软件质量管理

- 软件质量概念
- 软件质量保证
- 软件可靠性
- 软件配置管理

软件质量概念

- 软件质量的定义
- 软件质量特性
- 软件质量模型
- 软件质量的度量和评价

软件质量的定义

- ANSI/IEEE Std 729-1983定义软件质量为“与软件产品满足规定的和隐含的需求的能力有关的特征或特性的全体”。
- M.J. Fisher 定义软件质量为“所有描述计算机软件优秀程度的特性的组合”。

质量特性及其组合，是软件开发与维护中的重要考虑因素

- 为满足软件的各项精确定义的功能、性能需求，符合文档化的开发标准，需要相应地给出或设计一些质量特性及其组合。
- 如果这些质量特性及其组合都能在产品中得到满足，则这个软件产品质量就是高的。

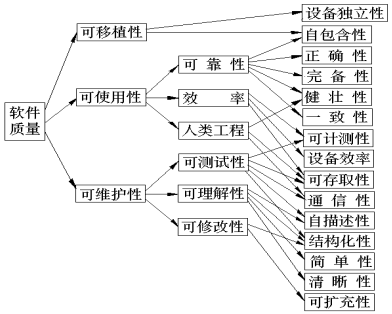
- 软件需求是度量软件质量的基础。不符合需求的软件就不具备质量。
- 标准定义了一组开发准则，用来指导软件人员用工程化的方法来开发软件。如果不遵守这些开发准则，软件质量就得不到保证。
- 软件质量是各种特性的复杂组合。它随着应用的不同而不同，随着用户提出的质量要求不同而不同。

软件质量特性

- 软件质量特性，反映了软件的本质。讨论一个软件的质量，问题最终要归结到定义软件的质量特性。
- 定义一个软件的质量，就等价于为该软件定义一系列质量特性。
- 人们通常把影响软件质量的特性用软件质量模型来描述。

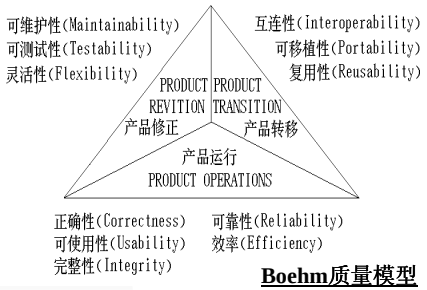
软件质量模型

- 软件质量特性定义成分层模型
- 最基本的叫做基本质量特性，它可以由一些子质量特性定义和度量。
- 二次特性在必要时又可由它的一些子质量特性定义和度量。
- 1976年 Boehm质量模型
- 1979年 McCall质量模型
- 1985年 ISO质量模型

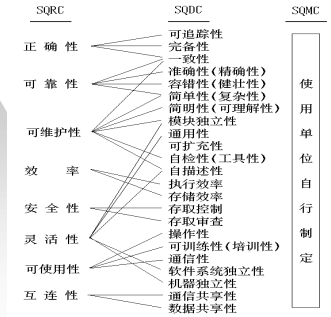


ISO的软件质量评价模型

- 按照ISO/TC97/SC7/WG3/1985-1-30/N382，软件质量度量模型由三层组成
- 软件质量需求评价准则（SQRC）
- 软件质量设计评价准则（SQDC）
- 软件质量度量评价准则（SQMC）
- 高层和中层建立国际标准，低层可由各使用单位视实际情况制定



Boehm质量模型



1991年 ISO质量特性国际标准（ISO/IEC9126）

- **质量特性**：功能性、可靠性、可维护性、效率、可使用性、可移植性
- **推荐21个子特性**：适合性 准确性 互用性 依从性 安全性 成熟性 容错性 可恢复性 可理解性 易学习性 操作性 时间特性 资源特性 可分析性 稳定性 可变更性 可测试性 可安装性 可替换性 适应性 一致性

	功能性	可靠性	可使用性	效 率	可维护性	可移植性
功 能 性		△			△	
可 靠 性				▽		△
可使用性				▽	△	△
效 率		▽			▽	▽
可维护性		△		▽		△
可移植性		▽		▽		

其中，△表示有利影响，▽表示不利影响。

软件质量的度量和评价

- 软件质量特性度量有两类：预测型和验收型。
- 预测度量是利用定量或定性的方法，估算软件质量的评价值，以得到软件质量的比较精确的估算值。
- 验收度量是在软件开发各阶段的检查点，对软件的要求质量进行确认性检查的具体评价值，它是对开发过程中的预测进行评价。

- 预测度量有两种。
- 第一种叫做尺度度量，这是一种定量度量。它适用于一些能够直接度量的特性，例如，出错率定义为：错误数／KLOC／单位时间。
- 第二种叫做二元度量，这是一种定性度量。它适用于一些只能间接度量的特性，例如，可使用性、灵活性等等。

尺度度量检查表

评价准则	度量	需求		设计		编码	
		是／否	值	是／否	值	是／否	值
程序复杂性	每一模块的复杂性度量(McCabe)		□		□		□
	系统复杂性度量 = 各模块复杂性度量之和 / 系统模块数		□		□		□

二元度量检查表

评价准则	度量	需求		设计		编码	
		是／否	值	是／否	值	是／否	值
设计文档的完备性	(1) 无二义性引用（输入／功能／输出）	□		□		□	
	(2) 所有数据引用都可以从一个个外部源定义、计算和取得	□		□		□	
	(3) 所有定义的功能都被使用	□		□		□	
	(4) 所有使用的功能都被定义	□		□		□	
	(5) 对每一个判定点，所有的条件和处理都已被定义	□		□		□	
	(6) 所有被定义、被引用的调用序列的参数一致	□		□		□	

- 通过对照检查项目，确定一种质量特性的有无。
- 例如，在设计和编码阶段的复杂性度量，利用尺度度量方法来做。对模块复杂性的度量采用**McCabe** 环路度量。
- 对于二元度量，可针对检查表中每一项都应给以记分，指定信息存在时记 **“1”**，否则记 **“0”**。表中所有各项的分数相加，即得度量结果。

软件的质量保证

- 质量保证的概念
- 软件质量保证的主要任务
- 质量保证与检验
- 软件质量保证体系
- 质量保证的实施
- 软件的质量设计

质量保证的概念

- 什么是质量保证，它是为保证产品和服务充分满足消费者要求的质量而进行的有计划、有组织的活动。
- 质量保证是面向消费者的活动，是为了使产品实现用户要求的功能，站在用户立场上来掌握产品质量的。
- 软件的质量保证就是向用户及社会提供满意的高质量的产品。

- 软件的质量保证活动也和一般的质量保证活动一样，是确保软件产品从诞生到消亡为止的所有阶段的质量的活动。即为了确定、达到和维护需要的软件质量而进行的所有有计划、有系统的管理活动。

软件质量保证的主要任务

- 为了提高软件的质量和软件的生产率，软件质量保证的主要任务大致可归结为8点。

1. 用户要求定义

- 熟练掌握正确定义用户要求的技术
- 熟练使用和指导他人使用定义软件需求的支持工具
- 重视领导全体开发人员收集和积累有关用户业务领域的各种业务的资料和技术技能。

2. 力争不重复劳动

- 考虑哪些既有软件可以复用
- 在开发过程中，随时考虑所生产软件的复用性。

3. 掌握开发新软件的方法

- 在开发新软件的过程中大力使用和推行软件工程学中所介绍的开发方法和工具。
  - 使用先进的开发技术：如结构化技术、面向对象技术
  - 使用数据库技术或网络化技术
  - 应用开发工具或环境
  - 改进开发过程

4. 组织外部力量协作的方法

- 一个软件自始至终由同一个软件开发单位来开发，也许是最理想的。但在现实中常常难以做到。
- 改善对外部协作部门的开发管理。必须明确规定进度管理、质量管理、交接检查、维护体制等各方面的要求，建立跟踪检查的体制。

5. 排除无效劳动

- 最大的无效劳动就是因需求规格说明有误、设计有误而造成的返工。定量记录返工工作量，收集和分析返工劳动花费数据
- 较大的无效劳动是重复劳动，即相似的软件在几个地方同时开发
- 建立互相交流、信息往来通畅、具横向交流特征的信息流通网

6. 发挥每个开发者的能力

- 软件生产是人的智能生产活动，它依赖于人的能力和开发组织团队的能力。
- 开发者必须有学习各专业业务知识、生产技术和管理技术的能动性。
- 管理者或产品服务者要制定技术培训计划、技术水平标准，以及适用于将来需要的中长期技术培训计划。

7. 提高软件开发的工程能力

- 要想生产出高质量的软件产品必须有高水平的软件工程能力。
- 在软件开发环境或软件工具箱的支持下，运用先进的开发技术、工具和管理方法开发软件的能力。

8. 提高计划和管理质量能力

- 项目开发初期计划阶段的项目计划评价
- 计划执行过程中及计划完成报告的评价
- 将评价、评审工作在工程实施之前就列入整个开发工程的工程计划中
- 提高软件开发项目管理的精确度

质量保证与检验

- 其一是切实搞好开发阶段的管理，检查各开发阶段的质量保证活动开展得如何；
- 其二是预先防止软件差错给用户造成损失。
- 为了确保每个开发过程的质量，防止把软件差错传递到下一个过程，必须进行质量检验。

质量检验的原则

- 用户要求的是产品所具有的功能，这是“真质量”。靠质量检验，一般检查的是“真质量”的质量特性。
- 能靠质量检验的质量特性，即使全数检验，也只是代表产品的部分质量特性。
- 必须在各开发阶段对影响产品质量的因素进行切实的管理，认真检查实施落实情况。

- 当开发阶段出现异常时，要从质量特性方面进行检验，看是否会给后续阶段带来影响。
- 虽然各开发阶段进展稳定，但由于工程能力不足，软件产品不能满足用户要求的质量。这时可通过检验对该产品做出评价，判断是否能向用户提供该产品。
- 要以一定的标准检验产品，根据产品的质量特性，检查各个过程的管理状态。

软件质量保证体系

- 软件的质量保证活动，是涉及各个部门的部门间的活动。
- 例如，如果在用户处发现了软件故障，产品服务部门就应听取用户的意见，再由检查部门调查该产品的检验结果，进而还要调查软件实现过程的状况，并根据情况检查设计是否有误，不当之处加以改进，防止再次发生问题。

- 为了顺利开展以上活动，事先明确部门间的质量保证业务，确立部门间的联合与协作的机构十分重要，这个机构就是质量保证体系。
  - 必须明确反馈途径。
  - 必须明确各部门的职责。
  - 必须确定保证系统运行的方法、工具、有关文档资料，以及系统管理的规程和标准。

- 必须明确决定是否可向下一阶段进展的评价项目和评价准则。
- 必须不断地总结系统管理的经验教训，能够修改系统。
- 制定质量保证计划，在计划中
  - 确定质量目标
  - 确定在每个阶段为达到总目标所应达到的要求
  - 确定进度安排
  - 确定所需人力、资源和成本等。

软件质量保证规程和技术准则

- 规定在项目的哪个阶段进行评审及如何评审；
- 规定在项目的哪个阶段应当产生哪些报告和计划；
- 规定产品各方面测试应达到的水平。
- 在每次评审和测试中发现的错误如何修正；

- 描述希望得到的质量度量；
- 说明各种软件人员的职责，规定为了达到质量目标他们必须进行哪些活动。
- 建立
  - 在各阶段中执行质量评价的质量评价和质量检查系统
  - 有效运用质量信息的质量信息系统，并使其运行。

质量保证的实施

- 软件质量保证的实施需要从纵向和横向两个方面展开。
  - 要求所有与软件生存期有关的人员都要参加
  - 要求对产品形成的全过程进行质量管理
- 这要求整个软件部门齐心协力，不断完善软件的开发环境。此外还需要与用户共同合作。

质量目标与度量

- 为了开发高质量的软件，需要明确软件的功能，明确软件应达到什么样的质量标准，即质量目标。
- 为了达到这个目标，在开发过程中的各个阶段进行检查和评价。
- 在做质量评价时，需要有对质量进行度量的准则和方法。
- 需要有在软件生存期中如何使用这些准则和方法的质量保证步骤，以及提高该项作业效率的工具

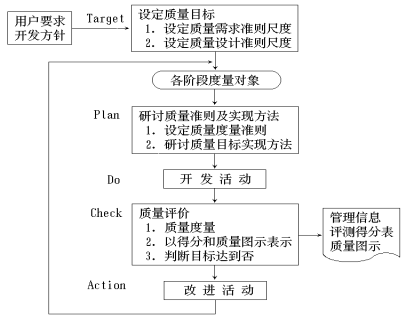
软件质量度量和保证的条件

- 适应性：适应各种用户、软件类型
- 易学性：不需要特殊技术，易掌握
- 可靠性：同个软件的评价结果一致
- 针对性：设计阶段就确立质量目标，在各个阶段实施落实。
- 客观性：
- 经济性：

质量保证活动的实施步骤：

- **Target**：以用户要求和开发方针为依据，对质量需求准则、质量设计准则的各质量特性设定质量目标。
- **Plan**：设定适合于被开发软件的评测检查项目(质量评价准则)。研讨实现质量目标的方法或手段。
- **Do**：制作高质量的规格说明和程序。在接受质量检查前先做自我检查。

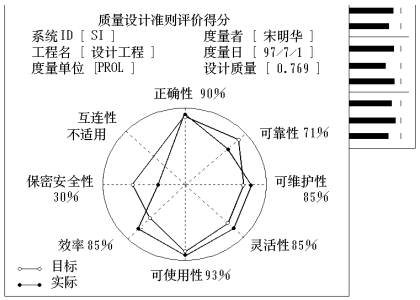
- **Check**：以Plan阶段设定的质量评价准则进行评价。计算结果用质量图的形式表示出来。比较评价结果的质量得分和质量目标，看其是否合格。
- **Action**：对评价发现的问题进行改进活动，如果实现并达到了质量目标就转入下一个工程阶段。这样重复“Plan”到“Action”的过程，直到整个开发项目完成。



质量设计准则评价得分

系统ID [SI] 度量者 [宋明华]
工程名 [设计工程] 度量日 [97/7/1]
度量单位 [PROL] 设计质量 [0.769]

质量需求准则	质量设计准则	得分	图 示
正确性	可追踪性	0.916	
	完备性	0.916	
	一致性	0.872	
可靠性	一致性	0.872	
	简单性	0.812	
	容错性	0.500	
	准确性	0.616	
可维护性	一致性	0.872	
	模块性	0.806	



软件的质量设计

- 质量特性转换为软件的内部结构
- 在软件定义阶段，必须定义对软件的质量需求。即确定软件的质量特性及必需的评价准则，并定量地设定其必须达到的质量水平
- 在以后软件开发的每一阶段结束时，要算出评价的分数，然后与目标值加以对照，以评估在这一阶段开发的软件质量是否达到要求。

- 为了实现规定的质量特性，就需要把这些质量特性转换为软件的内部结构的特性。
- 例如，软件质量需求中的“性能”，可以转换成软件内部结构中的构成元素，即每一个程序模块和物理数据各自应具有的性能特性。这些性能特性的累积就形成外部规格中的性能特性。

软件的结构特性与评价标准

- 结构特性 逻辑数据层次
- 评价标准
  - 全部数据元素定义完毕
  - 所有层次的操作符定义完毕
- 结构特性 功能层次
- 评价标准
  - 全部功能元素定义完毕
  - 所有层次的操作符定义完毕

- 结构特性 逻辑数据与功能的对应关系
- 评价准则
  - 所有数据都与功能对应
  - 所有功能元素都与数据对应
  - 逻辑数据与功能的相互关系个数（局部）

- 结构特性 物理数据层次
- 评价准则
  - 全部数据元素定义完毕
  - 物理数据之间的所有指针定义完毕
  - 上述指针都具有层次性

- 结构特性 模块层次
- 评价准则
  - 所有模块定义完毕
  - 模块之间所有控制关系定义完毕
  - 上述关系都是标准过程调用形式
  - 各层次上的模块大小适当

- 结构特性
  物理数据与模块的对应关系
- 评价准则
  - 所有物理数据都与模块对应
  - 所有模块都与物理数据对应
  - 对应于一个物理数据的模块数（以一对一为好）

- 结构特性
  逻辑数据与物理数据的对应关系
- 评价准则
  - 所有逻辑数据都与物理数据对应
  - 对应于一个物理数据的逻辑数据数（以一对一为好）

- 结构特性
  功能与模块的对应关系
- 评价准则
  - 所有功能都与模块对应
  - 对应模块的功能个数（以一对一为好）

软件可靠性

- 软件生存期与软件寿命的关系
- 在软件工程中常用的定义
- 软件可靠性定义
- 测试中的可靠性分析
- 测试精确度和测试覆盖度的评价

软件生存期与软件寿命的关系

- 一切有生命的东西都有一个“寿命”
- 这个概念也可以延伸到对非生命产品的质量评价上来。例如一个电子产品的寿命就是指该产品从出厂直到丧失使用价值的持续时间。
- 从软件工程的角度来说，软件产品的寿命是指软件的整个生存期。

- 从软件用户的角度来看，更关心的是软件在交付使用后的情况如何。
- 希望用一个指标平均失效间隔时间MTBF(MeanTime Between Failure)来表明，在规定的要求和条件下，能在多大的程度上依赖这个软件来完成任务。
- 我们把在使用期间软件能够正常工作的持续时间叫做软件的使用寿命。

- 软件的使用寿命与输入环境有关。
- 例如，有一个存在缺陷的编译程序，当用于学生做简单练习时，MTBF可能很长。而做一个大的课题时，由于程序连续出错，MTBF就会变得很短。
- MTBF可以看做是对软件可靠性做估计的样本数据，但不能看做是依据。

- “错误”这一术语。在没有特别加以说明的情况下，这是一个泛用的、模糊的概念。
- 它指的可能是bug(设计中的差错)、fault(故障)、error(错误)、failure(失效)、crash(重大事故)、problem(疑问)等。
- 在汉译中，这些术语的使用更加混乱。

在软件工程中常用的定义

- 故障(fault)：软件的内在缺陷。这些缺陷可在生存期各个阶段被引入。
- 错误(error)：故障在一定的环境条件下的暴露，导致系统在运行中出现了不正常、不正确、不按规范执行的状态，称为软件出错。
- 失效(failure)：对错误不做任何修正和恢复， 导致系统的输出不满足用户要求，称为软件的一次失效。

- 以上定义的故障、错误和失效，分别代表了广义的“错误”在不同的条件下所对应的术语。
- 它们可以理解为：设计者的失误—导致系统中留有错误的设计——缺陷或“故障”(fault)，这些故障导致系统的错误执行——错误(error)，由于错误导致系统的错误输出——失效(failure)。

- 故障是物理地或静态地存在的
- 失误、错误和失效都是系统的一种动态的转瞬即逝的现象
- 软件发生失效标志着软件一次使用寿命的结束
- 发生过失效的软件通常仍然是可用的。只有当软件频繁失效，或者公认已经“过时”了的时侯，软件才被废弃，意味着当前这一版本软件使用寿命的终结。

软件故障产生原因

- 支持软件工作的基本条件(除硬件外的操作系统、数据库管理系统、编译程序、微代码等)的缺陷
- 软件设计不当
- 加入了允许范围之外的输入

软件可靠性的定义

- 软件可靠性是软件在给定的时间间隔及给定的环境条件下，按设计要求，成功地运行程序的概率。
- 环境条件—指的是软件的使用环境。无论是什么软件，如果不对它的使用环境加以限制，都是会失效的。这种失效的数据，不能用来度量软件的可靠性。

- 规定的时间—在定义中，一般采用“运行时间”t 作为时间的尺度。因
- 具体要处理的问题是多种多样的
- 其对应的输入环境是随机
- 程序中相应程序路径的选取也是随机的
- 软件的失效也是随机的
- 应当把运行时间t当作随机变量来考虑。

- 规定的功能—在考虑软件可靠性时，首先应当明确软件的功能是什么，哪些功能是主要的，哪些功能是次要的。一般从软件需求分析说明书和设计说明书中可以了解这些情况。
  - 由于功能不同，失效带来的损失就不一样。因此，还要明确哪些失效是致命的，哪些失效是非致命的，哪些又是容易修复的。此外，还要明确，怎样才算是完成了一个规定的功能。

- 成功地运行程序—是指不仅程序能正确地运行，满足用户对它的功能要求， 而且当程序一旦受到意外的伤害，或系统故障时，能尽快恢复，仍能正常地运行。

测试中的可靠性分析

- 在软件开发的过程中，利用测试的统计数据，估算软件的可靠性，以控制软件的质量是至关重要的。
- 推测错误的产生频度，即推测错误产生的时间间隔
- 推测残留在程序中的错误数
- 评价测试的精确度和覆盖率

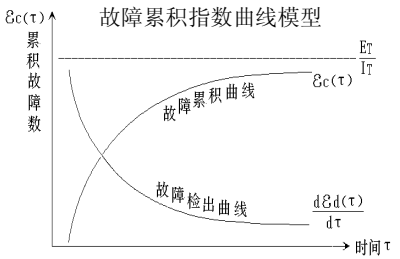
推测错误的产生频度

- 估算错误产生频度的一种方法是估算平均失效等待时间MTTF (Mean Time To Failure)
- MTTF估算公式(Shooman模型)

$$\mathrm{MTTF} = \frac{I_T}{K(E_T - E_C(t))}$$

$$E(t) = E_T(1 - e^{-K \cdot t})$$

故障累积指数曲线模型



估算软件中故障总数E_T的方法

利用Shooman模型估算程序中原来错误总量E_T—瞬间估算

$$\frac{E_C(t_1)}{t_1} = \frac{1}{\mathrm{MTTF}_1} = K\left(\frac{E_T}{I_T} - \frac{E_C(t_1)}{I_T}\right)$$

$$\frac{E_C(t_2)}{t_2} = \frac{1}{\mathrm{MTTF}_2} = K\left(\frac{E_T}{I_T} - \frac{E_C(t_2)}{I_T}\right)$$

解此方程组

$$\hat{E}_T = \frac{t_2 - t_1}{t_2 \cdot E_C(t_1) - t_1 \cdot E_C(t_2)} \cdot E_C(t_1) \cdot E_C(t_2)$$

$$\hat{K} = \frac{I_T \cdot E_C(t_1)}{t_1 \cdot (E_T - E_C(t_1))}$$

利用最小二乘法进行程序原有错误数E_T及K的估算

- 由失效率 $\lambda = K\left(\frac{E_T}{I_T} - \frac{E_c(t)}{I_T}\right)$
- 整理得 $\frac{E_c(t)}{I_T} = \frac{E_T}{I_T} - \frac{\lambda}{K}$
- 若对程序进行若干次不同的功能测试，可得到一系列实验数据

$E_c(t_i),\ \lambda(t_i),\ i = 1, 2, \ldots, n$

- 令 $-\frac{1}{K} = a,\quad \frac{E_T}{I_T} = b,$

$$\frac{E_C(t_i)}{I_T} = \varepsilon_i,\quad \lambda(t_i) = \lambda_i$$

- 有 $\varepsilon_i = a\lambda_i + b,\quad i = 1, 2, \cdots, n$

- 用最小二乘法解此方程组，可解出a、b的估计值
- 最后得到 K, E_T 的估计值

$$\hat{K} = -\frac{1}{\hat{a}},\quad \hat{E}_T = I_T \times \hat{b}$$

利用植入故障法估算程序中原有故障总数E_T —捕获—再捕获抽样法

- 设N_s是在测试前人为地向程序中植入的故障数(称播种故障)，n_s是经过一段时间测试后发现的播种故障的数目，n是在测试中又发现的程序原有故障数。设测试用例发现植入故障和原有故障的能力相同，则程序中原有故障总数 N ($=E_T$)估算值为

$$N = \frac{N_s}{n_s} n$$

Hyman分别测试法

- 由两个测试员同时互相独立地测试同一程序的两个副本，用 t 表示测试时间，记 $t=0$时，程序中原有故障总数是 B_0；$t=t_1$ 时，测试员甲发现的故障总数是 B_1；测试员乙发现的故障总数是 B_2；其中两人发现的相同故障数目是 bc；两人发现的不同故障数目是 bi。

- 在大程序测试时，头几个月两个测试员测试的结果应当比较接近，bi不是很大。这时有

$$B_0 = \frac{B_1 \cdot B_2}{bc}$$

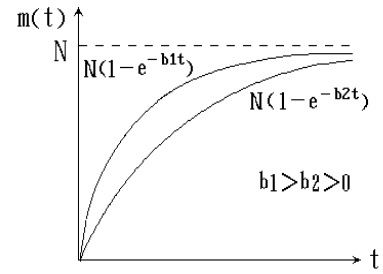
- 如果bi比较显著，应当每隔一段时间，由两个测试员再进行分别测试，分析测试结果，估算B_0。如果bi减小，或几次估算值的结果相差不多，则B_0作为原有错误总数的估算值。

测试精确度和测试覆盖度的评价

- 在软件测试过程中累积发现的故障数，可用带有平均值函数 $m(t)$ 的非齐次泊松过程(NHPP)来描述：

$$m(t) = N\left(1 - e^{-bt}\right)$$

- 其中，N是在测试中可能发现的故障总数，b是故障发现率。
- 当N一定时，b越大，在短期内发现的故障越多。



- *N* 可以认为是当测试时间无限延长时估计可能发现的故障总数。由于
  - 测试的不完全，在某些很难发现的故障未发现前就可能结束测试
  - 若程序中潜在的故障较少，则参数*M*的估计误差较大
- 因此，只用测试中累积发现的故障数来评价测试是不够的。需要从测试的量的方面和质的方面，全面地评价测试。

测试结束时软件产品质量水准

- SPQL (Software Product Quality Level) 用如下公式度量：

  SPQL ＝ Ac×Cv
- 其中，Ac (Test Accuracy) 是测试的精确度，它反映了测试的质量；Cv (Test Coveragy) 是测试的覆盖度，它反映了测试的数量。

- 测试质量的度量可以靠测试的故障捕捉率和遗漏率来衡量。
- 测试数量的度量指标是执行的测试用例数、确认的程序路径数等等；

测试精确度Ac

- 表明在测试的过程中以多大的把握捕捉了软件中潜在的故障。
- 测定Ac，需要预先植入播种故障，然后通过测试，根据播种故障的捕捉率来推测原有故障的捕获率。

$$Ac=\frac{\text{测试可能发现的播种故障数}}{\text{预先植入的播种故障总数}} \qquad (0\leqslant Ac\leqslant 1)$$

- 用n_s表示经过相当长时间测试可能发现的播种故障数，用N_s表示测试对象软件内预先埋设的播种故障总数,用平均值为$m(t)$的NHPP模型描述测试时发现播种故障的过程
- $m(t)$的收敛值 $m(\infty)=N$
- 测试精确度Ac的推测值：

$$A_c=\frac{n_s}{N_s}=\frac{m(\infty)}{N_s}=\frac{N}{N_s}$$

- 若设测试过程中到时刻 t_i 能发现的累积播种故障总数为 y_i，则在测试期间可得到一连串数据

  $(t_0, 0), (t_1, y_1), \ldots\ldots, (t_m, y_m)$
- 可得到一组方程：

$$y_i=N(1-e^{-bt_i}),\ i=0,\cdots,n$$

- 应用最小二乘法可得到参数N与b的估计值，并得到测试精确度Ac。

测试覆盖率Cv

- 表明在整个测试期间发现软件内潜在故障的可能性有多大。
- 可通过被测试对象软件内潜在的原有故障的捕捉率来测定的。

$$Cv=\frac{\text{测试过程中已发现的原有故障数}}{\text{测试可能发现的总原有故障数}} \qquad (0\leqslant Cv\leqslant 1)$$

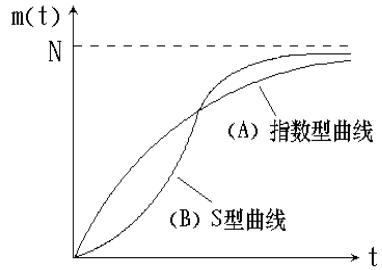
- 测试过程中已发现原有故障总数为n_0(实测值)，经过相当长时间测试后可能发现的原有故障总数为N_0，
- 采用平均值函数$m(t)$的NHPP模型描述测试发现原有故障的过程
- $m(t)$的收敛值$m(\infty)=Nc$
- 测试覆盖率Cv的推测值：

$$Cv=\frac{n_0}{N_0}=\frac{n_0}{m(\infty)}=\frac{n_0}{Nc}$$

- 测试开始后，由于测试员对程序和测试环境不熟悉，造成拖期。
- 为描述这种情形，对原来NHPP的指数型平均值函数加以改造：

$$m(t)=N(1-e^{-b(t-d)})$$

- 它是把原来的指数型平均值函数在时间轴上平移而得到的结果，是具有时间延迟的NHPP模型。



- 测试员从发现错误征兆到确认错误，需要反复执行程序，以再现错误，造成时间拖延。
- 因此，在使用测试结果进行软件质量评价时，只用指数型的NHPP的平均值曲线(A)是不够的。实测结果多是如(B)所示的S型曲线。

$$m(t)=N(1-(1+bt)e^{-bt})$$

- 实验表明：
  - 对于一般功能单纯的小规模的程序模块，具有时间延迟的NHPP模型比较合适；
  - 对于功能比较复杂的程序模块，S型NHPP模型比较合适；
  - 对于80000行以上的程序，最基本的指数型NHPP模型比较合适。

软件配置管理

- 在软件建立时变更是不可避免的，因为在进行变更前没有仔细分析，或没有进行变更控制，变更加剧了项目中软件人员之间的混乱。
- 协调软件开发使得混乱减到最小的技术叫做配置管理。
- 配置管理是一组标识、组织和控制修改的活动，目的是使错误达到最小并最有效地提高生产率。

软件配置管理的概念

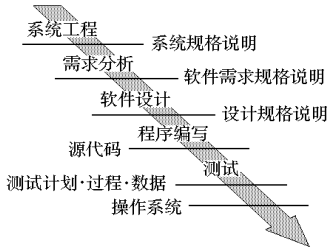
- 软件配置管理，简称SCM，是一种“保护伞”活动，它应用于整个软件工程过程。
- SCM活动的目标是为了
  (1) 标识变更；
  (2) 控制变更；
  (3) 确保变更正确地实现；
  (4) 向其他有关的人报告变更。

- 在软件工程过程中产生的所有信息项（文档、报告、程序、表格、数据）构成了软件配置。
- 软件配置是软件的具体形态在某一时刻的瞬时影像。
- 随着软件工程过程的进展，软件配置项(SCI)数目快速增加。系统规格说明可繁衍出软件项目实施计划和软件需求规格说明。它们又依次繁衍出建立信息层次的其它文档。

基线 (Baseline)

- 基线是软件生存期中各开发阶段末尾的特定点，又称里程碑。
- 由正式的技术评审而得到的SCI协议和软件配置的正式文本才能成为基线。
- 基线的作用是把各阶段工作的划分更加明确化，以便于检验和肯定阶段成果。

软件开发各阶段的基线

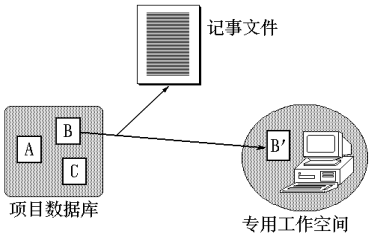


项目数据库

- 一旦一个SCI成为基线，就把它存放到项目数据库中。
- 当软件组织成员想要对基线SCI进行修改时，把它从项目数据库中复制到该工程师的专用工作区中。
- 例如，把一个名为B的SCI从项目数据库复制到工程师的专用工作区中。工程师在B'（B的副本）上完成要求的变更，再用B'来更新B。

- 有些系统中把这个基线SCI锁定。
- 在变更完成、评审和批准之前，不许对它做任何操作。

基线SCI和项目数据库



软件配置项 SCI

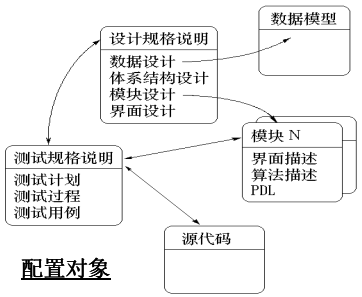
- 软件配置管理的对象就是SCI—软件配置项。
  - 系统规格说明
  - 软件项目实施计划
  - 软件需求说明
  - 可执行的原型
  - 初步的用户手册
  - 设计规格说明

- 源代码清单
- 测试计划和过程、测试用例和测试结果记录
- 操作和安装手册
- 可执行程序（可执行程序模块、连接模块）
- 数据库描述（模式和文件结构、初始内容）
- 正式的用户手册
- 维护文档（软件问题报告、维护请求、工程变更次序）

- 软件工程标准
- 项目开发总结
- 除以上所列SCI以外，许多软件工程组织还把配置控制之下的软件工具列入其中，即编辑程序、编译程序、其它CASE工具的特定版本。因为要使用这些工具来生成文档、程序和数据，如果编译程序的版本不同，可能产生的结果也不同。

配置对象

- 在实现SCM时，把SCI组织成配置对象，在项目数据库中用一个单一的名字来组织它们。
- 一个配置对象有一个名字和一组属性，并通过某些联系“连接”到其它对象。
- 每个对象与其它对象的联系用箭头表示。箭头指明了一种构造关系。



配置对象

- 双向箭头则表明一种相互关系。如果对“源代码”对象作了一个变更，软件工程师就可以根据这种相互关系确定，其它哪些对象（和SCI）可能受到影响。

软件配置管理的任务

- 软件配置管理（SCM）的任务是：
  - 标识单个的SCI
  - 标识和管理软件各种版本
  - 控制变更
  - 审查软件配置
  - 报告所有加在配置上的变更。

配置标识

- 一方面随着软件生存期的向前推进，SCI的数量不断增多。
- 整个软件生存期的软件配置就象一部不断演变的电影，而某一时刻的配置就是这部电影的一个片段。
- 为了方便对软件配置的各个片段（SCI）进行控制和管理，不致造成混乱，首先应给它们命名。

对象类型

- 基本对象：是由软件工程师在分析、设计、编码和测试时所建立的文本单元。例如，基本对象可能是需求规格说明中的一节，一个模块的源程序清单、一组用来测试一个等价类的测试用例。
- 复合对象：是基本对象或其它复合对象的一个收集。

- 对象标识：
  （名字、描述、资源、实现）
- 对象的名字明确地标识对象。
- 对象描述包括：SCI类型（如文档、程序、数据）、项目标识、变更和／或版本信息。
- 资源包括由对象产生的、处理的、引用的或其它需要的一些实体。
- 基本对象的实现是指向文本单元的指针，复合对象的实现为null。

命名对象之间的联系

- 对象的层次关系：一个对象可以是一个复合对象的一个组成部分，用联系< ***is part of*** >标识。

  E-R *diagram* 1.4 < *is part of* > *data model*;

  ***data model* < *is part of* > *Design Specification*;**
- 就可以建立SCI的一个层次。

- 对象的相互关联关系：对象跨越对象层次的分支相互关联。这些交叉的结构联系表达方式如下：

  data model <interrelated> data flow model;

  (两个复合对象之间的相互联系)

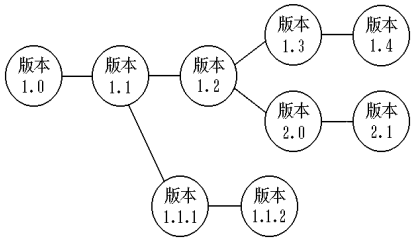
  data model <interrelated> test case class m;

  (一个复合对象与一个特定的基本对象之间的相互联系)

演变图

- 整个软件工程过程中所涉及的软件对象都必须加以标识。
- 在对象成为基线以前可能要做多次变更，在成为基线之后也可能需要频繁地变更。
- 对于每一配置对象都可以建立一个演变图，用演变图记叙对象的变更历史。

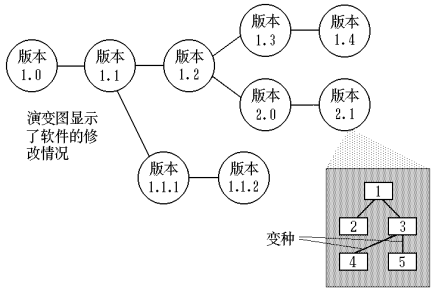
演变图



- 在某些工具中，当前保持的只是最后版本的完全副本。
- 为了得到较早时期(文档或程序)的版本，可以从最后版本中“提取”出(由工具编目的)变更，使得当前配置直接可用，并使得其它版本也可用。

版本控制

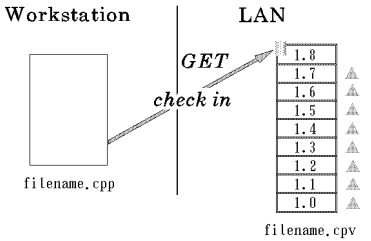
- 版本控制是SCM的基础，它管理并保护开发者的软件资源。
- 版本控制管理在软件工程过程中建立起配置对象的不同版本。
- 版本管理可以把一些属性结合到各个软件版本上。
- 通过描述所希望的属性集合来确定（或构造）所想要的配置。
- 使用演变图来表示系统的不同版本。



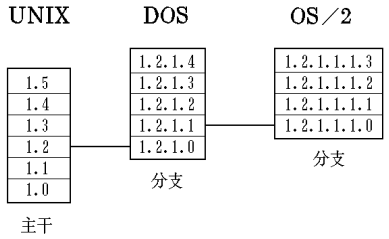
- 图中的各个结点都是聚合对象，是一个完全的软件版本。
- 软件的每一版本都是SCI（源代码、文档、数据）的一个收集，且各个版本都可能由不同的变种组成。
- 例如，一个简单的程序版本由1、2、3、4和5等部件组成。其中部件4在软件使用彩色显示器时使用，部件5在软件使用单色显示器时使用。因此，可以定义版本的两个变种。

版本管理的主要任务

- 集中管理档案，安全授权机制：
  - 版本管理的操作将开发组的档案集中地存放在服务器上，经系统管理员授权给各个用户。
  - 用户通过登入（***check in***）和检出（***check out***）的方式访问服务器上的文件，未经授权的用户无法访问服务器上的文件。



- 软件版本升级管理：
  - 每次登入时，在服务器上都会生成新的版本。
  - 任何版本都可以随时检出编辑，同一应用的不同版本可以像树枝一样向上增长。



- 加锁功能：
  - 目的是在文件更新时保护文件，避免不同用户更改同一文件时发生冲突。
  - 某一文件一旦被登入，锁即被解除，该文件可被其它用户使用。
  - 在更新一个文件之前锁定它，避免变更没有锁定的项目源文件。

- 在文件登入和检出时，需要注意登入和检出的使用：
  - 当需要修改某个小缺陷时，应只检出完成工作必需的最少文件；
  - 需要对文件变更时，应登入它并加锁，保留对每个变更的记录；
  - 应避免长时间地锁定文件。如果需要长时间工作于某个文件，最好能创建一个分支，并在分支上做工作。

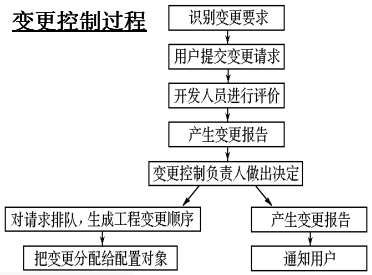
- 如果需要做较大的变更，可有两种选择：

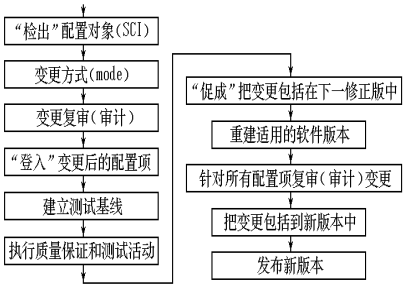
  a. 将需要的所有文件检出并加锁，然后正常处理；

  b. 为需要修改的所有分支创建分支，把变更与主干“脱机”，然后把结果合并回去。

变更控制

- 软件生存期内全部的软件配置是软件产品的真正代表，必须使其保持精确。
- 软件工程过程中某一阶段的变更，均要引起软件配置的变更，这种变更必须严格加以控制和管理，保持修改信息。
- 变更控制包括建立控制点和建立报告与审查制度。





- 在此过程中，首先用户提交书面的变更请求，详细申明变更的理由、变更方案、变更的影响范围等。
- 然后由变更控制机构确定控制变更的机制、评价其技术价值、潜在的副作用、对其它配置对象和系统功能的综合影响以及项目的开销、并把评价的结果以变更报告的形式提交给变更控制负责人（最终决定变更状态和优先权的某个人或小组）。

- 对每个批准了的变更产生一个工程变更顺序（***ECO***），描述进行的变更、必须考虑的约束、评审和审计的准则等。
- 要做变更的对象从项目数据库中检出（***check out***），对其做出变更，并实施适当的质量保证活动。然后再把对象登入（***check in***）到数据库中并使用适当的版本控制机制建立软件的下一版本。

软件变更有两类不同情况：

- 为改正小错误需要的变更。它是必须进行的，通常不需要从管理角度对这类变更进行审查和批准。但是，如果发现错误的阶段在造成错误的阶段的后面，例如在实现阶段发现了设计错误，则必须遵照标准的变更控制过程，把这个变更正式记入文档，把所有受这个变更影响的文档都做相应的修改。

- 为了增加或者删掉某些功能、或者为了改变完成某个功能的方法而需要的变更。这类变更必须经过某种正式的变更评价过程，以估计变更需要的成本和它对软件系统其它部分的影响。
  - 如果变更的代价比较小且对软件系统其它部分没有影响，或影响很小，通常应批准这个变更。

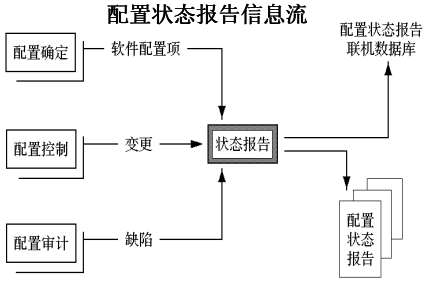
  - 如果变更的代价比较高，或者影响比较大，则必须权衡利弊，以决定是否进行这种变更。
  - 如果同意这种变更，需要进一步确定由谁来支付变更所需要的费用。如果是用户要求的变更，则用户应支付这笔费用；否则，必须完成某种成本／效益分析，以确定是否值得做这种变更。

- 这种变更报告和审查制度，对变更控制来说起了一个安全保证作用。
- 在一个SCI成为基线之前，可以对所有合理的项目和技术申请进行非正式的变更；
- 一旦某个SCI经过正式的技术评审并得到批准，它就成了基线。以后如果需要对它变更，就必须得到项目负责人的批准，或者必须得到变更控制负责人的批准。

配置状态报告

- 为了清楚、及时地记载软件配置的变化，需要对开发的过程做出系统的记录，以反映开发活动的历史情况。这就是配置状态登录的任务。
- 登录主要根据变更控制小组会议的记录，并产生配置状态报告。
- 对于每一项变更，记录：发生了什么？为什么会发生？谁做的？什么时侯发生的？会有什么影响？

配置状态报告信息流



- 每次新分配一个SCI，或更新一个已有SCI的标识，或一项变更申请被变更控制负责人批准，并给出了一个工程变更顺序时，在配置状态报告中就要增加一条变更记录条目。
- 一旦进行了配置审计，其结果也应该写入报告之中。

- 配置状态报告可以放在一个联机数据库中，以便软件开发人员或者软件维护人员可以对它进行查询或修改。此外在软件配置报告中新登录的变更应当及时通知给管理人员和软件工程师。
- 配置状态报告对于大型软件开发项目的成功起着至关重要的作用。避免了可能出现的不一致和冲突。

配置审计

- 软件的完整性，是指开发后期的软件产品能够正确地反映用户要求。
- 软件配置审计的目的就是要
  - 证实整个软件生存期中各项产品在技术上和管理上的完整性。
  - 确保所有文档的内容变动不超出当初确定的软件要求范围。使得软件配置具有良好的可跟踪性。

- 软件配置审计是软件变更控制人员掌握配置情况、进行审批的依据。
- 软件的变更控制机制通常只能跟踪到工程变更顺序产生为止。为确认变更是否正确完成？一般可以用以下两种方法去审查：
  - 正式技术评审
  - 软件配置审计

- 正式的技术评审着重检查已完成修改的软件配置对象的技术正确性，
- 评审者评价SCI，决定它与其它SCI的一致性，是否有遗漏或可能引起的副作用。
- 正式技术评审应对所有的变更进行，除了那些最无价值的变更之外。
- 软件配置审计作为正式技术评审的补充，评价在评审期间通常没有被考虑的SCI的特性。