一、软件验证与验证（评估）

软件工程的目标

- **构建高质量的系统**：按时、按预算完成项目，并具备所有初始规定的功能和特性。
- **软件危机**：许多软件项目未能达成上述目标。
    - **16.3% 的项目成功**：按时、按预算完成，具备所有初始规定的功能和特性。
    - **52.7%**
      的项目超支、延期或功能减少：项目完成并投入运营，但超出预算（189%），延期，并且功能和特性少于初始规定的（42%）。
    - **31% 的项目被取消**：在开发过程中某个阶段被叫停，未投入运营。

软件为何难以构建

- **复杂性**：软件系统的极端复杂性导致完成和运行的系统中存在残留错误。
    - **决策路径多**：几百行代码的程序可能包含数十个决策，意味着成千上万的替代执行路径。测试所有可能的路径是不现实的。
    - **现实情况复杂**：软件响应的现实情况几乎是无限的。
- **知识缺失**：我们确保软件可靠性的能力远低于应有的水平。
    - **数学分析不足**：其他工程师使用数学分析预测系统行为，从而在产品投入运营前发现缺陷。传统的数学方法适用于描述物理系统，但不适用于软件的合成二进制宇宙。
    - **离散数学不成熟**：软件系统领域受离散数学支配，这是一个相对较新且不成熟的学科，直到计算机出现后才得到深入探索。

Verification与validation

- **定义**：
    - Verification：根据IEEE标准，验证是评估产品、服务或系统是否符合法规、要求、规格或施加的条件。通常是一个内部过程。
    - validation：根据IEEE标准，验证是确保产品、服务或系统满足客户和其他已识别利益相关者的需求。通常涉及外部客户的接受和适用性。
- **Boehm 的定义**：
    - Verification：建立软件产品与其规范之间的对应关系的真实性（源自拉丁词“veritas”，意为“真理”）。
    - validation：建立软件产品对其操作任务的适用性或价值（源自拉丁词“valere”，意为“值得”）。
- **非正式定义**：

- Verification：我在正确地构建产品吗？
- validation：我正在构建正确的产品吗？

缺陷评估-Evaluating Functionality

- **目标**：在开发产品的各个阶段寻找缺陷。
  - **缺失的任务**：目标任务缺失。
  - **错误响应/效果**：出现错误的响应或效果。
  - **导致失败**：这些缺陷最终导致系统失败。
- **错误、故障和失败**：Error Fault Failure
  - **错误**：人类的错误推理导致的人类错误，最终引起故障。
  - **故障**：错误被写入软件产品，产品存在问题。
  - **失败**：软件系统未按预期行为，表现为故障。

静态评估与动态评估

- **静态评估**：
  - **目的**：检测故障。
  - **适用对象**：任何开发产品。
- **动态评估（测试）**：
  - **目的**：生成测试用例，检测失败。
  - **适用对象**：仅限代码。

软件质量控制和保证活动

- **预防措施**：在开发过程中评估开发产品，而不是等到代码完成后再评估。
- **代码彻底测试**：确保代码经过彻底的测试。

软件质量

- **目标**：交付高质量的产品。
- **质量定义**：一个抽象的概念，需要分解为更具体的属性。
- **质量标准**：
  - **可靠性**：系统不会失败。Reliability
  - **功能性**：系统完成用户期望的任务。Functionality
  - 功能性

    - **效率**：系统以合理速度响应。Efficiency
    - **可用性**：用户使用起来舒适。Usability
    - **可维护性**：易于修改，且更改成本合理。Maintainability
    - **可移植性**：在不同环境中运行。Portability
    - **安全性**：防御入侵和未经授权的资源使用。Security
    - 非功能性
- **功能性**：系统完成用户期望的任务。
- **可靠性**：系统不会失败。

如何测试

- **可靠性**：通过寻找导致系统失败的缺陷来评估。
- **功能性**：通过将系统完成的任务与用户期望（需求规范）进行比较来评估。
    - **任务是否完整**：系统是否完成了所有任务？
    - **任务是否正确**：系统是否给出了正确或约定的响应或效果？

何时评估

- **评估时机**：在开发过程中尽早评估可能的缺陷。
    - **任务缺失**：系统未能完成所有任务。
    - **任务错误**：任务执行错误（错误的响应或效果）。
    - **系统失败**：系统失败。
- **早期缺陷**：许多代码缺陷在早期阶段（需求、设计等）就已经形成。

修复软件的成本

阶段	相对成本
需求	0.1—0.2
设计	0.5
编码	1
单元测试	2
接受测试	5
维护	20

总结

- **软件评估的重要性**：开发产品应在开发过程中进行评估，而不是等到代码完成后再评估。
- **代码彻底测试**：确保代码经过彻底的测试。
- **质量目标**：软件工程师的目标是交付高质量的产品。

静态评估

概述

- **软件质量控制和保证活动**：
    - **预防措施**：软件评估（静态评估）。
    - **纠正措施**：软件评估（动态评估）。
    - **良好软件构造实践**：包括方法论、不执行软件的检查（静态方面）、任何软件产品的检查、执行软件结果的检查（动态方面）、仅限代码的检查。

静态评估的好处

1. **降低成本**：
    - **早期检测**：早期发现缺陷可以降低后续阶段的成本。
    - **需求缺陷**：52% 的软件项目平均只能交付预期功能的 42%，不完整的需求是需求阶段最大的问题来源（13.1%）。
    - **设计和代码**：静态评估可以检测到 30% 到 70% 的设计和代码缺陷。
    - **节省成本**：在静态评估期间进行修正更容易且成本更低。
2. **质量估计**：
    - **早期检测**：早期检测到一个缺陷意味着可能存在更多缺陷。
    - **质量评估**：产品中检测到的缺陷数量可以反映开发工作的质量。
    - **纠正措施**：如果测量结果显示工作质量不足，可以采取纠正措施。

缺陷类型

- **一般类型**：
    - **正确性**：程序或要求是否正确。
    - **完整性**：是否包含所有必要的信息。
    - **冗余性**：是否存在重复的信息。
    - **一致性**：是否存在矛盾的信息。

- **模糊性**：是否容易产生歧义。
- **可追溯性**：是否可以追溯到需求或设计。

- **具体类型**：
  - **需求缺陷**：
    - **正确性**：不正确的需求没有指定所需的内容。
    - **完整性**：需求包含所有必要的信息，不包含不必要的信息。
    - **冗余性**：不存在关于同一问题的多个要求。
    - **一致性**：不存在矛盾的要求。
    - **模糊性**：需求不应有歧义。
  - **设计缺陷**：
    - **正确性**：设计是否符合对应的要求。
    - **完整性**：设计包含所有必要的信息，不包含不必要的信息。
    - **冗余性**：设计中不存在处理同一要求的多个部分（除非特别要求）。
    - **一致性**：设计中不存在矛盾的部分，不同模型（静态、动态等）之间匹配，与需求一致。
    - **模糊性**：通常与完整性相关。
  - **代码缺陷**：
    - **正确性**：没有符号错误，行为符合设计（因此符合需求）。
    - **完整性**：提供系统所需的所有功能，不提供额外功能。
    - **冗余性**：不存在实现同一功能的两段代码。
    - **一致性**：与设计和需求一致。
    - **可追溯性**：与需求和设计一致。
    - **清晰性**：代码应清晰易懂。

缺陷的分类

- **基本产品缺陷（正确性）**：不依赖于早期产品的缺陷。
- **内在产品缺陷（verification）**：通过与阶段输入产品的关系检测到的缺陷。
- **外在产品缺陷（validation）**：只能通过与需求（通常是需求）的对比检测到的缺陷。

静态技术与阅读技术

静态技术

- **三类技术：**
  - **评审：**
    - **技术评审**：技术专家对产品进行评估。
    - **走查**：开发者向同行展示代码并讨论。
    - **检查**：结构化的评审过程，通常由专门的检查员进行。
    - **审计**：正式的、独立的评估过程。
    - **桌面评审**：个人对产品进行初步检查。
  - **形式证明**：使用数学方法证明程序的正确性。
  - **追踪**：确保产品与需求和设计保持一致。

阅读技术

- **定义**：阅读技术是帮助检测软件产品缺陷的指南，包括一系列步骤或程序，使读者能够深入了解所读的产品。
- **目标**：缺陷检测。
- **类型：**
  - **随意阅读**：不提供任何指示或指导，读者自行系统地审查产品。
  - **基于检查表的阅读**：通过问题列表引导读者，但问题通常过于通用，不够特定。
  - **逐步抽象**：通过将程序规范与程序实际行为进行比较来检测缺陷。
  - **其他**：其他特定的阅读技术。

总结

- **静态评估的重要性**：静态评估通过早期检测缺陷，降低后续阶段的成本，并提供产品质量的估计。
- **缺陷类型**：包括正确性、完整性、冗余性、一致性、模糊性和可追溯性。
- **静态技术**：包括评审、形式证明和追踪，帮助确保产品符合需求和设计。
- **阅读技术**：帮助检测软件产品中的缺陷，包括随意阅读、基于检查表的阅读和逐步抽象。

二、软件测试简介

软件测试的角色

- **质量控制和保证活动**：软件测试是确保软件质量的重要环节，包括预防和纠正软件缺陷。
- **静态分析与动态分析**：
    - **静态分析**：不执行软件的情况下对软件进行检查，适用于任何软件产品，主要检查静态方面。
    - **动态分析（测试）**：通过执行软件来检查结果，主要关注动态方面，通常针对代码。

成熟度级别

- **Level 0**：测试等同于调试。
- **Level 1**：证明软件可以工作。
- **Level 2**：证明软件不能工作。
- **Level 3**：降低软件不工作的风险。
- **Level 4**：测试是一项智力学科，旨在降低软件风险。

测试定义

- **过程**：运行程序或系统以发现缺陷。
- **活动**：评估程序或系统的属性或能力，确定是否达到预期结果。
- **信息收集**：收集信息以高效评估工作。

测试原则

* **测试是为了发现软件缺陷**：
- **好的测试用例**：能够发现未检测到的缺陷。
- **预期结果**：需要明确定义。
- **详细检查**：每个测试用例都需要详细检查。
- **有效输入**：测试用例应涵盖有效和无效、预期和非预期的输入条件。
- **功能性测试**：确保软件做它应该做的事情，不做它不应该做的事情。
- **独立测试**：程序员不应测试自己的程序，团队也不应测试自己的程序。
- **保留测试用例**：不要丢弃测试用例。
- **计划测试努力**：不应假设不会发现缺陷。
- **缺陷集中**：软件的一部分发现的缺陷越多，该部分可能存在更多缺陷。

- **全面测试不可能**：完全测试是不可能的。
- **预防缺陷**：测试的一个目的是预防缺陷的发生。
- **基于风险**：测试是基于风险的。
- **智力活动**：测试是一项极具挑战性、创造性和智力性的活动。

测试过程

1. **生成和指定测试用例**：
    - **测试用例ID**：测试用例的唯一标识。
    - **测试目标**：测试用例的高级描述，例如“测试未提供文件名时不创建文件”。
    - **上下文**：测试用例的背景信息。
    - **输入**：实际输入给程序的数据，例如“未提供文件名”。
    - **预期输出**：根据规范描述的程序输出，例如“程序显示错误消息：未提供输入文件”。
    - **观察输出**：程序运行时的实际输出，例如“程序显示错误消息：未提供文件”。
2. **运行测试用例**：执行测试用例。
3. **比较实际结果与预期结果**：对比实际结果和预期结果。
4. **识别系统故障**：发现系统中的故障。
5. **识别和修正故障原因**：找出并修复导致故障的原因（调试步骤）。

测试用例设计技术

- **测试用例设计技术**：介绍各种测试用例设计方法，帮助生成有效的测试用例。

第1章：自我评估测试

概述

- **软件测试的变化**：自本书首次出版以来，软件测试既变得更容易也变得更困难。编程语言、操作系统和硬件平台的多样化增加了测试的复杂性，但现代软件和操作系统的成熟度也提供了一些现成的、经过测试的功能模块，减少了程序员的工作量。

软件测试的重要性

- **影响广泛**：现代软件系统影响着更多的人和企业，因此软件的质量和可靠性变得更加重要。
- **功能模块的复用**：现代开发语言提供了预编程的对象，这些对象已经经过调试和测试，减少了对自定义代码的测试需求。

第2章：程序测试的心理和经济学

测试原则

1. **定义预期输出**：测试用例必须包含预期的输出或结果的定义。
2. **避免自我测试**：程序员应避免测试自己的程序。
3. **避免团队自测**：编程组织不应测试自己的程序。
4. **彻底检查测试结果**：每次测试的结果都应仔细检查。
5. **测试无效和意外输入**：测试用例应包括无效和意外的输入条件，而不仅仅是有效和预期的输入条件。
6. **检查不期望的行为**：不仅要检查程序是否做了它应该做的，还要检查它是否做了它不应该做的。
7. **保留测试用例**：除非程序本身是临时的，否则不应丢弃测试用例。
8. **避免假设无错误**：不应在假设不会发现错误的前提下计划测试工作。
9. **缺陷的聚集性**：程序的一部分发现的错误越多，该部分存在更多错误的可能性越大。
10. **测试的创造性**：测试是一项极具创意和智力挑战的任务，可能需要比设计程序更高的创造力。

黑盒测试

- **定义**：黑盒测试（也称为数据驱动或输入/输出驱动测试）将程序视为黑盒，测试者不关心程序的内部结构和行为，而是专注于找出程序不符合规范的情况。
- **测试数据**：测试数据仅根据规范生成，不利用对程序内部结构的知识。
- **穷尽输入测试**：理论上，为了确保程序正确处理所有情况，需要测试所有可能的输入组合。但实际上，这通常是不可行的，因为输入组合的数量可能非常庞大。

白盒测试

- **定义**：白盒测试（也称为逻辑驱动测试）允许测试者检查程序的内部结构。测试数据是从对程序逻辑的检查中得出的。
- **穷尽路径测试**：理论上的目标是通过测试用例执行程序中的所有可能路径。但实际上，这通常是不可行的，因为路径数量可能极其庞大。
- **路径测试的局限性**：
    - **路径数量巨大**：即使是简单的程序也可能有天文数字的路径。
    - **路径测试不保证正确性**：即使所有路径都被测试过，程序仍可能因未包含必要的路径或逻辑错误而存在错误。

测试的心理学问题

- **目标导向**：测试的目标应该是发现错误，而不是证明程序没有错误。心理研究表明，当人们试图证明某事时，他们往往会忽略不利于该结论的证据。
- **独立测试的重要性**：程序员很难客观地测试自己的程序，因为他们可能不愿意发现错误，而且可能对程序的误解也会带入测试中。
- **测试的破坏性**：测试本质上是一个破坏性的过程，类似于医生诊断疾病，目的是发现并修复问题。

总结

- **测试的重要性**：软件测试是确保软件质量的关键环节，通过发现和修复错误来提高软件的可靠性和性能。
- **测试的原则**：遵循一系列测试原则，如定义预期输出、避免自我测试、彻底检查测试结果、测试无效和意外输入、保留测试用例等，可以提高测试的有效性和效率。
- **测试方法**：黑盒测试和白盒测试各有优缺点，结合使用可以更全面地测试软件。

白盒测试简介

测试目标

- **发现缺陷**：提高感知质量。
- **评估软件质量**：根据发现的缺陷数量评估软件的整体质量。

目标能否实现

- **穷尽测试的不可能性**：穷尽测试所有可能的输入或路径是不可行的，因为输入和路径的数量可能极其庞大。
- **选择输入集**：当前方法是通过从输入域中选择一些输入来进行估算，但选择的输入会影响估算的质量，可能导致偏差。

测试用例设计技术

技术分类

- **分类标准**：
    - **充分性准则**：测试用例的选择标准。
    - **方法**：基于实现知识的方法（白盒）、基于故障的方法（变异）、基于错误的方法（功能随机）。

控制流技术

- **定义**：将程序视为白盒，根据程序的控制结构生成测试用例。

- **覆盖率概念：**
    - **语句覆盖率：**每条语句是否被执行。
    - **分支覆盖率：**每个控制结构的每个分支是否被执行。
    - **条件覆盖率：**每个布尔子表达式是否都评估为真和假。
    - **决策/条件覆盖率：**每个布尔子表达式和每个分支是否都被覆盖。
    - **多重条件覆盖率：**所有布尔子表达式的组合是否都被评估。

基本路径测试

- **路径定义：**从程序输入到输出的一系列语句。
- **策略：**
    - **分析控制流图：**找出一组线性独立的执行路径。
    - **生成测试用例：**为每条路径生成测试用例。
- **基于 McCabe 循环复杂度：**
    - **计算循环复杂度：**确定线性独立路径的数量。
    - **保证：**完全分支覆盖，但不必覆盖所有可能的路径。

数据流技术

- **定义：**将程序视为白盒，基于变量值的赋值和赋值对执行的影响生成测试用例。
- **变量发生：**
    - **定义（def）：**与节点关联，表示节点中包含全局定义的变量。
    - **计算使用（c-use）：**与节点关联，表示节点中包含全局计算使用的变量。
    - **谓词使用（p-use）：**与边关联，表示边中包含谓词使用的变量。
- **定义-使用对：**
    - **dcu(x, i)：**从节点 i 到节点 j 的定义自由路径上的所有节点 j，其中 x 是节点 j 中的计算使用变量。
    - **dpu(x, i)：**从节点 i 到节点 j 的定义自由路径上的所有边 (j, k)，其中 x 是边 (j, k) 中的谓词使用变量。

变异技术

- **定义：**将程序视为白盒，根据程序中可能的故障生成变异体。
- **变异算子：**使用与编程语言相关的变异算子生成变异体。
- **生成测试用例：**从变异体生成测试用例，生成的测试用例数量应足以杀死所有变异体。

- **技术变体**：
    - **强变异**：使用所有变异算子，目标是达到 100% 覆盖。
    - **弱变异**：放宽覆盖条件。
    - **选择变异**：仅使用部分变异算子。

测试用例生成

等价类划分

- **输入条件**：每个输入条件被划分为多个组，定义有效和无效的等价类。
- **生成测试用例**：
    - **覆盖尽可能多的有效等价类**：生成测试用例，直到覆盖所有有效等价类。
    - **覆盖无效等价类**：生成测试用例，直到覆盖所有无效等价类。

边界值分析

- **值范围**：
    - **设计测试用例**：覆盖范围的两个边界值以及刚好超出这两个边界值的情况。
- **值数量**：
    - **设计测试用例**：覆盖最小值和最大值，以及刚好超出最大值和小于最小值的情况。
- **有序集合**：
    - **关注集合的第一个和最后一个元素**。
- **输出数据**：将上述规则应用于输出数据。

测试用例说明

- **测试用例 ID**：测试用例的唯一标识。
- **测试目标**：测试用例的高级描述，例如“测试未提供文件名时不创建文件”。
- **上下文**：测试用例的背景信息。
- **输入**：实际输入给程序的数据，例如“未提供文件名”。
- **预期输出**：根据规范描述的程序输出，例如“程序显示错误消息：未提供输入文件”。
- **观察输出**：程序运行时的实际输出，例如“程序显示错误消息：未提供文件”。

总结

- **白盒测试的重要性**：通过检查程序的内部结构和逻辑，生成测试用例，确保程序的每个部分都得到充分测试。

- **测试用例设计技术**：包括控制流技术、数据流技术和变异技术，这些技术帮助生成有效的测试用例。
- **测试用例生成**：通过等价类划分和边界值分析，确保测试用例覆盖所有可能的输入和输出情况。

第2章：图覆盖

图覆盖的基础

- **图覆盖的目标**：通过图的抽象来生成测试用例，确保测试用例能够覆盖图中的各个部分。
- **图的定义**：图 (G) 由节点集合 (N)、初始节点集合 (N_0)、最终节点集合 (N_f) 和边集合 (E) 组成。

常见的图覆盖准则

- **节点覆盖**：确保每个节点都被访问。
- **边覆盖**：确保每条边都被执行。
- **条件覆盖**：确保每个布尔子表达式都被评估为真和假。
- **决策/条件覆盖**：确保每个布尔子表达式和每个分支都被覆盖。
- **多重条件覆盖**：确保所有布尔子表达式的组合都被评估。

图的应用

- **有限状态机（FSM）**：早期的研究集中在使用 FSM 生成测试用例，尤其是在电信系统中。
- **控制流图**：最常见的图抽象形式，将代码映射到控制流图。
- **图生成方法**：包括生成生成树、路径组合等方法。

第3章：逻辑覆盖

逻辑表达式的正式化

- **谓词**：评估为布尔值的表达式，包含布尔变量、非布尔变量的比较操作符和函数调用。
- **逻辑运算符**：
    - **否定运算符（¬）**
    - **与运算符（∧）**
    - **或运算符（∨）**
    - **蕴含运算符（→）**
    - **异或运算符（⊕）**

- **等价运算符（↔）**

常见的逻辑覆盖准则

- **谓词覆盖**：确保每个谓词的所有可能值都被评估。
- **条件覆盖**：确保每个布尔子表达式都被评估为真和假。
- **决策覆盖**：确保每个布尔子表达式和每个分支都被覆盖。
- **多重条件覆盖**：确保所有布尔子表达式的组合都被评估。

逻辑覆盖的局限性

- **路径组合爆炸**：复杂的逻辑表达式可能导致路径组合的数量急剧增加，使得测试用例生成变得不现实。
- **代码简化**：有时可以通过简化代码结构来减少路径组合的数量，例如合并重复的代码块。

第4章：输入空间划分

输入空间划分的组合策略

- **全组合（ACoC）**：覆盖所有输入值的组合。
- **成对组合（PWC）**：覆盖所有输入值的成对组合。
- **多基选择（MBCC）**：覆盖多个基选择的组合。

分区的属性

- **完整性**：每个输入值都属于某个分区。
- **互斥性**：每个输入值只属于一个分区。

分区的选择

- **有效值**：至少包含一组有效值。
- **子分区**：将有效值范围划分为更小的子分区。
- **边界值**：接近边界值的输入往往会导致问题。
- **正常使用**：如果操作配置主要集中在“正常使用”上，故障率取决于非边界条件的值。
- **无效值**：至少包含一组无效值。

总结

- **图覆盖**：通过图的抽象生成测试用例，确保测试用例能够覆盖图中的各个部分。常见的图覆盖准则包括节点覆盖、边覆盖、条件覆盖、决策/条件覆盖和多重条件覆盖。
- **逻辑覆盖**：通过逻辑表达式的评估生成测试用例，确保测试用例能够覆盖逻辑表达式的各个部分。常见的逻辑覆盖准则包括谓词覆盖、条件覆盖、决策覆盖和多重条件覆盖。

- **输入空间划分**：通过输入值的划分生成测试用例，确保测试用例能够覆盖所有可能的输入值组合。常见的输入空间划分策略包括全组合、成对组合和多基选择。

第4章：测试用例设计

概述

- **测试用例设计的重要性**：程序测试中最关键的部分是设计和创建有效的测试用例。
- **测试的局限性**：无论测试多么有创意和全面，都不能保证发现所有错误。
- **测试的必要性**：由于完全测试是不可能的，测试必须尽可能全面。

测试用例设计方法

- **黑盒测试方法**：
    - **等价划分**：将输入域划分为几个等价类，选择每个等价类中的代表性值进行测试。
    - **边界值分析**：测试输入域的边界值和刚好超出边界值的情况。
    - **因果图**：将输入条件和输出结果的关系绘制成图，生成测试用例。
    - **错误猜测**：基于经验和直觉，猜测可能存在的错误并设计测试用例。
- **白盒测试方法**：
    - **语句覆盖**：确保每个语句至少被执行一次。
    - **分支覆盖**：确保每个分支至少被执行一次。
    - **条件覆盖**：确保每个布尔子表达式至少被评估为真和假。
    - **决策/条件覆盖**：确保每个布尔子表达式和每个分支至少被覆盖一次。
    - **多重条件覆盖**：确保所有布尔子表达式的组合至少被评估一次。

测试用例设计的综合方法

- **综合方法**：建议结合使用黑盒和白盒测试方法，以设计更全面的测试用例。
    - **黑盒测试**：主要用于功能测试，确保程序满足需求规格。
    - **白盒测试**：主要用于逻辑测试，确保程序的内部结构和逻辑正确。

测试用例设计的具体方法

1. **等价划分**：
    - **定义等价类**：将输入域划分为几个等价类，每个等价类中的值在测试中被视为等效。
    - **选择测试用例**：从每个等价类中选择一个或多个代表性值作为测试用例。

2. **边界值分析：**
    - **选择边界值：**测试输入域的边界值和刚好超出边界值的情况。
    - **值的数量：**测试最小值和最大值，以及刚好超出最大值和小于最小值的情况。
    - **有序集合：**关注集合的第一个和最后一个元素。
3. **因果图：**
    - **定义因果关系：**将输入条件和输出结果的关系绘制成图。
    - **生成测试用例：**从因果图中生成测试用例，确保每个因果关系都被测试。
4. **错误猜测：**
    - **基于经验和直觉：**猜测可能存在的错误，并设计测试用例来验证这些错误。
5. **语句覆盖：**
    - **确保每个语句：**确保每个语句至少被执行一次。
6. **分支覆盖：**
    - **确保每个分支：**确保每个分支至少被执行一次。
7. **条件覆盖：**
    - **确保每个布尔子表达式：**确保每个布尔子表达式至少被评估为真和假。
8. **决策/条件覆盖：**
    - **确保每个布尔子表达式和分支：**确保每个布尔子表达式和每个分支至少被覆盖一次。
9. **多重条件覆盖：**
    - **确保所有布尔子表达式的组合：**确保所有布尔子表达式的组合至少被评估一次。

测试用例设计的注意事项

- **随机输入测试：**随机选择输入值进行测试是最不有效的方法，因为它可能无法发现大多数错误。
- **测试用例的选择：**选择测试用例时，应考虑输入值的有效性和无效性，以及边界值和特殊情况。
- **测试用例的保留：**测试用例不应被丢弃，除非程序本身是临时的。

总结

- **测试用例设计的重要性：**设计和创建有效的测试用例是确保软件质量的关键。
- **测试用例设计方法：**结合使用黑盒和白盒测试方法，可以更全面地测试软件。

- **具体方法：**等价划分、边界值分析、因果图、错误猜测、语句覆盖、分支覆盖、条件覆盖、决策/条件覆盖和多重条件覆盖等方法，可以帮助生成有效的测试用例。