软件高端人才修炼系列

企业级应用软件架构开发过程与实践

第一章

版本　**0.8**

胡协刚

首席软件架构师

szjinco@public.szptt.net.cn

www.chinaarchitect.org

目录

第一章　软件与软件的特性——从业务上下文出发的软件图景

软件是人类有史以来创造的一种非常特别的制品，它具备与传统制品完全不同的特性。

由于软件最初的形态只是用于科学计算的简单程序，这使得人们（特别是刚刚学习编程的初学者）往往倾向于将软件看作是一种相对独立的事物。传统软件工程也是从软件需求开始来阐述软件的生命周期，并给人一种错觉，即软件需求早就存在于用户那里，我们要做的只是去发现它们（需求获取）。这使得人们常常忽略一个简单的事实——软件是为了支持客户的业务（或解决领域问题）而被开发的（需求规格是被开发出来的，换句话说是被设计出来的），其内在特性受其上下文的制约，甚至就是上下文的一种直接反映。

我们在研究软件的时候，不能脱离于它的上下文，例如软件的复杂性根本上源至其问题域的复杂性。只有从业务（问题域）上下文入手，我们才能真正看到软件的真实图景。

本章试图从一个更为广泛的视角，来分析软件、还有软件的特性；而软件及其架构的开发策略与方法的研究，则始于人们对软件自身特性所引发的固有问题的一种解决努力。

第一节　软件的缘起与信息技术对传统行业的改造

我们要构造软件，所以必须先要了解软件；而直接了解软件本身还不够，还应当从其上下文入手，理解它的缘起以及相关的来龙去脉。

人们对软件往往充满了误解，倾向于将软件看作是一种特殊而相对独立的事物，这阻碍了人们更有效地开发和应用软件。软件技术并非独立的技术，它属于信息技术的一部分；它与硬件技术一道用来解决人类所面临的各种与信息有关的问题。

信息技术的产生源自计算机技术的革命性发展；而信息技术的大规模运用，则要依赖于软件技术的成熟。

我们在这里回顾一下软件的缘起，或许能让我们澄清以往的一些误区与模糊认识。

软件的缘起

计算机刚刚出现之时，其主要应用场景是替代人来承担那些复杂的数值计算；人们在计算机上编制一些最终通过转化能让计算机识别的代码，来控制科学计算流程的执行，这便是编程 programing 的由来。数值计算本身其实也是一种信息的处理过程，但是非常初级和范围狭窄。这也使得人们倾向于忽略所关联的科学计算问题，而将程序当作一种相对独立的事物来看待。

人类世界存在着海量的信息急待处理，人们使用手工方式来处理，其速度和工效已经远远满足不了要求。例如，随着银行业务的急剧膨胀，其资金账户的数量增长到千万甚至是亿万；银行柜员为了给储户取款，需要找到对应的账户记录，并核对其余额是否足够；虽然单个营业点的账户数量可能较少，在数万之间，但查找的难度仍然很大；而目前银行业已经转向了大集中的业务模式，所有账户都存放于一个数据中心，手工方式显然已经不可能应付这种处理量巨大的场景。

人们于是自然地联想到计算机，它既然能够对简单的数值数据进行处理，那么它能否处理更为复杂的数据，乃至是富有领域意义的信息本身呢？随着计算机处理指令越来越强大（具备对数据进行排序、对比等复杂的处理），存储器的成本不断缩减（较低成本地保存海量信息），I/O 设备的功能越来越丰富（方便人们使用计算机，和将现实世界的信息导入计算机），人们终于成功地实现了利用计算机来替代人处理信息的目标。另外，网络技术的完善，使得信息处理的模式首次打破了地域的界限。上文所述的银行大集中业务模式，便得益于网络技术的发展。

为了处理不同业务所涉及的各类信息，人们需要改变原来面向科学计算的简单编程模式。编程的关注焦点从控制计算的执行顺序，转向如何处理各类代表了领域意义的数据（例如，银行的资金账户数据）。最后，编程的实质已演变成为如何利用各类计算机资源来解决各种领域问题。此时，仍然使用程序 program 来称呼这些复杂度与规模提高了若干数量级的代码，显然已经不能真实地表达其背后的含义；于是，软件 software 这个概念被引入，它与硬件 hardware 一起组合成为实现某种应用的完整系统

system。

上述编程模式的转换，其意义远比想象的更为重大，它带来了一种全新的思路——应用信息技术来解决传统行业所面临的各类问题。

从程序员到软件工程师

笔者于 80 年代末进入大学，开始学习计算机和程序设计相关（甚至也包括数据库）的课程；这些课程的学习显然没有让我真正理解软件是怎么回事；我学懂了编程语言，但没有学会如何为实现某种领域应用而编程。记得我们在学习数据库技术时，花费了不少精力来开发一个模拟程序，将 DBase III 的插入、删除记录等指令都调用一遍；但当时就是没有人想到，用 DBase III 来开发一个诸如管理学生每个学期选修课程报名情况的简单应用。没有开发实际应用软件的经验，使得我们很难从简单编程的思维层面，上升到软件开发的思维层面。

本书主要面向那些期望成为软件架构师的人，而要成为一名称职的架构师，首先必须成为一名合格的软件工程师。软件工程师要面对的是软件，而不是纯粹的程序代码；软件所涵盖的范围比程序本身大一到两个数量级，涉及到大量的我们在学校不曾学习或被忽略的其它方面知识。例如，软件开发的最上游活动，并非需求开发，而是用来理解软件上下文的业务（领域）建模。

传统的业务执行过程

考察现实世界中的主要业务组织，从其静态结构看，我们可以抽象出：人、工作场所与工具、被加工的物质，以及相关的信息（这些信息大部分附着于前述元素之上）等元素；而从其动态行为看，又可以抽象出：物质流与信息流（财务上还存在资金流的概念，其客观上表现为物资流的形态，但实质上以信息流的方式发生作用）；为了驱动业务组织的运作，每个组织都有相应的业务目标，它贯穿于组织的行为和结构之间，是组织生存的理由之所在。

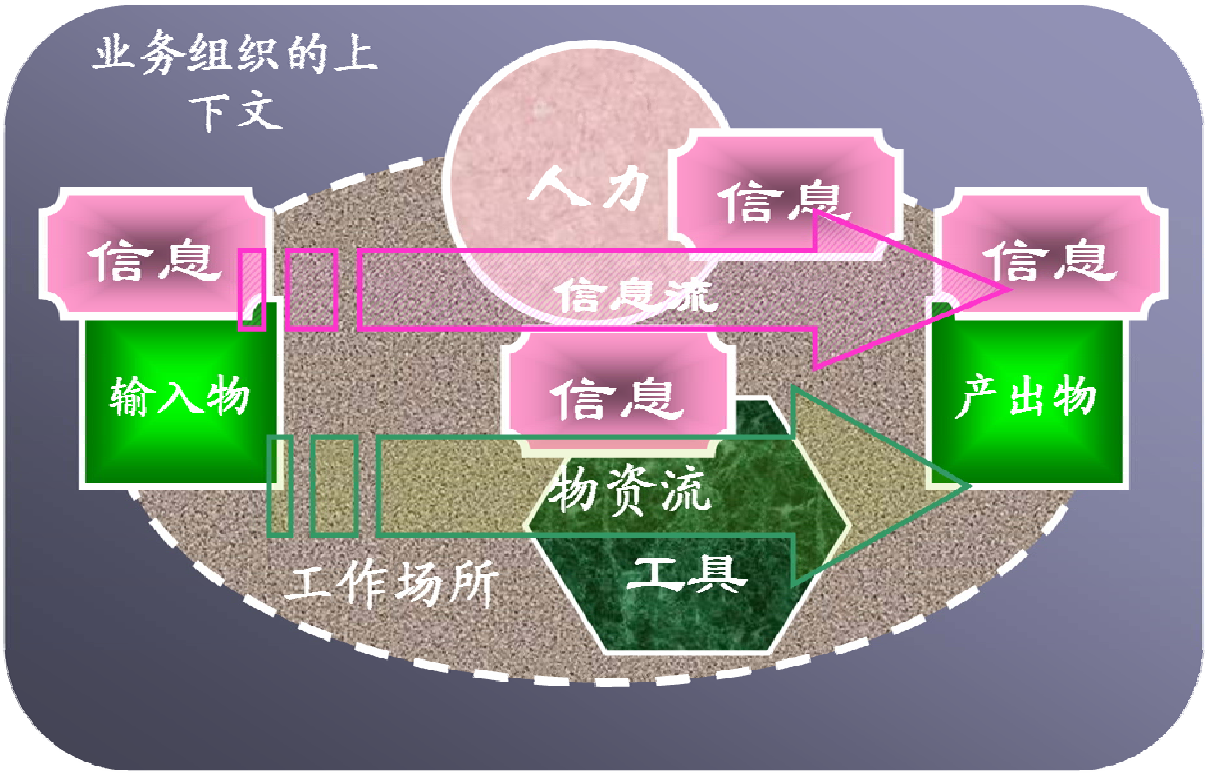


图 1.1 业务组织结构与活动示意

我们来观察传统的银行储蓄所为某个储户取款的业务过程。如上图所示，左边的输入物是储户递交给银行柜员的取款单和存折，取款单上承载了储户的基本资料、取款金额等信息；银行柜员接下来依据其掌握的工具操作知识（技能信息），操作带索引的账簿存储柜这类工具（存储柜的面板上有账簿的编号范围信息），找到储户的资金账户记录（上面记录了储户的资金余额，历史交易等信息）；然后依据业务规则（被柜员记忆在大脑中的规则信息）检查账户上余额是否足够等等...；最后，银行柜员将清点正确的现金，连同存折（此时其上的余额信息已经被修改）等产出物交回给储户。

从上述案例的分析，我们可以看到，信息、以及信息的流动，实际上是业务活动中必不可少的内容；而具体到储户取款过程，信息的处理实际上是其最核心的内容。计算机出现之前，还没有能帮助人们处理信息的强有力工具，这个时期的信息处理基本上由人来承担。人类对普通信息（创新型信息除外）的处理，其准确性和速度工效都不高，这严重地阻碍了传统行业生产率的提高。于是信息技术被发展起来，人们使用信息系统来替代人完成那些繁琐而容易出错的信息处理工作。

信息技术对传统行业的改造

信息技术具备传统行业所欠缺的诸多优势。目前利用信息技术来改造传统行业的模式主要有：

- 利用信息技术高效与灵活的特点，来改进原有手工的业务活动，从而提高整体的工作效率，但是并不改变业务本身的内容和主要流程。例如，原有的文案工作主要靠人们使用纸张和笔来完成，而现在一般都会使用文字处理软件来编制电子文档。
- 引入信息技术中的先进功能，以重新组织和排列业务流程中的各项活动，从而革命性地提高业务的生产率；这种模式改变了已有的工作方式，甚至创造出全新的业务与模式（业务流程再造与创新）。例如，银行利用互联网技术，将传统的转账、查账等业务功能，直接延伸到客户安装有浏览器的桌面电脑，让客户在家中就能够方便地操作相关业务。
- 利用信息系统的自动化执行能力，对业务（或工厂的设备）的运作进行实时的监督和控制；人类的生理与心理特性，决定了人类不能 24 小时不间断工作，也不适应长时间从事枯燥与乏味的工作；信息系统正好能够弥补人类的这些缺陷。例如，证券交易公司需要实时监督股民的股票买卖交易，一旦发现有人违规操作或有诈骗嫌疑，公司必须及时进行干涉，以避免资金的损失；目前的证券交易软件通常都会提供上述的实时监督功能。

企业级核心应用软件的特性

支撑业务的企业级核心应用软件通常具有如下特性：

对需要永久化的各类业务信息进行处理（引申为永久化的各类数据）。

牵涉的数据量极为庞大，需要专门的机制加以存储、访问与管理。

有大量的用户同时存取这些数据。

需要与其他的企业应用软件集成协作以完成附加值更高的工作。

不同的应用软件、业务逻辑对同一数据的含义，其理解可能完全相反。

运作的环境往往在地理上呈分布状，同时硬件、操作系统为异构环境。

企业的业务逻辑非常复杂而且多变，特别是有些业务规则往往在逻辑上没有明显的意义，例如可能有的销售部门要求财务周期比公司的时间提前两天，仅仅为了与客户的财务周期保持同步。

软件与硬件的分工协作

信息系统包含硬件和软件两个范畴。硬件包括：计算机、计算机网络、以及各类外围设备等；而在硬件上执行的代码，则属于软件范畴。

现实的业务对信息系统提出了相应的信息处理需求，这些需求最终要落实到系统的软件或硬件之上。那么如何向软件与硬件分配那些信息处理的需求呢？

业务过程本身其实也包含功能与性能两个不同的范畴。一方面，为了支持业务的

功能行为，信息系统必须能够处理各种不同类型的信息，而且其处理逻辑可能也是多样化的；另一方面，为了满足业务的执行效率等要求，信息系统在处理信息时，其速度、吞吐量等必须达到某种指标。这便是所谓的功能需求和非功能需求概念。

我们将偏向于利用软件来承担复杂而易变的（信息的）逻辑处理，从而实现种类繁多的行为功能（各类具体的信息处理步骤）；而利用硬件来提供必要的基本运行能力，并尽可能从硬件来获取性能（例如使用巨型机，以便支持软件从亿万数量级的资金账户中很快找到储户的余额数据）、可靠性（例如主机的热备份技术）、分布处理等非功能特性。

在上面的文案工作例子中，鼠标、键盘（或者还包括手写输入板）等硬件提供了文秘工作者输入文字和控制编辑的物理手段，显示器则让她能看到工作的结果；然而，具体输入什么字母或汉字，当前的编辑是复制还是删除，显示器上显示哪些内容，这些逻辑功能则是由文字处理软件和操作系统软件来承担的。另外，如果文秘工作者处理的文档比较大，主机的处理器主频和内存的配置则必须足够高，否则其工作速度将慢得难以忍受，甚至让人有重新拿起纸和笔的冲动。

性能等非功能属性不可能独立存在，它从属于功能属性。因为性能等非功能属性经常是类似的，这样不同的业务场景，其使用的硬件可能完全相同，而软件则不大可能一样。可以说，当前信息技术对传统行业的改造，其核心工作主要集中于软件上。

提升系统性能的着眼点

软件具备功能扩展的灵活性，在满足基本的性能要求下，实现系统功能需求相对开销较小，但要进一步提高软件执行效率则非常困难（除非开始的设计就存在重大性能漏洞，否则在原有基础上提高哪怕是 30％的性能都可能是奇迹）；而且常常会为了性能而破坏软件的结构与可理解性（例如，数据库存储过程的引入是为了提高性能，但破坏了纯面向对象的开发范式）；总之，很多情况下从软件方面来改进系统总体性能成本过高。

另一方面，硬件具备极大的性能可选择范围，从普通的 PC 服务器、到高端服务器、到小型机、甚至是巨型机，完全能够支持不同数量级的性能要求，而且硬件的升级符合摩尔定律，加之服务器集群技术的发展（例如，增加一倍的服务器数量，系统总体性能提高接近一倍是比较容易的），使得其成本下降非常迅速。

软件是对现实世界的一种映像

当程序刚刚出现，只是用于科学计算之时，程序与现实世界中的各行业领域基本上没有明显的关系。这个时期，计算机是名副其实的“计算机”computer；程序代码

描述的是如何对数值（数据）进行各类运算；程序与其解决的问题之间并没有明显的界线。而当计算机被当作信息处理设备，来解决各行业领域问题时，情况则发生了革命性的变化；行业领域中的信息，不是简单的数值，它们承载了领域的内涵；而这些信息只有被表示为机器能够识别的数据格式，才可能为计算机系统所处理；于是，计算机域与业务领域之间出现了明显的界线。

我们必须先将现实世界中的信息进行抽象，映射为计算机及其软件所能识别的数据，然后才能设计软件的行为来处理它们。也就是说，软件中的数据是现实世界中信息的一种映像，而软件运行时刻所表现的行为则是现实世界中信息流的一种映像。

为了表达现实世界的信息及其变化，人们开发了不同的计算机语言来描述它们，并将它们映射到计算机域。从下图可以看到，计算机语言的发展趋势，便是帮助人们更加方便地表达问题域（信息）。

在机器码或者汇编语言的时代，人们的主要精力耗费在如何控制 CPU 本身的执行细节上。面向过程语言的出现，将程序员从繁琐的机器底层细节中解放出来，使得他们还有余力去思考问题域中的相关内容。而面向对象语言的产生，则使得程序员第一次能够直接从问题域角度来思考编程的内容（如果说面向过程语言中的数据结构，多少还能让人联想到 0/1 比特或字节的概念，那么，“对象”几乎已经看不到其背后计算机的踪影了）。到了 MDD 模型驱动开发的时代，开发人员关注的焦点，已经完全从机器转向了问题域。建模语言（例如 UML 语言）帮助分析人员去抽象现实世界的业务内容，并形成反映现实世界的模型；这个模型随后被推演，从而得到对应的解决方案；最后，解决方案模型可以直接转化为中间语言，进而编译成为可执行的机器码（软件交付），并部署到机器上去运行。

图 1.2 计算机语言的发展示意

软件所面对的（业务）问题域非常广泛和复杂，存在大量形态各异、灵活多变的处理逻辑（功能）和丰富多彩的领域内容；而计算机本身只提供功能简单、形式单一的指令集，和最基本的数据表示方式；软件工程师需要充分发挥其聪明才智，通过创造性地组合那些功能简单的指令，来实现极其复杂的业务逻辑。传统的业务活动都是由人自己来执行的，因为人类思维的不精确性与随意性，造成业务执行过程含糊多变（国内单位因为管理不规范，这种人为的不确定性更为突出）、甚至充满了矛盾；然而，计算机只能接受严格、无二意的精确指令，任何存在二意的表示，将使得计算机进入不可预知的状态；开发团队必须发挥极高的主观能动性，去排除现实世界中的矛盾和不确定，将模糊转化为精确。用户或领域专家熟悉相关的业务，但是他们不了解计算机技术，他们无法提出具体和完整的系统需求来支持其业务；另一方面，让计算机来处理业务，只是将业务内容表示清楚还远远不够，大量繁琐、难懂的底层细节，等着软件工程师来一一实现；开发团队将面临如何去填补这其中巨大鸿沟的挑战。

计算机语言，特别是建模语言的发明，实际上是帮助人们来填补上述问题域与计算机域之间巨大鸿沟的主要工具。另外，人们设计了不同的开发方法（开发生命周期）来解决上述鸿沟，本书将在后续的软件工程相关章节继续深入探讨这些话题。

第二节　软件的特性及其意义

我们对软件的大体映像，建立于其根本特性之上；而软件的外部行为表现、和内部结构等，均受这些特性的影响和制约，并反反复复地印证着那些特性。

人们往往不愿意浪费时间去认真思考软件的本质特性，其后果常常是使用了错误的方法去开发软件。

软件是人类有史以来创造的一种非常特别的制品，它具备与传统制品完全不同的特性。而软件的这些特性既来自信息技术本身，也来自其所要解决的问题域。正是它的这种特殊性，使得人类至今仍然在苦苦寻觅解决软件开发问题的终极手段——所谓“一枪毙命的银弹”。

注：在西方传说中，有一种“人狼”的怪物，任何普通的刀剑、枪弹都杀不死它；只有使用一种银质的子弹，才能将其一枪毙命。在《人月神话》这本书中，作者用银弹来比喻解决软件开发问题的灵药；但作者认为，到目前为止，这种使得软件项目的成功成为大概率事件的技术还不存在。

软件中的精确性与模糊性

大部分人造物，比如象建筑，都可以允许一定的误差存在。我们在修建公路时，并不苛求最终的路面位置与设计图中的坐标值丝毫不差；在建造楼房时，也不要求施工队浇注的钢筋混泥土框架与设计的承重指标精确相符。一方面，这是因为人类还没有能力以合理的成本在建筑上实现极高的精度；另一方面，人类可以通过留有余量的手段来解决误差问题。例如建筑师在设计楼房时，会为承重指标留一定的余量，即使施工队浇注的框架结构强度比设计指标低一点，也不会因为这种不精确而导致其坍塌。（留余量、多付出一定的代价以规避风险的做法，实际上是传统工程的一种通行原则，称为 Overengineering 过度工程或过度设计；极端的例子是运载火箭上的控制设备至少备有三套，以防控制指令出错）

但是软件却不像上述其它人工制品，软件的最终交付形态是二进制的可执行码，执行码是不能容忍误差的。想象一下，我们复制一段可执行文件到主机上去运行，如果文件中的某个字节出现了错误，这时我们恐怕只能向上帝祷告，祈求系统不要突然挡机了。据说欧洲亚利安娜火箭一次发射失败的主要原因，就是其飞行控制软件中的一行代码有误所造成的。实际上，程序代码中的任何误差，都可能导致软件的整体失效；人们也无法通过简单地留余量的（过度工程）方式来解决这种问题。

其背后的原理是：建筑设计的客体内容（结构、材料、工艺等）本身并不是精确的东西（取值是连续的）；而软件开发的客体，是计算机的执行序列码，它本身是绝对精确的东西（取值是离散的，最终会转化为机器中代表 0/1 的电平信号），不允许误差的存在。（当然，如果未来哪一天，神经元计算体系取代目前的冯•诺依曼体系，局面可能就完全不同了）

软件的这种精确性，迫使人们在开发时，必须投入大量的精力来确保代码的正确性。而人类的思维特性正好与之相反，是模糊和充满误差的。因此，程序员很难在第一次就写出正确的代码；他总是要不断的测试这些代码，然后调试、跟踪，进行除错，直到获得正确的结果。

这样，软件的测试与验证在开发活动中就占据了极其重要的地位。另外，其它行业中很少会出现软件开发中的一种特有现象——程序员大部分时间中不是在编写新的代码，而是在排除已有代码中的各种 bug；更要命的是，程序员自己也没有把握，在其承诺的时间内，将主要的缺陷都排除掉。

软件的精确性还体现在软件的规格定义上，如果软件的需求规格没有定义准确，那么据此所开发出来的交付也不大可能符合用户的要求。

修复 bug 的痛苦之旅

由于人类的思维局限性，造成任何人造制品中都可能出现错误或瑕疵。软件对精确性的高要求，以及其本身的复杂与不可视性，使得软件开发过程中引入缺陷的概率比其它行业大了一个数量级；同时，其修复缺陷的难度也高得多。

笔者在以往编码生涯中，感觉最痛苦的事情，莫过于去排除程序中的 bug。修改 bug 本身其实并不难，真正的困难在于如何从成千上万行代码中找到它的病根。软件 bug 之所以被发现，是因为可以观察到其症状的表现；但是，引起症状的根源却隐藏在软件内部的任何一行或多行代码中，并不能直接看到它们。定位 bug 的根源，很多时候是需要极高的耐心与毅力的。

程序员在缺乏排除类似 bug 的经验时，只能先努力观测症状，再去猜想 bug 可能的根源，然后尝试修改对应的代码，之后运行之，以验证 bug 是否被消除。如果运气足够好话，bug 的症状可能就不再出现；这意味着，程序员也许终于不必再重复上述观测、猜想、尝试和验证的过程了。

然而，猜想最终被证实大多是要经历无数次反复的。另外，很有可能，程序员甚至永远也猜不透那个 bug 的谜底。

改善缺陷修复效率的主要途径，一是单元测试，这是因为每个单元的代码可以控制在几百行、甚至几十行之内，一旦在单元测试中观察到 bug 的症状，那么，这个 bug 的根源只是局限在这几百行代码中而已；另外就是增量式的开发，每次只是增加几百行代码，然后就进行全面的系统测试（这便是所谓的持续验证测试），一旦发现 bug，那么其根源隐藏在新增少量代码中的概率非常大（增量式开发加上持续验证，能够解决单元测试所不能解决的系统级 bug 定位问题）。

理论上已经证明，程序的百分之百正确性是不可能实现的。因此软件同样无可奈何地需要容忍一定误差的存在，但这种误差主要体现在软件的某些质量范围上，而软件的功能本身在逻辑上仍然是排斥误差的。

软件中的可以接受的误差或不精确主要体现在：

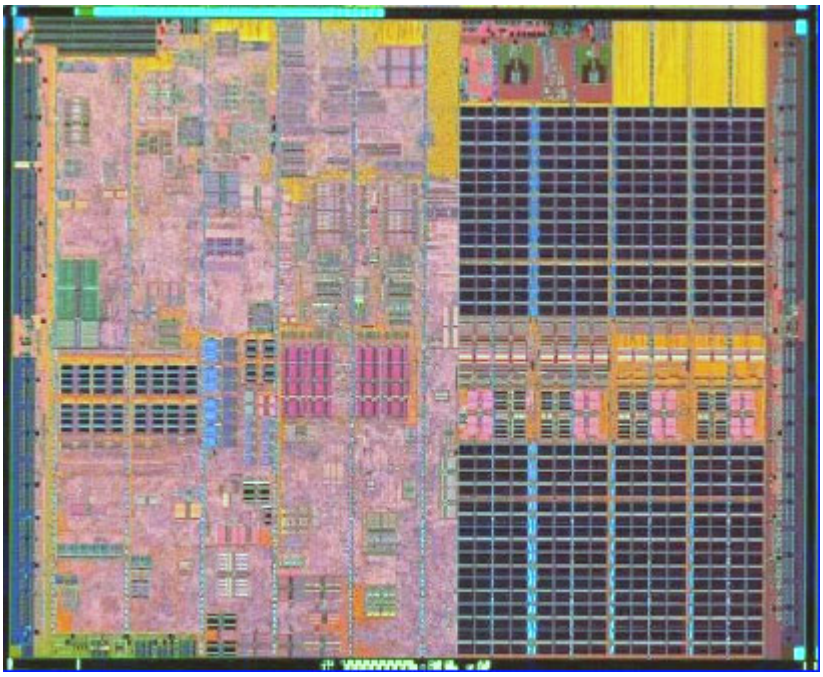
- 交付的可靠性误差——用户对交付中缺陷的接纳程度（用户能够接受一些非关键功能的失效）。开发组总是尽量减少交付中的缺陷数量，但实现零缺陷在理论上是做不到的；因此，只要将交付的缺陷率控制在一定范围之内，能够为用户所容忍就行了。
- 软件的健壮性误差——软件对规格定义之外的异常状况的适应能力（用户总是可能执行一些非法的操作等）。开发组应当在规格定义中就将异常处理囊括进去，并在设计时考虑更多的异常处理场景；这样用户的非法操作不至于引起系统的崩溃。
- 系统的性能误差——软件性能处于用户能够接受的范围之内。开发组尽力设计性能最优的方案，比用户的接受标准更高，为今后系统的演进留下更大的余量（例如用户数量随业务的发展而呈几何指数增长）。
- 范围类规格误差——软件中涉及到一些规格定义，这些规格本身是一种范围定义；开发组在不确定的时候，可以选择定义更大的范围值来规避风险。例如，为了避免数组的溢出，开发组可以尽量定义更长的数组结构；某个函数的某个输入参数，在运行时刻可能会是很大的整数，开发组将选择长整型作为参数的类型规格。
- 其它误差——例如，软件的易用性本身就是一个相对模糊的概念，开发组有很大的自由空间来满足它。

软件的复杂性

与软件类似的是数字电路，它同样不能容忍误差的存在；但幸运的是，数字电路的复杂程度不如软件那么高，这避免了追求精确与复杂性交织在一起所带来的巨大挑战。

从规模上看，先进的 CPU 可能集成了几千万个晶体管；但是这些晶体管元素的结构存在惊人的相似性。例如，一个主流的 CPU 可能具有 4Mbyte 大小的 2 级缓存，但其存储单元的基本结构都是一样的，只是重复了四百万次而已。这样，我们在设计这个 CPU 时，2 级缓存部分实际上只需要设计一个存储单元，其余的都是机械的复制而已。因此，数字电路的复杂度，并不与其晶体管的数量成正比。

参考下图中 Intel 双核处理器芯片的晶圆照片，我们可以看到其结构整齐划一，大片的细节都非常相象。



Intel Dual Core Processor Die

图 1.3 Intel 最新的双核处理器芯片晶圆照片

相对应的，在开发一个百万行源代码级的软件时，却享受不到这种重复的乐趣。实际上，软件中如果存在大量拷贝、粘贴式的重复的话，反而会给其维护带来灾难性的后果。软件与数字电路不同，功能完全一样的元素，在代码上都应当力求只出现一次。面向过程的语言，支持人们将重复的功能抽取成单独的子函数，并被重复调用；而面向对象的语言，则进一步提供了实例化机制的支持，鼓励人们将重复的内容定义一次（类），然后在运行时刻大量地实例化它们（对象）。因此，一个结构良好（底冗余度）的软件，其源代码行数基本上能够反映它的规模和复杂度。如果要寻找一种指标，来衡量人类对复杂事物的最高驾驭能力，那么，统计现有最复杂、最具挑战性的软件的所有源代码行数，也许是最有说服力的做法。

摩尔定律的幻觉

IT 界中有一个著名的摩尔定律，即电脑芯片中晶体管的数量每 18 个月将翻一番。这常常给人一种幻觉，以为数字电路的复杂度在急剧地膨胀，因而人类处理复杂事物的潜力也几乎没有极限。

上文所述的 2 级缓存例子，说明了一个事实——摩尔定律的存在，并不能证明人类的能力，真的象晶体管数量每 18 个月就翻番一样，在同步地提高。

相反，微软 Vista 操作系统的不断跳票，若干大规模软件的最终失败，以至于所谓“软件危机”概念的提出，却实实在在地严重打击了人类在解决复杂问题时的自信心。

在第一节，我们已经阐明软件用于处理业务（或者其它人造系统，比如国防武器系统）领域中的各类信息；也就是说，软件的内容要反映其所面对问题域中的事物。正是因为软件所承载的问题领域极其广泛和复杂，使得软件空前地复杂和规模庞大。例如微软的 Vista 操作系统，其源代码行数已进入千万数量级；可以理解的是，为了兼容成千上万种不同厂家所生产的硬件设备，每种设备都开发对应的驱动，那么，光是这些驱动软件，恐怕就占据了千万行源代码中的相当比例。

总之，软件的复杂性来源于其面对问题域的复杂性，和为了支持问题的解决，设计软件本身所带来的复杂性。

问题域本身的根本复杂多样性，通过工具或普通方法是解决不了的；而因为象使用汇编语言来开发业务软件之类做法，所带来的附加复杂性，则完全有可能使用更先进的工具来彻底解决掉（“软件是对现实世界的一种映像”段落图示中，UML 等建模语言在描述业务时，就远比汇编语言来的高效）。

消除冗余的终极手段

如何消除软件中的冗余，实际上是软件架构设计中的一个核心问题；有多种途径来去冗余，例如继承、元数据驱动技术等，我们将在后续章节专门做深入的探讨。

假设未来某一天，数字电路的运作原理发生革命性突破，我们或许可以模仿软件的机制，先定义好单个存储单元的结构，然后由 cpu 自己在物理上实例化出四百万个一摸一样的存储单元来。这样，数字电路的物理实现上便同样不再存在简单的复制类冗余。

软件元素之间的关联复杂性

人类对于复杂的事物，通常采用分而治之的途径来解决它。以往行业，比如汽车制造，可以将整车划分为若干个零件，先将这些零件制造好，然后再组装起来。汽车零件相对独立，相互之间的依赖比较松散；只要各个零部件的质量过关，最后组装起来的整车也基本没问题。软件的内部组成部件，通过良好的设计，同样可以做到一定的相互对立性；然而无论如何，集成后的整体软件行为，与单个部件的行为之间还是存在着巨大的差异。所有软件单元的质量通过了验证，并不等于集成后的交付也能够通过验证。

造成上述这种现象的根源，在于软件内部部件之间的关联比其它产品要复杂很多。软件所支持的问题域本身，其元素之间的关联与依赖关系就比较复杂；而软件本身所使用的各项技术，通常相互交织一团，并与问题域的内容整合在一起，造成部件之间的关系复杂度增高一个数量级。

传统项目，例如公路项目，投入的人力与工期能够互换；即增加人力，可以成比例地缩短项目的工期。然而，软件项目，由于软件元素之间的关联过于复杂，使得项目成员无论如何组织和分配任务，其相互之间的依赖性都很强。依赖的存在，造成项目成员难以并行工作，在已有成员都可能被迫停下工作来等待时，增加的人力无法对项目产生贡献。元素间众多复杂的关联关系，使得承担不同元素的项目成员之间，需要就这些关联进行对应的沟通，这种沟通量将随参与人数成几何指数增长；因此，增加人力后，即使能分担一定工作量，但其同时又增添了沟通上的不必要开销。总之，软件项目中，人月不能互换。（参见《人月神话》）

软件的不一致性、多样性

普通行业的问题域只有一个，例如建筑设计面对的问题不外乎就是如何设计合适的受力结构、使用合适的材料来构建一个满足各类建筑规则（力学等）的建筑体；另外，建筑行业经过多年的发展，已经非常成熟，建筑师能够利用大量的现成经验，使得其设计（解）空间比较窄小。

软件则要同时面对两个问题域。除了业务运作的问题域本身，还存在如何设计软件来支持解决业务问题的工作。在“软件是对现实世界的一种映像”段落中，我们看到问题域距离计算机域非常遥远；要将目标问题域最终转化为软件解决（机器码），中间还有大量工作要做。软件的解决方案，从普遍意义上看，不像建筑设计那样，存在明显一致的模式来套用，也没有一致的规律可循。企业级业务应用软件所采用的架构样式，与 CAD 桌面软件的就很不一样；实时软件常用的设计模式，与网站应用软件就毫无瓜葛。

企业业务流程的一个重要特点，就是其多样性和创新性。因为市场的竞争压力，企业业务流程不能同质化，而且变化非常频繁。业务问题域的多样性，带来了软件的多样性。

另外，普通行业的问题域，关注的是最终目标产品，例如建筑设计关注目标建筑本身的规律和约束。而软件的问题域，关注目标却要涵盖产品生产的整个过程。例如，

我们如果开发一个建筑的设计辅助软件，我们要研究建筑体本身的规律和约束，但同时，我们更关注如何进行建筑设计的过程和方法；因此软件问题域的范围极大地被扩展了。

软件的不可视性与主观性

从严格的意义上来讲，软件没有对应的物理存在形式。一张软件交付光碟，光碟（物质）本身只是一个载体或媒介，而软件则附着于那些 0/1 比特所构成的表示之上。虽然我们可以使用工具来间接观察软件的内容，但是在其被计算机执行之前，我们还是不能直接领会这些 0/1 比特所代表的真实意义。因此，与其它物品不同，软件在物理上是不可视的，而且也不能通过简单的工具来观察它。

人们完整和准确地感知软件的唯一途径，是在主机系统上运行软件；通过给系统以输入，然后观察系统的响应输出，来观察系统当前所运行软件的行为。如果软件使用了 GUI 图形界面的话，人们将对软件产生形象上的感知。另外，人们还可以感知软件执行所表现出的、在秒级以上范围的性能属性；对于毫秒级的性能，人们不能直接感知到，只能通过数据收集，以生成分析报告的方式来认识它们（性能测试比功能测试困难的原因就在于此）。

对于业务软件而言，其支撑的业务过程本身也存在不可视的问题。业务活动是动态的，我们不能以静态的方式来观察它；同时，业务过程又是长期的，我们无法以业务活动的某个片断来揭示整个业务的形态。这样，业务软件因为业务本身的不可视性，变得更为难以观察与感知。

软件的不可视性还表现在软件的开发过程上。传统工程，例如修建一条公路，设计师只要拿出设计图纸，那么整个工程的工作量、成本就可以较为准确地估算出来。然而，在软件交付完成之前，人们却永远也不能真正精确地估计开发的工作量。

软件的不可视，给软件开发活动带来了更多的主观性色彩。

软件被正确验证，取决于对软件交付行为的准确观察。而不能被执行的软件半成品，是不可视的。传统瀑布生命周期开发模型的失效，很多时候源自需求和设计没有被真正验证；瀑布模型下，可运行的交付在项目后期才能得到；在这之前，人们只能通过评审等手段来验证设计；而评审不能直接观察软件的行为，自然也就不能客观地对软件设计质量进行评判。

软件开发工作量、成本难以准确估算，使得软件项目计划总是充满了主观臆断。而项目的进展是否健康，项目经理也常常只能依赖其经验来做主观的判断。

测试是软件验证活动中唯一客观、也是最为可靠的途径。业务过程的长期动态特性，使得对应软件的测试工作难度成倍增大。

软件的易变性（可塑性）与不确定性

软件所面对的问题域本身经常就是含混与不确定的，这迫使软件需求要随着领域

的变化而进行调整。另外，软件对同样的业务，可以有不同的需求方案（软件复杂一些，使得业务的自动化程度高一些；而软件简单一点，同样也能改进业务的效率），这造成了软件需求的不确定。软件设计中，针对同一需求，可能存在多种实现方案；这也增加了软件开发的不确定性。

另一方面，软件本身容易被更改，可塑性 malleable 较强；这使得人们常常忽视变更背后的成本和潜在风险，更倾向于去决定变更软件。例如，有的客户往往不情愿在一开始就与开发者将需求的细节讨论清楚，并确定下来；而是指望在发现问题后，开发者有神通能迅速地修改好。而程序员往往也不喜欢花费太多的时间先去完善设计，而是直接投入编码，等到发现一大堆 bug 后，再去一一修改。

软件的不确定，迫使软件不得不变化；而软件容易被更改，又加重了人们轻易地决定变更软件的冲动。

软件易变性也有其有利的一面。人类思维的局限性，使得人们不可能一开始就做出一个完美的设计。软件的可塑性，则支持人们逐步精化、改进软件，乃至增量式地开发软件。例如，一条设计为四车道的公路已经构筑了一半，突然更改设计变为六车道，经过工人们的努力，总算变更成功；然而，我们总是会很容易地看出这条公路被改造过的痕迹（感官上将很别扭）。相对的，软件发生重大变更时，只要变更不超出可控范围，经过努力是有可能不在最终交付中留下任何被更改痕迹的（我们阅读一段最终交付代码，如果没有变更记录的话，通常很难看出它们到底经历了多少次的修改）。

有些预研性的软件，一开始很难想象其整体架构的模样；于是可以先从某个局部开始进行探索式开发，逐次开发一些代码来解决不同的问题，最后以滚雪球的方式整合出最终交付。这实质上是一种自底往向上的开发模式。软件的可塑性使得自底往向上的开发顺序在一定范围内有效。例如，汽车的生产过程，是先制造好所有的零部件，然后装配成整车的；然而，那些零件是根据事先设计好的规格来制造的，否则它们无法被整装在一起；汽车的生产，实质上是先自顶向下进行设计，然后自底往向上进行构造。而软件的自底往向上，很多时候是脱离整体设计而自然发生的；软件的可塑性，使得人们在集成时，可以重构这些原本独自开发的部件（对于不能更改的部件，还可以使用适配机制），最终将它们集成为一个完整的交付。

实际上，任何主要由人类思维来加工的制品，都具有一定的可塑性。回顾上文中汽车的例子，虽然到了物理生产阶段，汽车很难再被更改；但是在设计阶段，设计师却还有一定的空间来修改他的设计方案。这是因为物理形态的汽车变更成本高昂，而承载人类思维成果的图纸或模型，变更则比较容易。软件实质上是反映人类解决领域问题成果的一种载体，其物理形态主要为源代码，变更的成本较低。

不要被软件易变性的表象所蒙蔽

国内的软件项目，需求的频繁变更，往往成为项目失败的最主要原因。《人月神话》中将软件的易变性，列为软件最重要的四大特性之一。

从“软件是对现实世界的一种映像”段落的分析中，我们可以看到，软件易变性，归根结底来源于业务领域的不确定性。如果客户的业务本身非常混乱，那么，开发团队首先要做的，不是去考虑要开发一个什么样的软件，而是要帮助客户将其业务梳理清楚；否则，开发团队将不得不面对客户需求不断变更的窘境。

另外，软件需求并不能直接从客户或用户那里获取；开发人员从用户那里了解到的是他们对软件的期望和要求。

软件的复制式生产

与其它人造制品的生产不同，人们能够通过几乎零成本的复制方式来批量生产软件。这是因为软件本身也是一种信息，而信息是能够几乎零成本地被复制的。

正是这种零成本的复制生产方式，凸现了软件的可塑性。对软件代码的任何修改，只要使用工具重新自动编译一遍，就能轻易得到新的源本，之后再大量的复制，新的一批软件产品便这样被生产出来了。

这也决定了软件的质量，在生产阶段是没有太多实际意义的，因为软件的复制过程，可以非常容易地保证副本与源本不出现任何偏差。我们所关注的软件质量，应当是它被开发出来的那个原始样本的质量。

第三节　软件质量

我们了解软件的上下文，了解它的本质特性，其目的仍然是为了开发或应用软件；而开发软件的关注焦点则是如何得到令人满意的高质量软件交付。我们必须清晰地理解软件的质量属性、及其之间的关系，方能采取正确的渠道来获得这些质量结果。

由于软件自身的特殊性，使得其质量属性的内涵更为丰富，不同的质量属性之间可能相互加强、也可能相互冲突；评估这些复杂的属性是非常困难的，人们需要采用与以往完全不同的方式来管理软件的质量。

软件的质量与质量保证

与对待其它人造制品一样，人们要求软件产品可靠、容易使用，能够满足用户或客户的需要。这些都属于软件的质量属性。

软件被用来帮助用户执行业务活动或解决领域问题，因此其质量属性体现了其对解决业务或领域问题的支持良好程度。业务活动本身同样有正确、可靠、高效、迅速等相关的质量要求，人们在设计业务流程时就开始考虑如何满足这些质量约束，但最终这些约束大部分还是要落实到软件对应的质量属性上（例如功能正确性、可靠性、性能等）。另外，软件通常被人操作来完成业务任务，人的因素在此影响重大，因此软件的很多质量属性与用户相关（例如易用性、操作健壮性等）。

软件的质量与其成本（包括获取与使用成本）、交付时间一道，成为客户是否愿意接受它的三个要素。软件的质量如果不能达到用户所能接受的最低水平时，客户将放弃此软件；而当质量满足最低要求时，质量的高低将成为影响软件总体拥有成本的关键因素——质量较低的软件也许只须付出较低的开发成本，但其学习成本、操作成本较高（不方便、容易出错、常常死机等，使得用户时常要付出额外的努力），并且维护成本高，因此其总体拥有成本并一定较低。实际上，在三要素中，质量通常是刚性的指标，人们可以缩减范围、增加成本来保证工期，但降低质量要求的做法是非常危险和难以被人接受的。

从不同涉众的角度来划分，软件具备外部与内部两种质量属性；客户或用户所关注的可靠 reliable、有效 efficient、和容易使用 easy to use 等属于外部质量属性；而软件开发者所关注的可验证 verifiable、易维护 maintainable、可移植 portable 和可扩展 extensible 等，则属于内部质量属性。用户所期望的软件外部质量，依赖于其开发者对内部质量的追求而最终得以被实现。

从软件产品本身与开发它的过程来区分，又包含软件产品质量与软件的开发过程质量两个范畴。现代质量管理理论，认为产品的质量是由制造它的过程所决定的。质量保证部门将通过控制软件过程的质量，进而来达到控制软件产品交付的质量。

现代工业制品都要经历一系列的工序，最后才被生产出来。软件同样如此，这样整个软件的开发过程中，将出现若干中间制品（半成品）序列，越靠后的中间制品将越接近于最终交付产品。其间的质量管理原理，在于确保每个中间制品的质量（软件

的内部质量属性），从而实现最终交付产品的质量（软件的外部质量属性）。

确定项目中要开发哪些中间制品，中间制品的质量如何被验证，相关的活动顺序如何安排，开发组人员如何组织等，这些都属于软件过程的范畴。开发组织不会每个项目都重新去探索一套新的开发方法，相反有大量现成的、被验证为有效的过程可以给开发组参照；开发组只要严格地遵循那些过程，项目最终成功的可能性就非常大。因此开发组在项目展开之前就会选定一种合适的软件过程，据此制定项目计划；开发组按计划遵照既定的过程来开展项目；而质量保证部门将监督过程的执行是否偏离既定的规范，在此，过程质量就是过程被开发组遵循的良好程度。

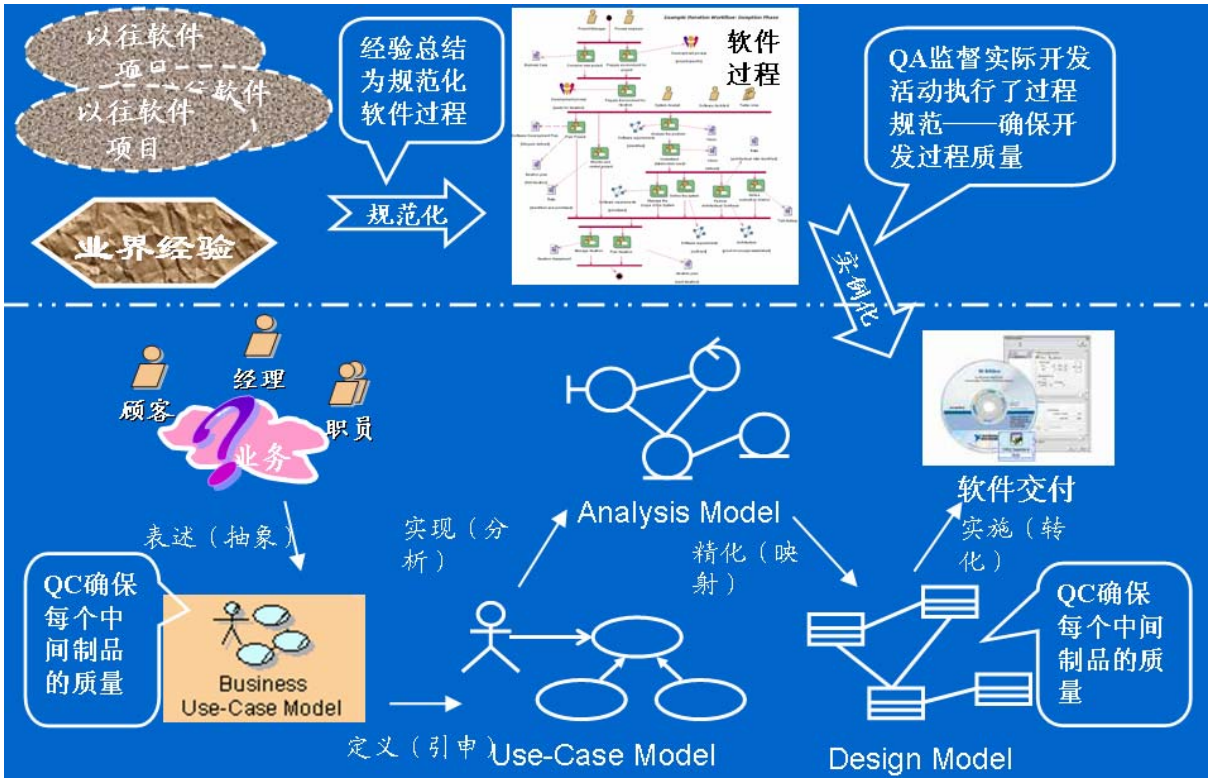


图 1.4 软件开发过程质量保证原理示意

实际上，现代软件质量管理人员也相应地被划分为两类，一类是 QC 质量控制人员，他们负责测试与验收软件交付与中间制品，以确保制品本身的质量；另一类是 QA 质量保证人员（很多人将 QA 和 QC 混为一谈），他们监督开发过程是否被遵循，通过确保过程的质量来保证项目能够重复以往的成功。

软件的质量属性与折衷

软件质量属性在软件被开发之前便会被定义，成为软件开发的主要需求之一。与功能性需求以及环境需求不同，质量需求描述的是对系统好坏的要求，体现系统“量”方面的差别。

不同的质量需求对系统的实现可能产生不同甚至相反的影响（例如实现可审计性将增加处理时间），因此系统的开发将是对各项需求的折衷过程。因此质量需求往往难以全部百分之百地被实现。

软件功能性 Functional 要求

软件必须具备满足用户相关应用目标（支持用户的业务活动，或解决领域问题）的基本能力，为客户提供可见的使用价值，也就是说软件要有用。软件的用处，在于其功能，并体现为功能性质量属性。它们属于软件外在的与其使用价值有关质量属性，用于衡量软件为客户所提供的价值。

- 适宜性 Suitability——软件支持业务（或领域应用）的必要功能必须存在，这些功能应当与其应用的业务（或领域）场景相适宜，否则软件就失去了其使用价值。适宜性是软件最根本的质量属性，满足不了它，软件将一钱不值。
- 正确性 Correctness——软件交付是精确的，符合在计算机上运行的所有要求；同时其执行将准确地实现事先所定义的功能规格，从而满足用户的需要。正确性包含两个方面：软件的行为是准确的，不能与规格定义有偏差；而软件要在主机上运行，其执行码哪怕是存在一个字节的误差，都会导致其整体失效，因此它同时也必须是精确的。
- 完备性 Completeness——软件提供的功能，相对于业务（或领域应用）、和用户操作的需要而言，应当是完备和全面的。软件功能如果缺少对某些业务环节的支持，将可能严重影响业务运作效率；而缺少相关的用户操作途径，则可能阻碍用户的工作。
- 互用性（互操作性）Interoperability——用户常常需要将目标系统与其它系统或工具一同配合使用，来完成相关业务活动；因此软件必须具备和特定系统互相作用（协同）的能力，并与环境和其它系统相适配而不冲突。
- 依从性 Compliance——软件产品必须依从法律或类似规定中与应用程序相关的标准、协定、规章，否则可能产生法律纠纷或带来其它的不便。
- 安全性 Security——软件产品需要具备防止未经授权访问程序或数据的能力，以免造成意外的损失。

软件有用，首先源至其使用价值，即软件对业务或领域应用的功能适宜性；同时，软件还必须正确地执行这些适宜的功能，以满足正确性要求，否则其使用价值无法真正被实现。而功能的完备性，则体现了软件使用价值的完备性。除此之外，软件具备

一定的互用性，则能够进一步增大其附加值。

软件的依从性和安全性，确保用户在获得其使用价值的同时，不会带来令人不快的副作用（法律纠纷、意外损失等）。

软件可用性 Usability 要求（易用性）

软件有用，但是如果操作起来相当繁琐、不好理解和难以记忆，让用户感觉别扭，那么用户仍然不会愿意使用它。可用性质量属性主要体现了系统对用户的友好性（交互质量），将直接影响用户使用软件的意愿。它们都属于软件外在的与其操作难易程度有关的质量属性。

- 简单明了 Conciseness——软件的操作方式应当简洁、没有复杂而繁琐的步骤；其行为表现直观明了，让人容易把握。
- （风格）一致性 Consistency——软件操作的风格、模式，表达的概念以及逻辑关系，应当是一致的。例如，界面上通常使用带阴影的独立 3D 圆角矩形来表示按钮 button 控件，如果同时又用它来凸现 highlight 需要引起注意的文本，那么这种不一致将让用户感到困惑，并容易导致误操作。
- 可理解性 Understandability——软件的操作方式与行为表现，应当符合常识，不与用户的认知习惯相冲突。软件任何逻辑上的不合理，都将使得用户难以理解其操作，并最终严重阻碍用户的使用。
- 可学习性 Learnability——与用户学习软件操作所需努力程度相关的软件属性。一般体现为软件所配备演示程序、试用示例、用户手册、在线帮助与培训材料等的完备性与质量等。
- 可操作性/易访问性 Operability——与用户操作、控制软件所需努力程度相关的软件属性。包括：系统的可见性（足够的反馈信息，友好的提示）、操作方便性（诸如较少的操作以完成较大的功能等）、响应一致性（系统的行为表现与用户的操作意图相一致）。
- 操作健壮性 Robustness——软件对用户错误操作或外部异常输入的免疫能力等相关质量属性。

软件操作的简单明了和风格一致是获得其它易用性质量的基础条件。而软件的可理解，是用户愿意并能够操作软件的前提条件；不可理解的东西，其可学习性和可操作性也就无从谈起。可学习性强的软件，极大地降低了其推广应用的门槛，同时还节省了培训成本。软件的操作性强，则方便了用户的使用，并提高了用户处理业务的效率。另一方面，人非机器，操作软件时总是容易出错，如果软件应付不了，并因此而失效或崩溃，将会引发用户的严重抵触情绪。

此外，以下各项属性也属于与操作有关的质量属性，它们体现了用户交互与界面的质量：

- 可见性 Visibility——好的软件界面应当让用户通过简单的观察，就能领会如

何进行操作。改善界面可视性的途径有：提供可视的指示来引导用户 User guidance/ Prompting；在界面上对类似的功能与显示内容进行分组 Grouping/distinction of items，方便用户感知；随着操作的展开逐步启发用户 Incremental Revealing。

- 暗示 Affordance——软件界面上各类对象的形态，应当直观地提示用户其作用和应当如何被使用。常用的暗示方式有：隐喻 Metaphors。例如，Windows 图形界面借用了桌面的概念，它隐喻了 Windows 主界面的一些基本操作方式；另一个例子，是按钮控件都显示为 3D 凸现的形态，暗示用户可以按下它。
- 自然应对 Natural Mapping——针对用户想要完成的任务，软件提供清晰和自然的对应机制来实现。软件的界面及交互方式，与其实现的任务之间是协调的 Compatibility、相称的；任何不自然、别扭的做法，都会降低软件的可理解性。
- 约束（交互行为）Constraints——软件在交互设计上，有意地限制用户的操作路线，从而减少用户为了执行相关任务所需掌握的知识。对应的做法有：最少化行动步骤 Minimizing actions，即将完成一项工作所需的操作步骤减少到最低数目；自解释 Self-explanation，即界面的展现能够说明用户操作的意图以及系统响应的变化结果，例如，界面的布局 layout 随用户的操作而发生变化，但用户一眼就能看出他一直与之交互的界面元素在何处。
- （交互）概念模型 Conceptual Models——软件的交互过程应当遵循一定的规则；例如，为了完成某个任务，用户需要提供足够的信息，并对系统下达相应的指令，而系统的响应与执行结果要符合用户的期望；用户对系统交互行为的一种理解和预期，被抽象为某种（交互）概念模型。遵从概念模型的软件，使得用户能够理解并预期系统的响应。具体的做法有：在界面上使用意义明显的代码（标识）或名称 Significance of codes，以便用户理解其背后所引用的含义；确保软件的交互行为（输入、输出、以及对话）与用户的特点（心智水平、知识技能等），以及目标任务的特性相协调 Compatibility，另外，这种协调关系也着眼于应用软件与其环境平台之间。
- 明确的控制 Explicit control——为了避免软件的行为失控，软件交互的设计必须是明确而无二义的。一方面，用户的动作 explicit user action 与软件对应的处理之间是关系明确而无二义的（例如，系统只处理那些用户真正需要的行动，而且是在用户确实发出了请求的时候）；另一方面，系统任何时候都应当处于用户可控 user control 的状态，即系统必须响应用户的指令（例如，中断、暂停、取消或继续系统的工作等）。
- （系统）响应 Feedback——软件应当及时给用户的操作以反馈，以便用户把握任务执行（交互）的进度和状态，防止用户接下来执行错误的行动，使得交互过程陷入混乱和失控。具体策略包括：系统及时地反馈 Immediate Feedback，并且反馈的内容让用户容易理解和辨认 Legibility（例如，以闪烁

的方式凸现需要被重点关注的文本）。

- （操作）安全可靠 Safety——用户容易出错，因此软件应当足够健壮，能够让用户较为随意地操作，不必担心因为不小心而造成重大损失。具体策略包括：使用错误预防 Error Prevention 技术来避免用户执行错误操作；如果不幸出错，则软件具备错误修正 Error Correction 能力，帮助用户进行扑救；另外，软件通过专门的设计，能够容纳（宽限）Forgiveness 用户的某些错误操作，自动地修正那些明显的错误。
- 适应 Adaptability 能力——用户在操作软件的过程中常常临时改变心意，这要求软件必须提供足够的弹性，来适应各种变化。一方面，需要提高软件的弹性 flexibility，使得其行为表现能适应不同的上下文场景和用户的多样化要求与偏好；另一方面，这种适应能力最终将影响用户的体验 user experience 感受。

软件可靠性 Reliability 要求

软件有用，而且好用，但是如果经常突然挡机或无法使用，用户肯定不会满意这种提心吊胆的感觉。因此软件还必须可靠，让人放心使用。可靠性质量属性，主要体现了系统持续不间断地满足客户相关应用目标的能力。它们属于软件外在的与其使用价值可获取度有关的质量属性。

- 有效性 Availability——与软件持续正常运行，而可供用户使用相关的软件属性。例如平均可用时间等。如果在用户急需软件来完成某项任务时，系统却不能使用，那么将有可能严重损坏客户的利益。
- 成熟性 Maturity——与由软件中的缺陷而造成故障的频度相关的软件属性。例如软件的系统缺陷率、平均无故障时间 MTBF 等。
- 故障承受能力 Fault tolerance——软件在运行时，有可能会遇到故障（例如硬件失效），或者其某些特定接口部分遭到侵害；在此情形下，软件应当仍然保持一定工作能力，从而避免彻底中断服务而造成的更大损失。
- 可恢复性 Recoverability——一旦遇到故障，软件重新恢复工作性能所需的时间越短，受损数据的修复程度越高，则因故障而造成的损坏就越轻。
- 可预测性 Predictability——软件的运行状态与行为是可以预测和把握的。行为不可预测的软件，将使得用户无法掌控它的运作，会给客户带来巨大的风险。

用户关注软件所带来的使用价值，而这种使用价值必须是随时能够获得的；也就是说，软件有用、好用，还要可靠、能用。软件有效性体现了它持续能用的程度；成熟性是软件在正常情形下确保有效性的基础；而故障承受能力、和可恢复性则是软件在异常情形下提高有效性的主要途径；另外，可预测性帮助用户消除因使用软件而带来风险的疑虑。

软件可靠性质量，往往难以独立于硬件之外来单独度量，而且其本身与硬件有密

切的关系，但与性能、效率等不同，其本质上还是属于软件的一种属性。

软件性能 Performance 与效率 Efficiency 要求

软件有用，且好用，同时也很可靠，但是如果用户执行一项简单的操作，就得等上几分钟，恐怕谁也没有这么好的耐心来忍受这种等待的煎熬；因此，软件的执行必须具备让人可以接受的性能。性能等质量属性，是软件与硬件一同协作，所体现出的系统满足客户对相关运行效率、速度与规模方面要求的能力。它们属于软件外在的与其时空特性有关的质量属性。

- 时间行为 Time behaviour 上的效率——与响应、处理时间以及执行功能时的吞吐率相关的软件属性。

响应时间：系统从用户提交请求到返回处理结果的时间。

响应速度：系统对用户操作的反馈速度。有反馈并不意味系统已有了处理结果；在响应时间过长的情形下，中途插入及时的反馈，可以避免用户对系统是否死机的疑虑。

处理时间：系统处理用户所提交请求的时间。

延迟时间：用户提交的请求传递给系统，和系统的反馈结果返还用户所花费的时间。

吞吐量（扇入扇出量）：系统在单位时间内完成的处理量。

负载（并发性）：系统在单位时间内同时被服务的用户数量。

处理能力：系统可有效支持的最大负载（通常是能够满足可接受的响应时间阀值之下的最大负载）。

另外，还有关机时间、开机时间、恢复时间等。

- 资源行为 Resource behaviour 上的效率——与系统完成某项任务时，所使用的资源数量及其持续时间相关的软件属性。包括：内存占用量、存储空间、CPU 的占用率、网络带宽使用率等。

软件的功能属性体现了其使用价值中“质”的方面，而性能等属性则体现了其使用价值中“量”的方面。例如，负载或并发用户数体现了软件使用价值在空间上的量；而响应时间则体现了软件使用价值在时间上的量。换句话说，性能等属性代表了软件功能行为在时空上所表现的附属特性。

软件有用、好用，而且可靠，同时还要速度快，这样单个用户才有长期使用它的意愿；而软件还应支持更多的负载，这样才能让众多的用户来接受它。

软件的性能，与可靠性、易用性一样，都会直接影响软件运行过程中的使用成本。例如，性能低下的软件将浪费用户大量的时间用于等待。

软件性能等质量属性，不能独立于硬件来度量，而且其本身体现的就是软件对硬件资源的利用效率。我们无法脱离硬件来单独定义软件的性能规格，因此，在定义性能需求时，必须列出对应的硬件配置规格。（笔者曾经见到不少项目的需求规格说明中，在性能需求这一项，就没有列出对应的硬件配置）

软件可支持 Supportability 与维护性 Maintainability 要求

软件的外部质量属性，归根结底是由其内部质量属性所决定的。这些内部属性中，诸如软件架构质量、源代码质量、其它中间制品质量等，主要用于衡量开发过程中间产出的质量属性，并为软件开发者所关注；而类似软件可支持、可维护性、可扩展性等，主要在软件开发成功之后，用以衡量软件满足客户长期应用系统而保持较低成本的能力。后者将同时为客户、开发者所关注。

软件被交付之后，并不意味着开发者的工作可以完结。用户在使用软件的过程中，总是可能发现遗留下来的 bug；或者发现有的功能操作不便，需要改进。这些问题都需要开发者来加以解决。而下列软件的内部相关质量属性，将直接影响到整个维护过程中的成本。

- 可验证(测试)性 Testability——与验证被修改软件所需努力相关的软件属性。软件具有不可视的特性，测试是让软件可视，同时也是唯一能客观验证软件质量的途径。然而测试是需要付出代价的，如果没有针对测试进行专门设计的话，测试的成本将非常高昂（例如，准备大量的数据、配置复杂的前置条件等）。高可测试性，能够大幅度降低开发成本，和降低软件质量风险。
- 可分析性 Analyzability——与诊断缺陷、故障，或者确定需要修改的部分所需努力相关的软件属性。“分析”的字面解释是：把物质分为若干成分来确定它们的性质（定性分析）或比例（定量分析）。为了定位软件缺陷的根源，通常要划分不同的区域来分别进行判断与尝试确诊；而软件的内部结构支撑了这种划分。所谓“可分析性”主要受软件结构的良好与可理解性、代码的可读性、元素间的低耦合性等因素的影响。
- 可变性 Changeability——与修改、去除错误、实施需求或设计变更（例如改变运行环境）等难易程度相关的软件属性。这是度量软件内部架构合理性、代码冗余度、可读性、元素间耦合性等质量的重要指标。
- 稳定性 Stability——与修改过程中未预料到作用的风险相关的软件属性。软件容易修改和变更，但是如果一旦实施了变更，就会造成软件的崩溃与失效，并引入更多新的 bug，那么可变性仍然只是一句空话。
- 可审计性 Auditability——与找出客观的证据来检验软件是否符合标准、规范或既定规格之难易程度相关的软件属性。为了避免人为的主观因素影响，应当尽量使用客观的证据来证实软件的质量、开发进度等内容。所谓“审计”是一种事后的核查措施，因此那些用于审计的证据，主要是可以被记录、并长期保存的工件。这些工件与软件交付的关系愈密切，记录完备与保存完好程度愈高，软件的可审计性就愈强。配置管理中的逻辑审计与物理审计是软件项目中的主要审计途径。
- 可维护性 Maintainability——与软件在使用过程中被修改、更新、演进和修复所需努力相关的软件属性。它基本上由上述其它可支持性质量属性所综合决

定。

移植适应性 Portability 与可扩展性 Extensibility 要求

软件被交付之后，除了普通的维护需求之外，很多时候还会遇到客户的其它要求。客户有可能升级了新的硬件设备，而原有软件并不能直接在新的平台环境下运行；客户的业务不断发展，业务量急剧增长，原有系统的性能无法支撑；客户面临很大的竞争压力，需要开展新的业务，而原有软件不具备相应的功能。这些问题最终都依靠开发者来加以解决。下列软件的内部相关质量属性，将直接影响到解决前述这些问题的成本。

下面的质量属性主要体现了系统满足客户不同的应用场景与环境要求的能力：

- 适应性 Adaptability——与是否能适应在不同环境下运行使用相关的软件属性。
- 一致性/兼容性 Conformance——软件与特定标准、协定，以及平台环境相容的软件属性。
- 可配置性 Configurability——与软件为了适应不同的环境和应用场景，而进行简单的配置以便实现此目的所付出努力相关的软件属性。
- 可移植性 Portability——将软件通过重新编译或改造，从而能够在新的环境下被运行使用，为此所需要付出努力相关的软件属性。例如平台移植、本地化移植等。
- 可安装性 Installability——与将软件安装在指定环境中所需努力相关的软件属性。

下面的质量属性主要体现了系统满足客户未来发展与变化了的应用场景与规模要求的能力：

- 功能可扩充性 Extensibility——向系统增加新功能的能力。
- 性能可伸缩性 Scalability——通过扩充硬件效能，而实现提高系统处理效能的能力。

软件支持维护性以及移植扩展性等质量属性，实际上不仅仅只在软件交付之后发生作用，它们同样会影响软件开发本身。例如，软件开发过程中总是有大量的测试与 bug 修复活动，较高的可验证（测试）性、可分析性、可变性以及稳定性等都直接有助于上述活动的执行；又如，软件的构造过程，总是从部分到全部逐步完成的，高可扩充性的架构，使得开发者能够逐次地向软件交付添加功能增量（实际上，迭代开发就严重依赖于这一内在的质量属性）。

软件开发阶段内部质量要求

软件在交付之后的质量属性，无论是外部属性还是内部属性，都是由软件在开发

阶段的内部质量属性所决定的。这些质量属性，主要体现于不同中间制品的质量属性上。

- 业务建模质量属性——业务模型应当**准确（性）**地反映客户的业务现状 isBe 或期望状况 toBe；同时它还是**完备（性）**的，能够反映客户业务的全局关系；另外，业务模型必须是**可理解（性）**的，方便涉众进行沟通时使用。
- 需求质量属性——软件规格定义（需求）相对于其支持的业务或领域应用而言应当是**适宜（性）**的，以满足业务和用户操作的需要；它同时还必须是**精确（无二义性）**的，以避免理解偏差和最终交付的不一致；而需求规格定义的**完备性**则决定了软件交付的功能完备性；为了实现软件的易用性质量属性，需要**交互设计良好**，以消除交互逻辑上的缺陷；需求的**可理解性**往往也决定了软件操作的可理解性；最后，需求必须是**可验证（性）**的，否则软件交付将失去验收的客观依据。
- 架构与分析设计质量属性——软件设计必须是**正确（性）**的，要满足各类功能与非功能需求；架构设计应当**结构良好**，模块的划分是合适的，职责的分配是自然的；软件架构应当是**健壮（性）**的和**可扩展（性）**的，支持方便地添加功能，同时又不打破原有结构；设计的各个模块或元素本身应当是**高内聚（性）**的，模块或元素之间的关联应当是**低耦合（性）**的；软件设计在整体上是**低冗余（性）**的；另外，设计模型应当**简洁优雅（性）**、具备较强的**可理解性**，方便开发人员掌握和遵照实施；最后，软件架构还应当具备较强的**可测试性**，大幅减少测试成本，并最终有助于提升软件交付的质量。
- 代码质量属性——代码必须**正确（性）**地实现设计规格中所定义的功能；同时还提供一定的**健壮（性）**，以容纳一些超出规格定义之外的输入参数、以及错误调用；另外，代码应当具备良好的**可读（性）**，例如使用意义明确的命名，书写足够的注释等。
- 测试质量属性——测试案例必须是非常具体而**可执行（性）**的；通过执行这些测试案例，能够达到要求的代码语句或分支等**覆盖率**；测试活动应当是**有效（性）**的，能够最大限度地发现代码中隐藏的缺陷。
- 部署质量属性——软件交付应当是**易部署（性）**的，例如，提供了自动安装工具；部署后的软件副本应当是**易配置（性）**的，方便客户进行调整来适应其不同的要求。

本书将在后续章节具体讨论上述质量属性，特别是如何来实现它们，从而最终实现软件交付之后的各项质量属性。

软件质量属性的相互关系

软件的质量属性种类繁多，分别涉及到软件的多个方面。而这些质量属性之间存在着密切的关联关系：有的质量属性被其它质量属性所支持；而有些质量属性则相互增强对方；另外，也存在一些质量属性，它们之间存在相互冲突。

下面的图表，列出了主要质量属性之间的相互关系。

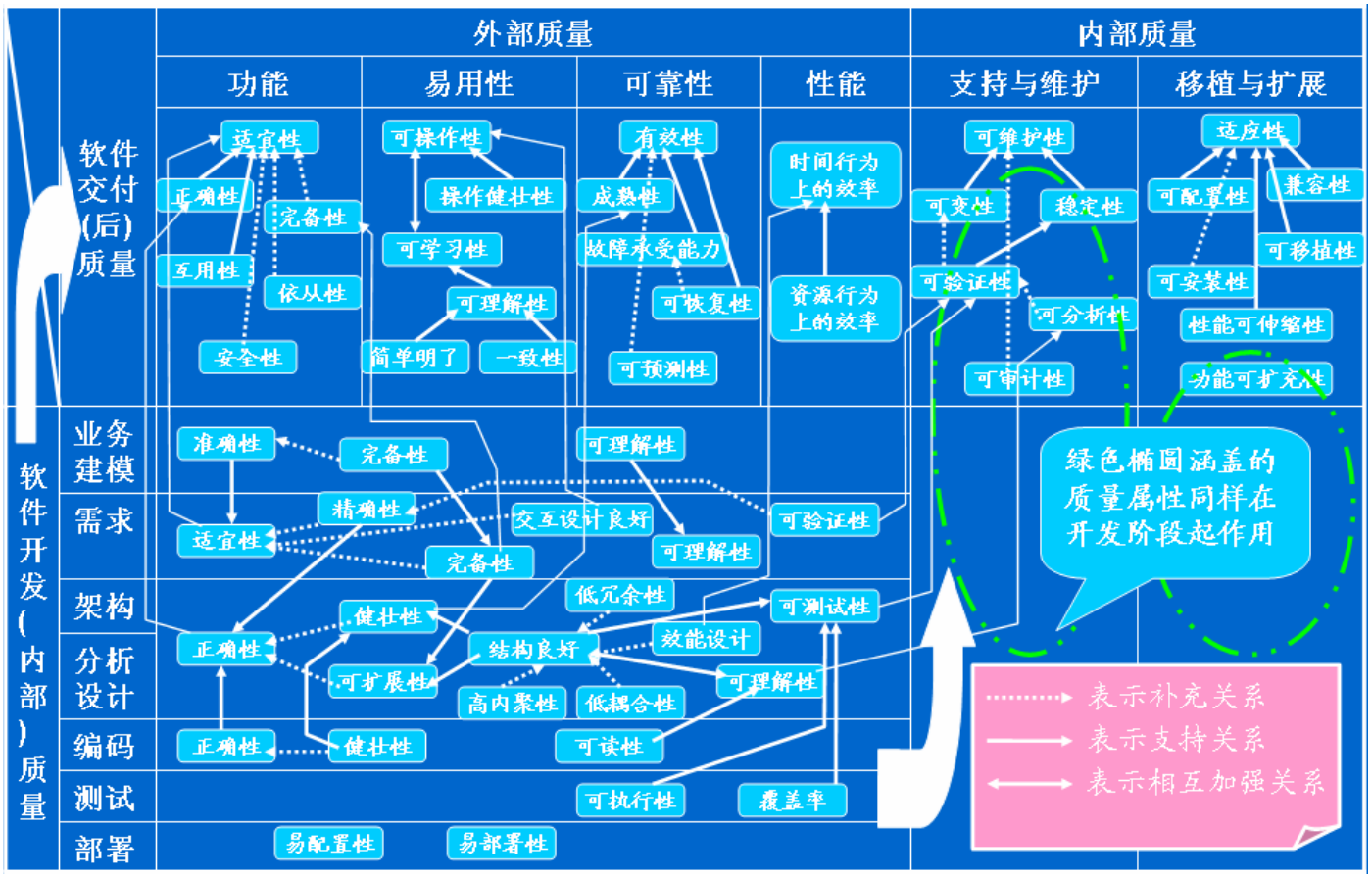


图 1.5 软件质量属性关系示意

本章小节

要构造软件，包括设计软件的架构，就必须先理解软件是什么，有什么特性。

我们应当从更大的视角来看待软件，从历史中了解它的缘起及其来龙去脉；从其上下文出发，消除将软件看作是一种特殊而相对独立的事物的误解，理解软件属于信息技术的一部分，是与硬件技术一道用来解决人类所面临的各种与信息有关的问题。

软件是人类有史以来创造的一种非常特别的制品，它具备与传统制品完全不同的特性；我们对软件的大体映像，便建立于这些根本特性之上。软件的特性既来自信息技术本身，也来自其所要解决的问题域。

我们开发软件的关注焦点是如何得到令人满意的高质量软件交付。我们必须清晰地理解软件的质量属性，方能采取正确的渠道来获得这些质量结果。由于软件自身的特殊性，使得其质量属性的内涵更为丰富，一种质量属性可能支持其它质量属性的实现，也可能与其它质量属性相互加强、或者相互冲突。