第一章 概 论

计算机系统工程分为硬件和软件两大范畴。

计算机硬件的工程技术由电子设计发展而来，并且在过去的30多年中已经达到了相当成熟的状态。硬件设计技术已经很好地建立起来，硬件的制造方法一直在不断地改进，可靠性已是一种现实的要求，而不再是一种朴素的愿望。

而在软件工程技术方面，在以计算机为基础的系统中，软件已成为最难设计、最少可能成功(指在时间上以及成本方面)，并且管理起来最危险的系统成分。随着以计算机为基础的系统在数量、复杂程度和应用方面的激增，对软件的需要却在不断增加，因此促使供求矛盾日趋激化。

世界上第一台电子计算机诞生以后，手工的程序设计方法就受到了严厉的指责，许多人都认为它严重地限制和妨碍了计算机的应用。从那时以来，计算机硬件经过了从第一代至第四代的发展，在硬件制造技术方面，人类已经获得了巨大的进展。现在，又在进行新一代计算机的研制工作。随着计算机系统越来越庞大，它的应用范围越来越广泛，计算机的软件系统也变得越来越复杂，随之而来的问题就是：大的软件系统成本高、可靠性差，甚至有时人的大脑已无法理解、无法驾驭人类本身所创造出来的复杂逻辑系统。例如：在60年代，美国IBM公司生产的OS 360花费了5000人-年的巨大代价，但是，由于它太庞大，OS 360变得相当不可靠，平均每次修改以后的新版本都大约存在1000个左右的错误，并且有理由认为这是一个常数。另外，美国空军的范登堡中心在60年代后期发生过多次导弹试射失败的事故，事后检查几乎都是由于计算机的软件有错误而造成的。因此，由于人类大脑对复杂逻辑系统理解与控制的局限性，对先进的计算机硬设备的应用施加了一种无形的束缚。人类智力的不完善和不一致，在软件的开发过程中，势必会产生大量的错误。每一个这样的错误都被看作是与整个系统无关的单个逻辑缺陷，但是，经验证明：如果将它们看作一个整体的话，那么，它们就都涉及到一个共同的问题，即所谓的“软件危机”。许多软件系统的复杂程度已使它们变得无法管理，其自然的结果就是它们的开发周期延长，成本增加，可靠性降低。

软件危机是指在计算机软件开发中所遇到的一系列问题，而不仅限于指“不能正确地工作”的软件。确切地说，软件危机包含和下列问题有关的问题：我们怎样开发软件？我们怎样维护现有的、容量又在不断增加的软件？我们怎样做才能满足对软件不断增长的需求？

软件危机以许多问题为表征，而负责软件开发的管理人员则往往将注意力集中于“底线”问题：进度和成本的估计往往很不精确。实际中不乏这样的例子：成本超过计划的一个数量级；进度一拖就是几个月甚至几年。许多软件开发的困难明显表现在以下方面：

- 人们还没有时间去收集关于软件开发过程的数据。由于缺乏历史数据作为指南，所有关于进度和成本的估算都十分粗略，因为没有切实的生产率指标，我们无法精确评价新工具、新技术或新标准的效能。
- 经常发生用户不满足于"完全的"软件系统。软件开发项目往往在只有模糊的用户要求指示情况下就着手进行。用户与开发者之间的沟通常常是很糟的。
- 软件质量常常是很可疑的。人们只是最近才开始理解系统的、技术上完全的测试的重要性。软件可靠性和质量保证的切实定量概念还刚开始出现。
- 现有的软件可能很难维护。软件维护任务耗费了软件经费的主要部分。人们还没有真正把软件的可维护性作为验收的一个重要准则。

与软件系统的不可靠性密切联系的经济上的含意也是十分明显的：大约 1/3～1/2 的开发和维护软件系统的努力，都花费在测试和排错上面。因为在大计算机系统的运行过程中，绝大部分资源都是分配给软件的，而花在克服软件系统的不可靠性上面的直接开销，其总数也就大体上是计算机的全部开销了。进而言之，如果花在软件错误上的非直接开销（如整个计算机系统的收益上的损失等）也计算在内的话，那么，系统的不可靠性所带来的经济上的损失就变得更加惊人。

与软件危机有关的许多问题都起源于软件本身的特点、担负软件开发职责的人员的弱点、以及在软件开发的早期人们对于软件开发实质的种种不切实际的误解。软件危机不会很快消失，认识问题的本质所在及其产生的根源，是走向解决问题的第一步。

计算机软件最终可能成为以计算机为基础的系统发展中的限制因素。由于软件开发、管理和技术问题表现为软件危机，由于对软件和开发软件所需方法的诸多误解，管理问题将继续存在。而由于有了新开发的工具和技术，技术方面的问题可能解决。为了更多、更快、更好地开发计算机软件，在 1968 年前后产生了一种工程的方法——一种集成了具有最好技术的、经过考验的、关于控制的管理技术的方法，即软件工程的方法。

由于软件开发过程是知识密集型的过程，在软件开发过程之前、之中、之后，都要运用大量的知识，它们主要包括：

- 有关软件工程技巧的知识
- 有关程序设计技巧的知识
- 有关程序设计语言的知识
- 有关计算机硬件的知识（即机器知识）
- 有关应用领域的知识
- 有关软件开发历史的知识

但是，只靠现有的软件工程方法并不可能解决软件危机的根本问题。于是，有人又提出了基于知识的软件工程方法的设想，力求将软件工程与知识工程、人工智能技术结合起来，以构造基于知识的软件开发环境。

§1.1 软件可靠性研究的意义和基本概念

计算机硬件的性能价格比以每十年大约1000的速率在增加。信息处理系统的应用日趋广泛，由此刺激了软件工程技术的进步，以应付日益膨胀的软件需求。具体地说，影响到当今软件工程发展的因素主要有：

1. 国际上商业竞争的势态越来越激烈。
2. 信息处理系统的开发成本，以及由于它们发生故障而造成的经济损失也正在增加。
3. 计算技术的发展速度越来越快。
4. 对于信息系统开发的管理变得更加复杂。

绝大多数信息系统被用于商业，而今天商业上竞争的激烈程度使许多信息系统的顾客真正认识到他们对于软件产品的需要。而存在于软件产业界的激烈竞争，又使他们进一步认识到软件产品连同一起提供的有效服务对于他们自身利益的意义。他们曾经很天真地完全信赖提供软件产品的公司或厂家，但现在他们对于软件产品的质量以及厂家提供的服务，也变得越来越苛求和挑剔。软件开发者们应充分而清楚地认识到用户的需要主要集中于这样三个方面：质量要求、交货时间和开发成本。与此同时，软件的开发和运行成本实际上也在增加，且软件产品的大小、复杂性和分布式的程度也都在不断增加。目前已有越来越多的计算机系统由网络连接起来，有许多不同种类的软件同时在运行并相互发生作用，其直接的结果就是使开发成本变得更大。信息系统应用的激增更增加了对上述系统的依赖性，这一依赖性已发展到要依赖于更小、更低级的组织结构，并且以实时方式工作的系统的比例也在增加。于是，发生故障以后造成的损失更大、更具关键性。这样的事例很多，如：飞机公司预订票系统的失败，银行计算机系统的故障，自动飞行控制系统的失灵，导弹控制系统的失误，核电站安全系统的故障，等等，它们造成的经济损失是巨大的，有的甚至是后果不堪设想的。成本还不仅只包含那些直接的开销，而且还包括对产品开发债务风险所承担的经济赔偿以及对公司声誉的赔偿金。而后一种赔偿往往对公司占领市场及所得利润产生戏剧性的影响。

计算技术的飞快变化使得许多信息系统很快就变得过时了，这样淘汰旧产品也会使软件产业承受很大的经济损失。使软件产品很快变得落后的威胁主要来自两个方面：硬件技术的发展和软件技术的进步。因此软件产品的交货时间就变得十分重要，谁也不希望在软件产品还未交付使用之前，运行该软件系统的硬件环境就已被淘汰。人们必须充分认识到将一个新的软件产品打入市场的有效时限变得越来越短暂了。

基于上述原因，一个软件开发公司如果企图同时提供开发周期短而且成本低廉的高质量软件产品，几乎是不可能的。要认识到，在当前科学技术飞速进步的情况下，这种想法

早已陈旧不堪。

随着软件系统的大小及复杂性的增加，对于信息系统开发的管理也变得更加复杂。现在许多软件系统都被划分成许多部件，由不同的公司来进行开发，这样就要求对系统以及它的部件的特性有清晰的了解，并且对于开发期间的进度要有明确的表示。从管理、签约及法律的观点来看，对于所有特性的了解都是重要的。而且其结果是对软件产品特性以及开发期间对那些特性的当前状态的测量和预测，都变得更重要了。

上面已叙述过软件产品最重要的三个特性是质量、成本和进度，它们基本上是面向用户的而不是面向开发者的属性。其中后两个特性比较容易把握，而软件质量的度量则要困难得多。鉴于目前仍然缺乏度量软件质量的有效手段，人们自然把注意力集中于此。事实上，这种对软件产品质量的定量测量手段的缺乏，正是许多软件产品存在重要质量问题的基本原因。

在软件质量的范畴内，或许最重要的固有特性之一就是系统的可靠性，它关心的问题主要是存在于软件产品中的缺陷，而正如 Jones 在 1986 年指出的那样，软件系统中的缺陷代表了程序设计过程中最大的成本因素。软件可靠性关心的是软件本身的功能如何才能最大限度地满足软件用户的要求。

现在有不少人都在热切地追求软件可靠性的目标，不过大多都只停留在理论的抽象上，却很少有人将它表示成为一个程序本身所具有的属性。理论上的抽象是可以做得十分完美的，但是在软件产品开发过程中，实际上为获取更高的软件可靠性目标，其作法通常是以软件产品本身的某些其它特性（如程序大小、程序的运行时间或程序的响应时间、软件产品本身的可维护性，等等）作为代价来换取的；或者是在软件产品的开发过程中，采取诸如：增加开发成本、增加对资源的要求、推迟生产的进度等措施来换取的。B oehm 等人已将人们希望予以考虑的某些一般的软件生产的特性进行了分类。人们在软件产品的生产过程中，一旦想要对这些因素权衡取舍，以取得一个折衷的方案时，软件可靠性就是一个首先必须加以考虑的因素。在某些情形下，软件可靠性甚至是更一般的特性之一。比如说，考虑实时系统的工作情况，要使用户的要求得到满足或提高用户的工作效率，软件可靠性就是一个很重要的因素。

为达到人们期望的软件可靠性指标，必须在软件产品的特性中，以及在软件产品的生产过程的特性中进行适当的取舍。而这样的取舍，首先得由系统设计师们来作。所以关于软件可靠性的定量评估技术，在他们看来就具有特别重要的意义。当然，这些技术对于软件工程师和软件生产的管理人员而言，也是同样重要的。

软件系统在特定的环境（条件）下，在给定的时间内，不发生故障地工作的概率，称为该软件系统的**可靠度**。而这种性质，就称为软件系统的**可靠性**，亦简称为**软件可靠性**。

以 E 表示特定的环境，t 表示给定的时间，设系统从时间 0 开始运行，直到 T 时发生故障，则：

$$R(E,t) = P_r\{T > t | E\}$$

就表示软件系统在特定的环境 E 下的、正常工作到时刻 t 时的概率，其中，T 是从时刻 0 开始，软件系统运行到发生故障时的时间。

$R(E,t)$ 具有下列的性质：

$R(E,0)=1$；即在 0 时刻，系统绝对不发生故障。

$R(E,+\infty)=0$；即在无限远的时刻，系统必定失败。

在时间区间$(0,+\infty)$上，函数 $R(E,t)$是单调下降的。

按照概率论的观点，“软件系统正常工作”是一事件，则它的对立事件“软件系统运行出错”定义了一个故障概率函数：$Q(E,t)$，因而显然有：

$$R(E,t)+Q(E,t)=1.$$

通常，特定的环境 E 要从描述测量的上下文去了解，而毋须确切地给出，因此，可以把可靠度函数 $R(E,t)$简写为 $R(t)$，而把故障概率函数 $Q(E,t)$简写为 $Q(t)$。因此，有：

$$R(t)+Q(t)=1,$$

$$R(t)=1-Q(t)=P_r\{T>t\},$$

其中 T 的意义同上。

硬件可靠性的概念是十分明显的，它充分体现在对元器件的产品质量检查的过程之中。但是，软件产品可靠性定义中的概率性质体现在什么地方呢？不可能说我们在一批生产出来的相同的软件产品中，随机地抽取出若干个来进行测试。一般可以这样说，软件产品可靠性中的概率性质主要体现在输入的选取上。

如果把程序看作输入空间到输出空间的映射，那么程序运行出错就是由于程序没有将某些输入映射到为人们所期望的输出上去。设：输入空间一共有 I 个点，若点 i 为输入时程序运行正确，引入一个程序执行变量 Y_i 如下：

$$Y(i)=\begin{cases}1, & \text{输入点 } i \text{ 时，程序运行正确；}\\ 0, & \text{否则。}\end{cases}$$

在特定的应用中，设 $P(i)$为输入点 i 的概率，则在这一特定的应用中的一次输入导致程序正确运行的概率为：

$$\sum_{i=1}^{I}P(i)Y(i).$$

因此就有：

$$R(t)=\Big[\sum_{i=1}^{I}P(i)Y(i)\Big]^n,$$

其中，n 是在时间区间$(0,t)$内程序总共运行的次数。

值得注意的是：上面我们提及的“一次运行”，也是一个有时很难以确定或量化的含糊概念。

上面所述描述软件可靠度的方法，的确从本质上反映了软件可靠度定义的概率性质，但是它对实际确定软件的可靠度并无任何意义。这主要是因为：第一，输入空间的大小 I 即使不是无穷大，通常也是十分大的数字；第二，也是更重要的一点就是，在某一特定的应用中如何来确定 $P(i)$的大小，在上面所叙述的方法中并未明确指出，而且，事实上要真正确定 $P(i)$的大小，也是一件十分困难的工作。因此一种更为实际的，基于运行的软件可靠度定义可以这样来描述：

设 n 表示在一特定应用中程序实际运行的次数，c_n 表示在这 n 次运行中正确运行的次数，则：

$\lim\limits_{n\to\infty}\frac{c_n}{n}$就表示一次运行正确的概率。于是有：

$$R(t)=\left[\lim_{n\to\infty}\frac{c_n}{n}\right]^r,$$

其中，r 是在时间区间$(0,t)$内，程序运行的总次数。

故障率，也有人称它为风险函数(hazard function)，也是一个直接源于硬件可靠性的术语。

一般地说，我们用 $\lambda(t)$来表示风险函数，$\lambda(t)$就是程序正确地运行到时刻 t 时，单位时间内程序发生故障的概率(实际上，$\lambda(t)$应该是概率密度，真正的概率应该是 $\lambda(t)\cdot\Delta t$)。

如果以 T 表示从 0 开始运行一程序，到程序发生故障为止所经过的时间。则对于不同的运行，T 的值显然是不同的。因而可以断定，T 是一个连续型的随机变量。于是有：

$$\lambda(t)\Delta t=P_r\{t<T\leqslant t+\Delta t\,|\,T>t\},$$

因此，我们又可以写：

$$\begin{aligned}R(t+\Delta t)&=P_r\{T>(t+\Delta t)\}\\&=P_r\{(T>t)\wedge\text{在区间}[t,t+\Delta t]\text{内程序不发生故障}\}\\&=R(t)[1-\lambda(t)\Delta t]\end{aligned}\tag{1.1.1}$$

对等式(1.1.1)关于 t 求微分，我们得到：

$$\mathrm{d}R=-\lambda(t)R(t)\mathrm{d}t.$$

从而可以导出下面的微分方程：

$$\frac{\mathrm{d}R}{R}=-\lambda(t)\mathrm{d}t\tag{1.1.2}$$

解之，得：

$$R(t)=\exp\left[-\int_0^t\lambda(x)\mathrm{d}x\right]\tag{1.1.3}$$

上式描述了风险函数 $\lambda(t)$与可靠度函数 $R(t)$的关系。

风险函数 $\lambda(t)$在软件可靠性的研究中，受到了广泛的关注。因为从式(1.1.3)我们可以看出，一旦知道了 $\lambda(t)$，则 $R(t)$即可计算出来。但是在实际的工作中，却十分令人失望，直到目前为止，人们还只能对它作出形形色色的假设。

有的研究者认为，存在于软件中的错误，它们引起软件在运行过程中发生故障的可能性都是相同的，因此有理由认为 $\lambda(t)$是一个常数。但是，在我们的实践中，经常可以观察到这样笼统地假设 $\lambda(t)$为一个常数的作法与事实不符。不过可以认为 $\lambda(t)$在分段的时间内是一个常数。

联系到排错的实际过程来看，软件中的错误随着排错的进展，不断地有错误相继地被发现、被纠正、被排除，因此可以乐观地认为，存在于软件中的错误是越来越少了。与之相适应的一个事实就是：发生故障的频率也会随之下降。因此，有理由说：$\lambda(t)$本质上是一个减函数。

然而，无论是常数论者也好，还是减函数论者也好，他们的共同点在于，不会认为 $\lambda(t)$是一个增函数，并且也没有人能够具体地给出一个实实在在的风险函数的表达式。这并不是说我们的数学家们以及他们所掌握的数学工具的无能，而是因为软件在运行的过程中发生故障的诱因多种多样，影响到风险函数的因素过于复杂的缘故。

正是由于这样一个原因，才决定了目前在软件可靠性评估的研究工作中，一个普遍的作法就是：每一个估测模型的作者在提出模型之前，都不得不首先提出一系列的假设，然后在这些假设的基础上来开展工作。当然，他们作出的假设必须具有一定的事实作为基础。然而有的假设在实际的测试、排错过程中与事实有些出入的现象也时有发生，但为了在数学上处理起来容易，暂时认为它们还是成立的。这似乎已成为一种默契。

§1.2 软件可靠性模型的作用及意义

软件可靠性模型是随机过程的一种表示，通过这一表示，可以将软件可靠性或与软件可靠性直接有关的量，如：平均无故障时间或故障率等，表示成时间以及软件产品的特性，或者开发过程的函数。软件可靠性模型通常描述了软件可靠性对上述各变量的一种依赖关系。其描述的形式则根据通常由已知的故障数据出发所作的统计推断过程而定。对于软件的模块、子系统、系统都可以应用软件可靠性模型。

为了给软件可靠性的估测建立数学模型，应首先考虑影响到估测的基本要素。它们是：错误引入软件的方式、排错的实际过程以及程序运行的环境。错误的引入主要依赖于已开发出的程序代码的特性(程序代码开发出来主要是为了实际的应用，以及对程序的修改过程)，以及开发过程的特性。最明显的代码特性就是程序代码的长度。开发过程特性包括：软件工程技术，使用的工具，以及开发者个人的业务经历。有一点必须注意的是：程序既可以开发出来用以增加软件的功能，也可以开发出来用于排错。排错依赖于用于排错的时间、排错时的运行环境、用于排错的输入数据、以及修复行为的质量。环境则直接依赖于操作剖面。操作剖面(Operational Profile)主要指各种类型的运行出现的概率，而它们则由各自的输入状态描述其各种运行的性质。因为上述各基本要素大多具有随机性且均与时间有关，所以软件可靠性估测模型大多都处理成随机过程的形式。模型与模型之间，大多根据故障时间或已发生的故障次数的概率分布来加以区分，也可以根据与时间有关的随机过程的不同处理方式来加以区分。

一个好的软件可靠性模型还应描述出上述诸基本要素间的总的故障过程的依赖关系。根据定义，我们已假定模型以时间为基础(这并不等于说不以时间为基础的模型就不能用)。用不同的数学公式描述故障过程的可能性是几乎没有什么限制的。对于各种不同的数学表达式，我们可以通过建立模型参数的有效数据来加以判定。这样就有下面两种方式可以采用：

(1) 估计模型参数——将统计推断过程应用于程序运行过程中产生的故障数据；

(2) 预测程序将来的故障行为——由软件产品的特性和它的开发过程来进行判断(这可以在程序的任何执行之前进行)。

在对数学表达式进行判定时，总存在着某种不确定性。可以考虑采用置信区间的概念

来予以解决。置信区间就是指对于给定的一个确定的置信度，待估计的参数值确切地可以落在某个范围之内。

一旦确立了数学表达式，对许多不同的故障特性都能加以判定。对于各种不同的模型，它们分别采用了许多不同的表达式：

(1) 在任一时刻所发生的故障的平均数；
(2) 在一时间区间内发生的故障的平均数；
(3) 在任一时刻的故障密度；
(4) 故障间隔的概率分布。

一个好的软件可靠性模型应有一系列重要的性质：

(1) 对于将来的故障行为能给出好的预测；
(2) 对有用的量能进行计算；
(3) 简单明了；
(4) 具有广泛的应用；
(5) 它应在合理的、与实际情况完全吻合的或十分接近的假设基础上作出。

在对将来的故障行为进行预测时，应保证模型参数的值不发生变化。如果在进行预测时发现引入了新的错误，或修复行为使新的故障不断发生，就应停止预测，并等至足够多的故障出现以后，再重新进行模型参数的估计。否则，这样的变化会因为增加问题的复杂程度而使模型的实用性降低。

一般说来，软件可靠性模型是以在固定不变的运行环境中运行的不变的程序作为估测实体的。这也就是说，程序的代码和操作剖面都不发生变化。但它们往往总要发生变化的，于是在这种情况之下，就应采取分段处理的方式来进行工作。因此，模型主要地就要集中注意力于排错。但是，也有的模型具有能处理缓慢地引进错误的情况的能力。

对于一个已发行并正在运行的程序，应暂缓安装新的功能和对下一次发行的版本的修复。如果能保持一个不变的操作剖面，则程序的故障密度将显示为一个常数。

一般说来，一个好的软件可靠性模型增加了关于开发项目的通讯，并对了解软件开发过程提供了一个共同的工作基础。它也增加了管理的透明度和其它令人感兴趣的东西。即使在特殊的情况之下，通过模型作出的预测并不是很精确的话，上面的这些优点也仍然是明显而有价值的。

实际建立一个有用的软件可靠性模型，一般包括：坚实的理论研究工作、有关工具的建造、实际工作经验的积累。通常这些工作要求许多人一年的工作量。相反，要应用一个好的软件可靠性模型，则要求极少的项目资源就可以在实际工作中产生好的效益。

§1.3 软件可靠性模型发展简述

最早的模型出现在1956年，由H. K. Weiss提出了一系列的公式。但由于它们太复杂了，这些公式对以后软件可靠性模型的建立几乎没有产生什么影响。

1967年，Hudson观察到软件的开发过程是一个生灭过程，一个典型的马尔可夫过程。其中错误的产生是诞生期，错误的改正是死亡期，在任一时刻存在于软件中的错误个数可用来定义过程的状态，状态的转移概率则与生灭函数有关。为了在数学上容易处理，他的工作只限于研究纯死亡过程方面。他假设：错误改正率与剩余错误个数成正比，还与时间的某个正次幂成正比，即，错误改正率随时间的增加而增加。由此，他得出故障间隔(Interval Between Failures)的Weibull分布。同时他还发表了由系统测试阶段收集的数据，并声称：如果将系统测试分为三个既互相衔接又适当分离的子阶段，就可获得模型与数据之间合理的一致性。

对于软件可靠性模型发展首次起到较重要作用的两个模型，发表于1971年。Shooman模型由M. L. Shooman发表，J-M模型由Z. Jelinski和P. B. Moranda发表。它们有着惊人的相似之处，且Shooman在他稍晚的一篇文章中指出，存在着一个简单的参数转换集可以将J-M模型转换成Shooman模型。它们都假设：

- 软件中的初始错误数为N($N\geqslant 0$)。
- 故障率与软件中的剩余错误个数成正比。
- 一个错误一旦被发现，立即排除且排错不引入新的错误。

这些假设都被随后许多软件可靠性模型以各种方式所采用。

另外，J-M模型还假设故障的风险率为分段常数，它在每次改错过程中由一个常量来改变，但在两次改错过程之间保持常数。Jelinski与Moranda应用最大似然估计来估计软件中的总错误个数、剩余错误数与风险率之间的比例常数。

1972年，B. Littlewood和J. L. Verrall发表了第一个贝叶斯模型，他假设：故障间隔时间服从含参数λ_i的指数分布，且λ'服从先验的Γ-分布。据此由标准的贝叶斯过程可以获得$t_{n+1}|\{t_1,\cdots,t_n\}$。同年，G. J. Schick与R. W. Wolverton提出的模型与其它模型的主要区别就在于：他们假定连续的故障之间的排错时间服从Rayleigh分布。1973，W. L. Wagoner发表的模型与此相类似，但假设了风险函数服从Weibull分布。它的特例就是Schick-Wolverton模型。

J. D. Musa在1975年发表了执行时间模型，引入了一系列新的显式参数：

- 测试压缩因子，以表述这样一种思想：使用测试条件和数据(Test Cases)，比使用用户的输入数据有着更高引起故障发生的概率。
- 关于软件系统的初始MTTF。
- 与日历时间相对的执行时间(即CPU时间)。

同年，P. B. Moranda发表了几何泊松模型。他认为，故障率随时间的增加以几何级数

下降，并且下降的过程出现在每次故障的纠正期间。因此，他假设在第 i 个时间段内的错误数满足含参数 λK^{i-1} 的泊松分布。

同年，K. Trividi 和 M. L. Shooman 还发表了第一个马尔可夫方式的模型。它将系统区分为“Up”状态与“Down”状态：工作正常与需要修复。所有的 Up 状态与 Down 状态间的转换概率都被假定为相同。模型的输出结果是一个概率的集合，其中每个的值都是：

$$P(\text{在}[0,t]\text{内找出}\ k\ \text{个错误})。$$

还是在这一年，N. F. Schneidewind 发表了第一个非齐次泊松过程(NHPP)模型。他建议对不同的可靠性函数，如：指数函数、正态函数、Γ-函数、Weibull 函数等进行研究，并针对开发的具体软件项目，选用最适合于实际问题要求的可靠性函数，以估测软件系统的可靠度。他建议在可靠性估测过程中，可从实际的数据出发，用经常校正时标的方式来判定查明故障到纠正故障之间的时延。他使用离散的时间步以构造模型。

1976 年，M. L. Shooman 和 S. Natarajan 首次发表了考虑到排错时引入新错的模型。他们使用查错率、改错率以及引入新错率来处理新错的引入问题。

1979 年，A. Goel 和 K. Okumoto 关于连续时间的 NHPP 模型，对软件界产生了持续的影响。

B. Littlewood 采用贝叶斯方法以研究软件可靠性建模。他认为，在具有常数风险率和程序操作期间，故障的发生都是随机的，但却将风险率当作一个已经发生的故障的随机过程。因此，在 Littlewood 的模型中，风险率是一个有条件的概念。1980 年，在他发表的微分模型中，假设对于程序的风险率取不同的基值，因此不同的错误将以不同的频率出现。

1983 年，由 Yamada，Ohba 和 Osaki 发表的一个 NHPP 模型，给出呈 S-形的可靠性增长曲线的均值函数。由 K. Okumoto 和 J. D. Musa 提出的对数泊松过程模型，具有一个初始错误率 λ，以及对应于 λ 的一个递减率。模型的日历时间部分则与执行时间模型的日历时间部分相同。

T. Bendell 将试探性数据分析法(EDA)引入软件可靠性评估，且使用时间序列分析法以及成比例的风险函数建立软件可靠性模型。

1989 年，Y. Tohma，K. Tokunaga，S. Nagase 和 Y. Murata 提出一个新的、以超几何分布为基础的模型，用于估计软件中的剩余错误个数。

Tsu-Feng Ho，Wah-Chun Chan 和 Chyan-Goei Chung 提出一个模块结构模型，通过每个模块的可靠性来估计软件系统的可靠性。另外他们还提出了结合信息论的方法，来建立软件可靠性模型的方法。

Y. Masuda 等人认为，在软件执行期间的任一时刻，只有 k 个模块(k 为常数，即一软件中含有的模块总数)中的一个在执行。他们据此开发出根据软件的模块结构判定最佳投放时间的模型。而 H. Ohtera 和 S. Yamada 则不仅考虑测试，而且也考虑到在运行期间的查错过程中，根据软件可靠性目标和平均无故障时间指标，来判定软件的最佳投放时间。N. D. Singpurwalla 以如何在不确定的情况下进行决策的思想为基础，提出两个判定在软件投放前应测试和排错多长时间，且使软件的实用性达到最大的实用性函数(utility functions)，而它们分别以成本指标和软件可靠性指标作为控制。他指出，根据软件可靠性的概率模型使用于软件故障数据的结果，关于单个状态测试的情况，该最优化问题可以用

非数值化的技术求解。

硬件可靠性分配技术已成熟，但软件可靠性的分配则缺乏成熟的技术。F. Zahedi 和 N. Ashrafi 采用了对确立的系统划分层次的方法，应用分析分层过程(AHP：Analytic Hierarchy Process)以推定需要的模型参数。她们的模型具有非线性规划的形式，在模块和程序级别上考虑软件可靠性的各种技术和经济上的约束条件的同时，使软件系统的实用性达到最大。求解这一非线性规划问题，则可确定在模块和程序级别上的软件可靠性的分配方案。

现代的电子设备系统，既有硬件，也有软件，如何评估它们的可靠性，问题难度更大。J. C. Laprie 和 K. Kanoun 在这一方面有杰出的工作。

G. Pucci 应用恢复块结构技术，对软件可靠性等软件质量进行估计。而且根据错误对软件系统所产生的可观察到的后果进行分类，使得将测试数据应用于模型参数的估计成为可能。T. Downs 针对现有大多数软件可靠性模型在测试阶段将软件处理成黑盒子的作法，考虑对测试过程直接建模。

K. W. Miller 等人，开发出以软件的黑盒子模型为基础的理论分析方法，解决：(1) 在随机测试过程中，当观察到的故障次数为零时，如何估计当前版本的故障概率；(2) 在使用分布(use distribution)与测试分布(test distribution)不匹配时，对估计的故障概率进行调整；(3) 在估计故障概率时，将随机测试的结果与其它的信息结合起来进行分析。

N. Karunanith，Y. Malaiya 和 D. Whitley 应用神经网络系统理论预测软件可靠性。他们利用前向神经网络(a feed-forward neural network)对三个数据集合进行软件可靠性估测，结果发现：(1) 神经网络方法在软件可靠性估测中显示出良好的一致性，这是一般现有软件可靠性模型所无法做到的。因此，神经网络方法对于提高估测精度有着极大的贡献；(2) 由神经网络方法估计出的软件中的错误个数，始终较使用现有其它软件可靠性模型所估计出的结果要少；(3) 神经网络方法的估测能力，可通过由进行分析的数据，与其它的模型进行比较获知。

第二章　软件开发与软件可靠性

本章将简要介绍软件产品的一般开发过程，对软件中的错误做一些粗略的分析并进而对它们分类，概要地列举一些常用的软件度量，最后讨论一下软件可靠性与硬件可靠性的区别。

§2.1　软件的开发过程

传统的软件寿命周期，是包含了软件开始开发以前和它进入实际使用以后所出现的各种活动的一个长远观点。概略地讲，软件生存周期主要包括：软件计划阶段、软件开发阶段和软件维护阶段。

一般地，软件总是一个更大的以计算机为基础的系统的一个部分，所以，系统分析和定义必须先于（或至少是同时开始）软件计划。必须把系统功能分配给软件。

软件计划阶段以软件计划开始，对软件的作用范围必须作出描述，预测开发这一软件所要求的必不可少的资源，并且要作出代价和进度的初步估算。

软件计划的下一步任务是软件要求分析和定义，它要达到的目标主要有：

- 通过揭露信息流程和结构以提供软件开发的基础；
- 通过标识接口细节，提出深入的功能说明书以描述软件，确定设计约束和定义软件有效性要求；
- 建立和保持与用户及要求者的通讯，以便达到上述两个目标。

软件要求分析可以分成四个方面的工作：问题识别、评价和综合、写出规格说明书、复审。软件要求分析是软件计划阶段的关键一步，在这一步要完成的任务是将用户和要求者关于软件的模糊概念变换成一个具体的、有条理的、严谨的、无二义性的规格说明，这个规格说明乃是以后软件开发的基础，随后的一切开发活动皆以此为依据而展开。因此要求系统分析员具有敏锐的观察和综合问题的能力、由个别到一般的逻辑思维能力、由具体到抽象的分析归纳能力，以及用准确的语言与用户和要求者通讯并将这一切写下来的能力。

软件开发阶段是软件寿命周期中的中心阶段，关于软件的规格说明书一经确定，这一阶段的工作就应立即开始。

软件开发阶段的任务主要分为：初步设计、详细设计、编码、测试四大部分。

在软件的设计阶段，人们开发出各种行之有效的技术和方法，如：自顶向下、逐步精化、结构化设计方法，等等。许多研究者也提出了一些把数据流或数据结构翻译成设计定义的方法。

初步设计建立软件的模块结构，详细设计则完成所有必要的过程细节，给出设计表示，为下一步的编码打下基础，使之能据此直接而简单地导出源代码。

详细设计阶段可以利用诸如：图形工具、表格工具、语言工具等不同类型的设计工具，以完成设计描述。

编码阶段是一个翻译过程，将详细设计翻译成程序设计语言，最终由计算机自动地（利用与选用的程序设计语言对应的编译程序）转换成机器可执行的指令。风格是源代码的一个重要特性，简明性和清晰性是关键，即使牺牲一些执行效率和存储空间也是值得的。

软件测试是软件质量保证的关键一步，它的重要性及其与可靠性的密切联系，怎么强调也不过分。开发软件系统涉及到一系列的生产活动，人们在每一步中犯错误的机会多得举不胜举，人们无法完美无缺地行动和彼此通讯，因此，如何保证软件的质量是一个十分紧迫而重大的任务。软件测试则包括了对规格说明书、设计表示以及编码的最终复审。由测试所提供的有关软件系统的故障数据，则构成了进行软件可靠性分析的基础。

软件维护被有的人形容为漂浮在海洋中的一座冰山，大量潜在的问题和代价被隐藏着。变化是引起一切问题的根本。计算机程序总是在变动的。要纠错，要增补新的功能，要优化原有的功能，这一切都要求变动，而变动本身又引起一些新的问题。软件维护一般分为：校正性维护，适应性维护，完善性维护，预防性维护四种方式。为了纠正在软件使用以后所暴露的错误，就需要进行校正性维护；当外部环境的改变促使对软件进行修改时，就应进行适应性维护；完善性维护主要指由于用户集团所要求的对软件拥有功能的增补；为改进软件将来的可维护性及可靠性，并为将来的功能的进一步增补提供一个基础，我们必须采取预防性维护措施。

软件的开发除了传统的寿命周期方法以外，人们又提出并研究了诸如：快速原型法、面向对象的程序设计方法，以及计算机程序的自动生成等开发方法。有的已取得重要进展，有的则代表了新的方向。

关于软件开发过程的详细叙述，读者可参阅 R. S. Pressman: Software Engineering: A Practitioner's Approach.

§2.2 软件中的错误及其分类

对可能出现在一软件系统中的错误进行分类，有利于软件可靠性分析工作的进行。出现于各类参考文献中的用词不统一，在语义上间或也会产生一些混乱。在此列举一些术语并加以解释，同时力图在本书中采取前后一致的用法。

- 错误(error)：在某系统中，人们原来所期望的以及系统实际具有的状态或行为之间的偏差。
- 故障(failure)：在软件运行期间出现的错误，它是因软件中存在的错误引起的、在执行期间的动态表现。
- 缺陷(fault)：泛指系统中的所有错误。
- 过失(mistake)：人类在设计、运行系统过程中所犯的错误。在软件中一切错误皆因此而产生。

我们如果要区分一个错误的影响和一个错误的起因之间的区别，我们就要讨论错误的征兆和错误的诱因。

仅仅当状态与预先定义的标准发生了偏差时，人们才能辨认出一个错误的存在，因此错误也只是一个相对性的概念，特别在缺少明确而严格的、统一的标准时，它的意义就会显得十分含糊不清。在对给定的状态进行评价期间，当应用那些事先制定的标准的时候，将这些状态本身进行正确的分类，或者错误地依据制定的标准而行事，都是可能的。

在开发软件系统的过程中，一个可供参考的标准可以来源于系统说明，在这里当然指的是功能要求标准和系统的认可、接收标准。如果不存在明晰而不含糊的系统说明，那么，在开发软件系统的接收过程中间，关于系统功能的实现，人们就可能以一个模糊不清的假设去取代一个客观的标准。然而，这样做的结果，在许多情况之下往往都会使人们对具体的问题产生严重的误解。

软件错误和硬件错误

首先，最普通的分类方式就是将错误划分成软件错误和硬件错误。软件错误包括由系统分析或者程序设计而产生的全部错误；硬件错误则是由于机器的失灵而产生的错误。虽然乍看起来这一区分是明确的、毫不含糊的，但是，许多实际存在的难以区分的模棱两可的情况，增加了将错误按此法分类的难度。随着技术的进步，硬软件之间的界线变得越来越不清楚了，人们在构造计算机系统时，越来越多地采用了“软件硬化”以及“硬件软化”的技术。如：中央处理机中只读存储器中的微程序，它的一个错误在许多情况下，都会引起一条硬件指令的错误执行。存在于微程序中的错误，一般都是在进行微程序设计时产生的，照理说，它应该划入软件错误的范畴，但是，它引起的都是一条硬件指令的错误执行，是否又该将它划归硬件错误的范畴呢？

根据错误的征兆来进行分类

根据错误直接显示出来的征兆对错误进行分类，是特别有意义的。因为这样一种分类方法可以直截了当地进行，而无须事先进行那些不得不进行的、而且既困难又使得问题更趋复杂化的错误诊断。当我们把对错误分类的问题与软件系统的输入空间联系起来考虑时，我们可以根据错误的征兆，按表 2.1 所示的方式对错误进行分类。在这种分类的方式中，某种类型的错误所引起的实际后果，与特殊的应用项目的要求是密切相关的，也就是说，实际后果紧密地依赖于该应用的具体要求。一般地说，第 I 种类型的错误(即因系统本

身辨认出不能够处理其正常的、但超出预先规定的输入范围的输入，因而对该输入加以拒绝）是并不如其他各种类型（第Ⅱ—Ⅴ种）的错误那么严重的。但是，这一点也有一个例外的情况，我们可以在实时系统中找到。在一个预先确切规定的时间范围之内，系统必须对任何的输入做出反应。因为在实时系统中，系统对某个输入的拒绝反应将可能导致一个灾难性的后果。

表 2.1　根据查错并联系到输入范围的输入对错误进行分类

输入 / 系统反应	超出预先规定的输入范围（输入错误）	在预先规定的输入范围之内
对输入的拒绝（系统查错）	正确	第Ⅰ种类型的错误
得出错误结果（系统外查错）	第Ⅱ种类型的错误（严重）	第Ⅲ种类型的错误（严重）
系统崩溃	第Ⅳ种类型的错误（严重）	第Ⅴ种类型的错误（严重）

根据错误的起因对错误进行分类

在计算机系统中的每一个错误，究其产生的原因，大都可以归纳为下列几种情况之一：在硬件或软件中的错误设计；由于环境或部件的老化所引起的硬件恶化；错误的输入数据（包括操作人员的失误或者实际的输入数据是错误的）。设计错误、硬件的恶化和数据错误是构成错误空间的三个互相垂直的轴，如图 2.1 所示。

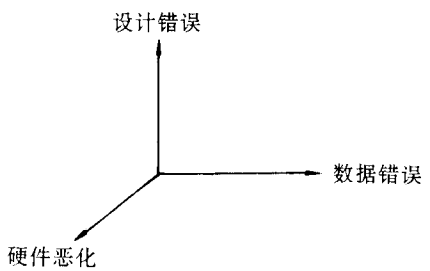

图 2.1　计算机系统的错误空间

将错误划分成这三个部分的错误分类法，是以错误的起因作为基点的，而并非以错误所产生的影响作为基点。因此，这一分类方法必须以一个完全的、成功的错误诊断作为它的先决条件。

在一个设计错误出现的时候，任凭所有的硬部件的操作都是正确的（这只要参照它们的说明，是很容易判别出来的），任凭输入数据都是正确的，但是系统总不可能产生出人们所企望 的正确计算结果来。

在一个系统部件由于恶化而出现错误时（主要由于元部件的老化或者环境因素的影响），系统所表现出来的状态，在说明中是不会有相应的叙述的。设计错误和元部件的老化，在硬件中也都会出现，但是，在软件中，仅仅会出现的只有设计错误这一项。

起源于系统的不正确操作而发生的一个输入错误，也即是说，由于在相关的操作手册中没有给出正确的操作命令，以至产生了输入错误；或者是实际的输入数据确实是错误的；这样的两种情况都会导致把错误的数据输入到系统中去。因此，正确设计出在发生输入数据错误时系统本身应该采取的应急处理措施，是系统设计的任务之一。在表 2.2 中概括地表示出这三种类型的错误的特性。

表 2.2 各种类型错误的发生情况

	硬件错误(恶化)	软件错误(设计)	输入错误
错误起因	部件的老化故障 环境	设计的复杂性 输入分布	人类的过失
作为时间的函数的错误率的变化情况	初期阶段减少,然后为一常数,在生命周期的后期又增加。	常数(假定程序和输入分布都不变)	初期阶段减少(一个学习曲线)然后是一常数。
这一类型的错误能否完全地从系统中排除掉(理论上/实际)	不可/不可	可[①]/不可	不可/不可

根据这一对错误进行分类的方法,存在于数据库中的一个错误的数据元,它或者是一个软件错误,或者它将引起一个错误的输入。如果我们采取这样一个观点:数据库是软件系统的一个部分,则存在于数据库中的一个错误的数据元,也是一个软件错误。但是,如果数据库被认为是独立于软件系统之外的,则存在于该数据库中的一个错误的数据元,将导致一个输入错误的发生。

根据错误发生的持续时间来分类

对于硬件错误和输入错误,我们可以根据它们发生时的持续时间来进一步进行分类。如果一个错误从某一特定的时间开始,往后它总不被打断,则可以将它称为永久的错误。可是,当某个错误发生时,它只使得系统特性产生一个十分短暂的变化,然后系统又可能完全恢复正常状态,这样的错误可以被称为短暂的错误或瞬时错误。此时虽然错误的持续时间是短暂的,但它产生的后果却可以是十分严重的,特别是对于程序往后的进一步执行要受到严重的干扰。由于短暂的错误不可能用系统的方法予以重现,因而使得对它们的诊断和排除成为十分困难的工作。

在硬件中发生的短暂错误,往往很难与软件错误所造成的影响区分开来。例如:一个硬件的缺陷,它产生的影响是将某位二进制数位取反,则这一影响可能导致一个错误数据的存储,或者一条错误指令的执行,由此而引起的征兆就非常难得与软件错误区分开来。

内部错误与外部错误

一个错误的直接后果,有时常常是不能够被直接观察到的,除非只有在诸如一个"内部错误"的影响传播到一个影响到输出的点上时,该错误才能够被从系统的外部观察出来。内部错误与外部错误之间的区分,就是以可观察性作为依据的。如果一个系统采用了

① 关于在软件中的设计错误,在理论上能否从系统中完全排除掉的问题,至今仍有两种截然不同的意见,一种认为可以——如图上所示;另一种则认为人类的过失在设计过程中无法完全避免,所以是不能够完全从系统中排除掉的。

冗余技术，并非每一个内部错误都将必定会导致一个外部错误。内部错误和外部错误的细分，在极大的程度上要依赖于在每一个特殊情况中的、被选择用来查错的查错界面。例如，如果一个查错界面具有分层次的结构，则同一个错误既可以被看作一个内部错误，也可以被看作一个外部错误，它的区分则依据于作为划分标准的层次的级别。而且，深入研究内部错误与外部错误之间的区别，是进一步分析可靠性的基础之一。为了给从系统中排错打下一个基础，就要求对在某些特定的查错界面上查出所有内部错误的起因的能力进行量测；或者，要求在采用冗余部件以后，对于随后的外部错误的预防能力进行量测。如果一个错误在系统内部查不出来，也就是说，查错技巧只能在系统以外来实施的话，那么，该错误就可以引起一个严重的系统错误，也可能就是引起系统崩溃的根源。

根据应用的结果对错误进行分类

关于复杂系统的说明，除了包含有全部的、以经济的观点来解释的系统执行的基本功能以外，往往还要包含对那些有助于增强整个系统的有效性的次要一些的功能的解释。假如这些次要功能失效的话，对于用户来说，结果也许并不会是灾难性的。但是，如果一个错误导致系统基本功能执行的中断，那么这一错误就是关键性的。如果一个错误仅仅只影响到系统的那些次要功能，那么，这个错误就是非关键性的。只有关于系统的说明中能对全部的基本功能和次要功能进行清晰的划分并加以区分的时候，将错误划分成关键性的和非关键性的这一分类方法才是行之有效的。对于可靠的系统的开发来说，这样的区分是十分重要的。而且十分明显，对于关键性错误，设计者们必须要给予足够的重视。在确定性的情况之下，为了纠正一个已经出现的关键性错误而故意中断一个次要功能的执行，即使不是十分必要的，那也是完全可行的。Fragola 和 Spahn 于 1973 年甚至提出比仅仅区分关键性错误和非关键性错误更强的区分法，以便对一个错误的后果亦可进行分类。在他们提出的方法当中，对每一个错误都标以一个数字，以便指示出它们的严重性的程度。

根据开发阶段的分类法

一个软件系统的开发与使用可以分成许多阶段，而其中的每一个阶段又是下面一个阶段的先导。因此，出现于开发过程中的错误，也可以根据开发阶段来予以分类(如表 2.3 所示)。对于一个具体的问题都能得出一个具体的解，换句话说，当在软件开发周期中的每一阶段都是正确的时候，也只有在这时，才可以认为一个给定的结果是正确无误的。因此，对于开发过程中的每一阶段，我们都必须给以足够的重视。只注意于开发的某一个环节而忽视其它的环节，这样做的结果只能对整个系统的可靠性起到相反的作用。

系统分析中的错误

我们周围的大千世界，包罗了万事万物，有许多的问题也并不总是以数学公式的形式出现的。因此，通过一个长期的、反复的抽象过程，以便从实际的问题中提取出对该问题的描述，写出对该问题的抽象化表示，往往是十分必要的。在技术上来讲，这一点对于上面所叙述的开发阶段，也往往是很困难的一个环节。它要求从大量的信息中抽取出与具体问题有关的每一个要素，去掉那些并不涉及到问题本质的次要因素，去掉许多根本与问题无关