，即直接调用矩阵D的传递闭包

$$I=D1\cup D2\cup D3\cup\ldots\cup Dn$$

其中，n是所包含的过程总数.

例如，过程 i 调用 j，j 调用 k，

则 $D[i][j]=1$，$D[j][k]=1$，

$I[i][k]=1$。

(4) 分析各个过程的接口，估计更改的复杂性。

2. 数据跟踪

(1) 建立各层次的程序级上的接口图，展示各模块或过程的调用方式和接口参数；

(2) 利用数据流分析方法，对过程内部的一些变量进行跟踪。可获得有关数据在过程间如何传递，在过程内如何处理等信息。对于判断问题原因特别有用。在跟踪的过程中可在源程序中间插入自己的注释。

3. 控制跟踪

控制流跟踪可采用符号执行或实际动态跟踪的方法，了解数据如何从一个输入源到达输出点的。

4. 充分阅读和使用源程序清单和文档，分析现有文档的合理性。

5. 充分使用由编译程序或汇编程序提供的交叉引用表、符号表、以及其它有用的信息。

6. 如有可能，积极参加开发工作。

修改程序

- 对程序的修改，必须事先做出计划，有预谋地、周密有效地实施修改。

1. 设计程序的修改计划

程序的修改计划要考虑人员和资源的安排。小的修改可以不需要详细的计划，而对于需要耗时数月的修改，就需要计划立案。

在编写有关问题解决的方案时，必须充分描述修改作业的规格说明。

☞ 规格说明信息：数据修改、处理修改、作业控制语言修改、系统之间接口的修改等；

☞ 维护资源：新程序版本、测试数据、所需软件、计算机时间等；

☞ 人员；

☞ 支持：纸面、计算机媒体等。

通常，可采用自顶向下的方法，在理解程序的基础上，

(1) 研究程序的各个模块、模块的接口、及数据库，从全局的观点，提出修改计划。

(2) 依次地把要修改的、以及那些受修改影响的模块和数据结构分离出来。为此，要

☞ 识别受修改影响的数据；

☞ 识别使用这些数据的程序模块；

☞ 对于上面程序模块，按是产生数据、修改数据、还是删除数据进行分类；

☞ 识别对这些数据元素的外部控制信息；

☞ 识别编辑和检查这些数据元素的地方；

☞ 隔离要修改的部分；

(3) 详细地分析要修改的、以及那些受变更影响的模块和数据结构的内部细节，设计修改计划，标明新逻辑及要改动的现有逻辑。

(4) 向用户提供回避措施。用户的某些业务因软件中发生问题而中断，为不让系统长时间停止运行，需把问题局部化，在可能的范围内继续开展业务。

可以采取的措施有：

① 查找问题原因，可能情况有：
- ☞ 意外停机
- ☞ 安装的期限到期
- ☞ 系统运行中发现错误

② 如果弄清了问题的原因，可通过临时修改或改变运行控制以回避在系统运行时产生的问题。

2. 修改代码，以适应变化
在修改时，要求：
(1) 正确、有效地编写修改代码；
(2) 要谨慎地修改程序，尽量保持程序的风格及格式，要在程序清单上注明改动的指令；
(3) 不要删除程序语句，除非完全肯定它是无用的；
(4) 不要试图共用程序中已有的临时变量或工作区，为了避免冲突或混淆用途，应设置自己的变量；

(5) 插入错误检测语句；
(6) 在修改过程中做好修改的详细记录，消除变更中任何有害的副作用（波动效应）；

3. 修改程序的副作用
所谓副作用是指因修改软件而造成的错误或其它不希望发生的情况。副作用有三种：修改代码的副作用、修改数据的副作用、文档的副作用。

(1) 修改代码的副作用

- 在修改源代码时，都可能引入错误。例如，删除或修改一个子程序、删除或修改一个标号、 删除或修改一个标识符、改变程序代码的时序关系、改变占用存储的大小、改变逻辑运算符、修改文件的打开或关闭、改进程序的执行效率，以及把设计上的改变翻译成代码的改变时，都容易引入错误。

(2) 修改数据的副作用

- 在修改数据结构时，有可能造成软件设计与数据结构不匹配，因而导致软件出错。
- 数据副作用就是修改软件信息结构导致的结果。
- 容易导致设计与数据不相容的错误可以有：
  - ◆ 重新定义局部的或全局的常量

  - ◆ 重新定义记录或文件的格式
  - ◆ 增大或减小一个数组或高层数据结构的大小
  - ◆ 修改全局或公共数据
  - ◆ 重新初始化控制标志或指针
  - ◆ 重新排列输入 / 输出或子程序的参数
- 数据副作用可以通过交叉引用表加以控制。把数据元素、记录、文件和其它结构联系起来。

(3) 文档的副作用

- 对数据流、软件结构、 模块逻辑或任何其它有关特性进行修改时，必须对相关技术文档进行相应修改。否则会导致文档与程序功能不匹配，缺省条件改变，新错误信息不正确等错误。使得软件文档不能反映软件的当前状态。
- 对于用户来说，软件事实上就是文档。

- 如果对可执行软件的修改不反映在文档里，就会产生文档的副作用。
  - ◆ 对交互输入的顺序或格式进行修改，如果没有正确地记入文档中，就可能引起重大的问题。
  - ◆ 过时的文档内容、索引和文本可能造成冲突，引起用户失败和不满。
- 因此，必须在软件交付之前对整个软件配置进行评审，以减少文档的副作用。

- 为了控制因修改而引起的副作用，要做到：
(1) 按模块把修改分组；
(2) 自顶向下地安排被修改模块的顺序；
(3) 每次修改一个模块；
(4) 对于每个修改了的模块，在安排修改下一个模块之前，要确定这个修改的副作用。可以使用交叉引用表、存储映象表、执行流程跟踪等。

重新验证程序

- 在将修改后的程序提交用户之前，需要进行充分的确认和测试，以保证整个修改后程序的正确性。
- 静态确认
修改软件，伴随着引起新的错误的危险。为了能够做出正确的判断，验证修改后的程序至少需要两个人参加。要检查：

(1) 修改是否涉及到规格说明？修改结果是否符合规格说明？有没有歪曲规格说明?
(2) 程序的修改是否足以修正软件中的问题？源程序代码有无逻辑错误？修改时有无修补失误?
(3) 修改部分对其它部分有无不良影响(副作用)?
对软件进行修改，常常会引发别的问题，有必要检查修改的影响范围。

- 计算机确认
在进行了以上确认的基础上，用计算机对修改程序进行确认测试：
(1) 确认测试顺序：先对修改部分进行测试，然后隔离修改部分，测试程序的未修改部分，最后再把它们集成起来进行测试。这种测试称为回归测试。
(2) 准备标准的测试用例。
(3) 充分利用软件工具帮助重新验证过程。

(4) 在重新确认过程中，需邀请用户参加。
- 维护后的验收——在交付新软件之前，维护主管部门要检验：
(1) 全部文档是否完备，并已更新；
(2) 所有测试用例和测试结果已经正确记载；
(3) 记录软件配置所有副本的工作已经完成；
(4) 维护工序和责任已经确定。

从维护角度来看所需测试种类

(1) 对修改事务的测试；
(2) 对修改程序的测试；
(3) 操作过程的测试；
(4) 应用系统运用过程的测试；
(5) 系统各部分之间接口的测试；
(6) 作业控制语言的测试；
(7) 与系统软件接口的测试；

(8) 软件系统之间接口的测试；
(9) 安全性测试；
(10) 后备 / 恢复过程的测试。

软件可维护性

- 许多软件的维护十分困难，原因在于这些软件的文档不全、质量差、开发过程不注意采用好的方法，忽视程序设计风格等。
- 许多维护要求并不是因为程序中出错而提出的，而是为适应环境变化或需求变化而提出的。
- 为了使得软件能够易于维护，必须考虑使软件具有可维护性。

软件可维护性的定义

- 软件可维护性是指纠正软件系统出现的错误和缺陷，以及为满足新的要求进行修改、扩充或压缩的容易程度。
- 可维护性、可使用性、可靠性是衡量软件质量的主要质量特性，也是用户十分关心的几个方面。
- 软件的可维护性是软件开发阶段各个时期的关键目标。

- 目前广泛使用的是用如下的七个特性来衡量程序的可维护性。

  可理解性　　可使用性

  可测试性　　可移植性

  可修改性　　效率

  可靠性

- 而且对于不同类型的维护，这七种特性的侧重点也不相同。

在各类维护中的侧重点

	改正性维护	适应性维护	完善性维护
可理解性	o		
可测试性	o		
可修改性	o	o	
可 靠 性	o		
可移植性		o	
可使用性		o	o
效 率			o

- 这些质量特性通常体现在软件产品的许多方面;
- 为使每一个质量特性都达到预定的要求，需要在软件开发的各个阶段采取相应的措施加以保证。
- 这些质量要求要渗透到而各开发阶段的各个步骤当中。因此，软件的可维护性是产品投入运行以前各阶段面向上述各质量特性要求进行开发的最终结果。

可维护性的度量

- 人们一直期望对软件的可维护性做出定量度量，但要做到这一点并不容易。
- 常用的度量一个可维护的程序的七种特性的方法。就是
  - 质量检查表
  - 质量测试
  - 质量标准

- 质量检查表是用于测试程序中某些质量特性是否存在的一个问题清单。
- 评价者针对检查表上的每一个问题，依据自己的定性判断，回答“Yes”或者“No”。
- 质量测试与质量标准则用于定量分析和评价程序的质量。
- 由于许多质量特性是相互抵触的，要考虑几种不同的度量标准，相应地去度量不同的质量特性。

1. 可理解性

- 可理解性表明人们通过阅读源代码和相关文档，了解程序功能及其如何运行的容易程度。
- 一个可理解的程序应具备以下一些特性：模块化，风格一致性，不使用令人捉摸不定或含糊不清的代码，使用有意义的数据名和过程名，结构化，完整性等。

2. 可靠性

- 可靠性表明一个程序按照用户的要求和设计目标，在给定的一段时间内正确执行的概率。
- 关于可靠性，度量的标准主要有：
  - 平均失效间隔时间MTTF
  - 平均修复时间MTTR
  - 有效性A = MTBD/(MTBD+MDT)

度量可靠性的方法

- 根据程序错误统计数字，进行可靠性预测。常用方法是利用一些可靠性模型，根据程序测试时发现并排除的错误数预测平均失效间隔时间MTTF。
- 根据程序复杂性，预测软件可靠性。用程序复杂性预测可靠性，前提条件是可靠性与复杂性有关。因此可用复杂性预测出错率。程序复杂性度量标准可用于预测哪些模块最可能发生错误，以及可能出现的错误类型。

3. 可测试性

- 可测试性表明论证程序正确性的容易程度。程序越简单，证明其正确性就越容易。而且设计合用的测试用例，取决于对程序的全面理解。
- 一个可测试的程序应当是可理解的，可靠的，简单的。
- 用于可测试性度量的检查项目如下：
  - 程序是否模块化？结构是否良好?

- 程序是否可理解？程序是否可靠?
- 程序是否能显示任意中间结果?
- 程序是否能以清楚的方式描述它的输出?
- 程序是否能及时地按照要求显示所有的输入?
- 程序是否有跟踪及显示逻辑控制流程的能力?
- 程序是否能从检查点再启动?
- 程序是否能显示带说明的错误信息?

4. 可修改性

- 可修改性表明程序容易修改的程度。
- 一个可修改的程序应当是可理解的、通用的、灵活的、简单的。
- 通用性是指程序适用于各种功能变化而无需修改。
- 灵活性是指能够容易地对程序进行修改。

- 测试可修改性的一种定量方法是修改练习。其基本思想是通过做几个简单的修改，来评价修改的难度。
- 设*C*是程序中各个模块的平均复杂性，*n*是必须修改的模块数，*A* 是要修改的模块的平均复杂性。 则修改的难度*D*由下式计算：

$$D = A / C$$

5. 可移植性

- 可移植性表明程序转移到一个新的计算环境的可能性的大小。或者它表明程序可以容易地、有效地在各种各样的计算环境中运行的容易程度。
- 一个可移植的程序应具有结构良好、灵活、不依赖于某一具体计算机或操作系统的性能。
- 用于可移植性度量的检查项目如下：

- 是否是用高级的独立于机器的语言来编写程序?
- 是否使用广泛使用的标准化的程序设计语言来编写程序？是否仅使用了这种语言的标准版本和特性?
- 程序中是否使用了标准的普遍使用的库功能和子程序?
- 程序中是否极少使用或根本不使用操作系统的功能?

- 程序在执行之前是否初始化内存?
- 程序在执行之前是否测定当前的输入／输出设备?
- 程序是否把与机器相关的语句分离了出来，集中放在了一些单独的程序模块中，并有说明文件?
- 程序是否结构化? 并允许在小一些的计算机上分段(覆盖)运行?
- 程序中是否避免了依赖于字母数字或特殊字符的内部位表示?

6. 效率

- 效率表明一个程序能执行预定功能而又不浪费机器资源的程度。
- 这些机器资源包括内存容量、外存容量、通道容量和执行时间。
- 用于效率度量的检查项目如下：
  - 程序是否模块化？结构是否良好？
  - 是否消除了无用的标号与表达式，以充分发挥编译器优化作用？

- 程序的编译器是否有优化功能？
- 是否把特殊子程序和错误处理子程序都归入了单独的模块中？
- 是否以快速的数学运算代替了较慢的数学运算？
- 是否尽可能地使用了整数运算，而不是实数运算？
- 是否在表达式中避免了混合数据类型的使用，消除了不必要的类型转换？

- 程序是否避免了非标准的函数或子程序的调用？
- 在几条分支结构中，是否最有可能为“真”的分支首先得到测试？
- 在复杂的逻辑条件中，是否最有可能为“真“的表达式首先得到测试？

7. 可使用性

- 从用户观点出发，可使用性定义为程序方便、实用、及易于使用的程度。一个可使用的程序应是易于使用的、能允许用户出错和改变，并尽可能不使用户陷入混乱状态的程序。
- 用于可使用性度量的检查项目如下：
  - 程序是否具有自描述性？

- 程序是否能始终如一地按照用户的要求运行？
- 程序是否让用户对数据处理有一个满意的和适当的控制？
- 程序是否容易学会使用？
- 程序是否使用数据管理系统来自动地处理事务性工作和管理格式化、地址分配及存储器组织。
- 程序是否具有容错性？
- 程序是否灵活？

其它间接定量度量可维护性的方法

- 问题识别的时间；
- 因管理活动拖延的时间；
- 收集维护工具的时间；
- 分析、诊断问题的时间；
- 修改规格说明的时间；
- 具体的改错或修改的时间；
- 局部测试的时间；
- 集成或回归测试的时间；
- 维护的评审时间；

- 这些数据反映了维护全过程中检错－纠错－验证的周期，即从检测出软件存在的问题开始至修正它们并经回归测试验证这段时间。
- 可以粗略地认为，这个周期越短，维护越容易。

提高可维护性的方法

- 建立明确的软件质量目标和优先级
- 使用提高软件质量的技术和工具
- 进行明确的质量保证审查
- 选择可维护的程序设计语言
- 改进程序的文档

建立明确的软件质量目标和优先级

- 一个可维护的程序应是可理解的、可靠的、可测试的、可修改的、可移植的、效率高的、可使用的。
- 要实现这所有的目标，需要付出很大的代价，而且也不一定行得通。
- 某些质量特性是相互促进的，例如可理解性和可测试性、可理解性和可修改性。

- 另一些质量特性是相互抵触的，如效率和可移植性、效率和可修改性等。
- 每一种质量特性的相对重要性应随程序的用途及计算环境的不同而不同。例如，对编译程序来说，可能强调效率；但对管理信息系统来说，则可能强调可使用性和可修改性。
- 应当对程序的质量特性，在提出目标的同时还必须规定它们的优先级。

使用提高软件质量的技术和工具

- 模块化
  - 如果需要改变某个模块的功能，则只要改变这个模块，对其它模块影响很小；
  - 如果需要增加程序的某些功能，则仅需增加完成这些功能的新的模块或模块层；
  - 程序的测试与重复测试比较容易；
  - 程序错误易于定位和纠正；

- 结构化程序设计
  - 程序被划分成分层的模块结构；
  - 模块调用控制必须从模块的入口点进入，从其出口点退出。
  - 模块的控制结构仅限于顺序、选择、重复三种，且没有GOTO语句。
  - 每个程序变量只用于唯一的程序目的，而且变量的作用范围应是明确的、有限制的。

- 使用结构化程序设计技术，提高现有系统的可维护性
  - 采用备用件的方法——用一个新的结构良好的模块替换掉整个要修改的模块。
  - 采用自动重建结构和重新格式化的工具(结构更新技术)——把非结构化代码转换成良好结构代码 。
  - 改进现有程序的不完善的文档——建立或补充系统说明书、设计文档、模块说明书、以及在源程序中插入必要的注释。

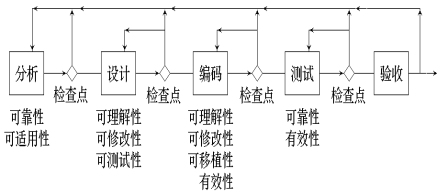
进行明确的质量保证审查

- 质量保证审查对于获得和维持软件的质量，是一个很有用的技术。
- 审查可以用来检测在开发和维护阶段内发生的质量变化。
- 一旦检测出问题来，就可以采取措施来纠正，以控制不断增长的软件维护成本，延长软件系统的有效生命期。

1. 在检查点进行复审

- 保证软件质量的最佳方法是在软件开发的最初阶段把质量要求考虑进去，并在开发过程每一阶段的终点，设置检查点进行检查。
- 检查的目的是要证实，已开发的软件是否符合标准，是否满足规定的质量需求。在不同的检查点，检查的重点不完全相同。

软件开发期间各个检查点的检查重点



- 在设计阶段，检查重点是可理解性、可修改性、可测试性。
- 可理解性检查的重点是程序的复杂性。对每个模块可用McCabe环路来计算模块的复杂性，若大于10，则需重新设计。
- 可以使用各种质量特性检查表，或用度量标准来检查可维护性。
- 审查小组可以采用人工测试一类的方式，进行审查。

2. 验收检查

- 验收检查是一个特殊的检查点的检查，是交付使用前的最后一次检查，
- 验收检查实际上是验收测试的一部分，只不过它是从维护的角度提出验收的条件和标准。
- 验收检查必须遵循的最小验收标准。

(1) 需求和规范标准

① 需求应当以可测试的术语进行书写，排列优先次序和定义；

② 区分必须的、任选的、将来的需求；

③ 包括对系统运行时的计算机设备的需求；对维护、测试、操作、以及维护人员的需求；对测试工具等的需求。

(2) 设计标准

① 程序应设计成分层的模块结构。每个模块应完成唯一的功能，并达到高内聚、低耦合；

② 通过一些知道预期变化的实例，说明设计的可扩充性、可缩减性和可适应性。

(3) 源代码标准

① 尽可能使用最高级的程序设计语言，且只使用语言的标准版本；

② 所有的代码都必须具有良好的结构；

③ 所有的代码都必须文档化，在注释中说明它的输入、输出、以及便于测试／再测试的一些特点与风格。

(4) 文档标准

文档中应说明

- 程序的输入／输出
- 使用的方法／算法
- 错误恢复方法
- 所有参数的范围
- 缺省条件等。

3. 周期性地维护审查

- 检查点复查和验收检查，可用来保证新软件系统的可维护性。
- 对已有的软件系统，则应当进行周期性的维护检查。
- 软件在运行期间进行修改，会导致软件质量有变坏的危险，破坏程序概念的完整性。
- 必须定期检查，对软件做周期性的维护审查，以跟踪软件质量的变化。

- 周期性维护审查实际上是开发阶段检查点复查的继续，并且采用的检查方法、检查内容都是相同的。
- 维护审查的结果可以同以前的维护审查的结果，以前的验收检查的结果、检查点检查的结果相比较，任何一种改变都表明在软件质量上或其它类型的问题上可能起了变化。
- 对于改变的原因应当进行分析。

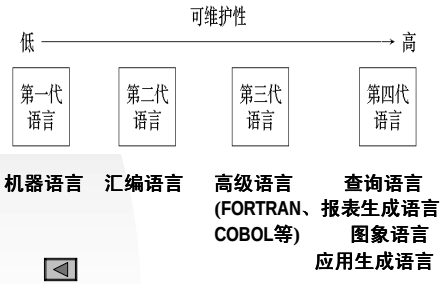
4. 对软件包进行检查

- 软件包是一种标准化的，可为不同单位、不同用户使用的软件。
- 一般源代码和程序文档不会提供给用户。
- 对软件包的维护采取以下方法。
  - 使用单位的维护人员首先要仔细分析、研究卖主提供的用户手册、操作手册、培训教程等，以及卖方提供的验收测试报告等。

- 在此基础上，深入了解本单位的希望和要求，编制软件包的检验程序。检查软件包程序所执行的功能是否与用户的要求和条件相一致。
- 为了建立这个程序，维护人员可以利用卖方提供的验收测试实例，还可以自己重新设计新的测试实例。
- 根据测试结果，检查和验证软件包的参数或控制结构，以完成软件包的维护。

选择可维护的程序设计语言

- 程序设计语言的选择，对程序的可维护性影响很大。



改进程序的文档

- 程序文档是对程序总目标、程序各组成部分之间的关系、程序设计策略、程序实现过程的历史数据等的说明和补充。
- 即使是一个十分简单的程序，要想有效地、高效率地维护它，也需要编制文档来解释其目的及任务。

- 对于程序维护人员来说，要想按程序编制人员的意图重新改造程序，并对今后变化的可能性进行估计，缺了文档是不行的。
- 因此，为了维护程序，人们必须阅读和理解文档。
- 另外，在软件维护阶段，利用历史文档，可以大大简化维护工作。通过了解原设计思想，可以判断出错之处，指导维护人员选择适当的方法修改代码而不危及系统的完整性。

- 历史文档有三种：
  - 系统开发日志
  - 错误记载
  - 系统维护日志