第3章 静态测试


思想引领

3.1 评　　审

3.1.1 评审作用

评审有以下几项作用。

(1) 提高质量。

类似于动态测试,发现缺陷也是评审的最主要目的之一。尽早发现软件工作产品中的缺陷,通过修复缺陷直接提高软件产品的质量;同时尽早发现和修复缺陷,也可以减少将缺陷带入下个阶段的机会,间接地提高质量。

(2) 降低成本。

缺陷发现和修复的成本随着开发阶段的演进而上升,因此尽早发现和修复缺陷可以直接降低成本;减少缺陷的雪崩效应也可以间接地降低成本。

(3) 加快进度。

在开发阶段的后期发现缺陷,不仅发现缺陷的难度增加,其发现的效率也将降低;同时对开发团队而言,其定位和修复缺陷的难度也将增加,从而需要花费更多的时间,导致时间进度的延后。

(4) 提升能力。

参与评审活动,对每个评审员而言都相当于参加了一次培训,有助于在将来的项目中输出质量更高的工作产品。通过评审员之间的分析和讨论,除了项目技术相关的知识和技能,大家还在评审过程、规则和实践等方面实现了共享。

3.1.2 评审基本原则

评审有以下一些基本原则。

(1) 评审对象是产品,讨论只针对事,不针对人。评审会只考虑问题是否是什么现象,不涉及负责人。

(2) 注重评审效率。

(3) 某阶段未通过阶段评审不得进入下一个软件研制阶段。

(4) 评审就要挑刺,找问题、缺陷和隐患。

(5) 评审组人员面越广越好。

(6) 评审组不做无休止的争论和辩驳,将争论点记录下来,供以后甄别。

(7) 评审只是提出问题,没有解决问题的任务。

(8) 使用“评审检查单”提高评审的效果。

3.1.3 评审基本过程

评审的基本过程如下。

(1) 组建评审组。

(2) 评审组长负责主持和控制全部评审活动。

(3) 评审计划。

(4) 评审准备。

(5) 评审会。

(6) 提交评审报告。

(7) 建立评审过程。

评审过程如图 3-1 所示。

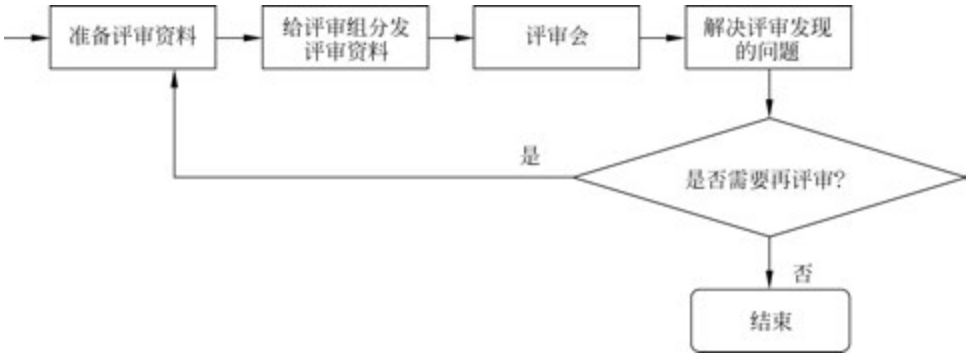

图 3-1 评审过程

3.1.4 角色和职责

1. 参与评审会的角色

(1) 主审员：协调本次审查并主持讨论。

(2) 责任人：负责被审查的产品。

(3) 讲解员：在审查会上讲解被审查的产品。

(4) 审查员：审查产品。

(5) 记录员：记录在审查会议上讨论的问题。

(6) 产品经理：责任人的管理者。

2. 产品经理的具体职责

(1) 帮助决定审查的内容。

(2) 将审查工作纳入项目计划。

(3) 分配审查资源。

(4) 保障审查培训工作。

(5) 参与主审员的选定工作。

(6) 支持主审员完成所要求的任何修改工作。

3. 主审员

(1) 主审员是审查过程成败的关键,应该具备主审员的素质、经验,具有专业技能和管理技能。

(2) 主审员一般由产品经理和责任人选择,能够负责:

① 了解正在审查的信息。

② 领导审查组展开有效的讨论。

③ 调解争端。

④ 分辨主要问题引导审查组侧重这些问题。

⑤ 客观性地提出意见。

⑥ 适当地赋予职责。

4. 主审员的具体职责

(1) 选定审查组成员。

(2) 确保审查组成员用于审查的时间。

(3) 确保产品查经理了解审查工作。

(4) 制订审查会计划,安排资料、后勤的准备。

(5) 审查会前验收审查准备情况。

(6) 确保审查会高效、有序进行。

(7) 确保审查会上确定的问题文档化。

(8) 问题追踪直到解决。

(9) 审查会后完成会议记录和审查报告。

5. 责任人的具体职责

责任人负责准备要审查的信息或工作产品,具体职责包括:

(1) 确保要审查的工作产品已就绪。

(2) 按时提供审查所需要的信息。

(3) 帮助主审员做好会议安排、资料准备、问题改正进度安排。

(4) 及时解决审查组确定的所有问题。

(5) 坚持客观性,避免辩解。

(6) 在审查会上阐明审查员不清楚的问题。

6. 讲解员的具体职责

讲解员负责对被审的工作产品进行释义,具体职责包括:

(1) 完全熟悉正在审查的工作产品。

(2) 确定信息的逻辑块并能解释每一个信息。

(3) 支持主审员工作。

7. 审查员的具体职责

审查员负责寻找工作产品与所依据的文档或标准之间的差异,确定存在的问题,具体职责包括以下内容。

(1) 完全熟悉要审查的工作产品。

(2) 完全熟悉审查依据的文档和标准。

(3) 鉴别工作产品中存在的问题。

(4) 保持客观性。

(5) 对产品而不是责任人提出批评。

(6) 支持主审员工作。

8. 记录员的具体职责

记录员负责在审查会上记录审查组确定的问题及其说明,具体职责包括以下内容。

(1) 完全熟悉要审查的工作产品。

(2) 记录审查组提出的所有问题。

(3) 提供主审员要求的其他补充信息。

(4) 支持主审员工作。

3.2 评 审 类 型

3.2.1 需求评审

1. 需求评审概述

软件需求是软件开发最重要的一个步骤,需求的质量很大程度上决定了项目质量或产品质量。需求风险也常常是软件开发过程中最大的一个风险,降低需求风险的一个重要手段就是需求评审。但是,需求评审是所有评审活动中最难也最容易被忽视的一个评审。

在需求评审中经常存在以下问题。

(1) 需求报告很长,短时间内评审者根本不能把需求报告读懂、想清楚。

(2) 没有做好前期准备工作,需求评审的效率很低。

(3) 需求评审的节奏无法控制。

(4) 找不到合格的评审员,与会的评审员无法提出深入的问题。

2. 如何做好需求评审

为了做好需求评审,采用如下建议可以收到较好效果。

(1) 分层次评审。

用户的需求是可以分层次的,一般而言可以分成如下的层次。

① 目标性需求:定义整个系统需要达到的目标。

② 功能性需求:定义整个系统必须完成的任务。

③ 操作性需求:定义完成每个任务的具体的人机交互。

目标性需求是企业高层管理人员所关注的,功能性需求是企业中层管理人员所关注的,操作性需求是企业的具体操作人员所关注的。对不同层次的需求,其描述形式是有区别的,参与评审的人员也是不同的。如果让具体的操作人员去评审目标性需求,可能会很容易地导致“捡了芝麻,丢了西瓜”的现象;如果让高层管理人员也去评审那些操作性需求,无疑是一种资源的浪费。

(2) 正式评审与非正式评审结合。

正式评审是指通过开评审会的形式,组织多个专家,将需求涉及的人员集合在一起,并定义好参与评审人员的角色和职责,对需求进行正规的会议评审。而非正式的评审并没有这种严格的组织形式,一般也不需要将人员集合在一起评审,而是通过电子邮件、文件汇签和网络会议等多种形式对需求进行评审。这两种形式各有利弊,但非正式的评审往往比正

式的评审效率更高，更容易发现问题。因此，在评审时应该灵活地利用这两种方式。

(3) 分阶段评审。

应该在需求形成的过程中进行分阶段评审，而不是在需求最终形成后再进行评审。分阶段评审可以将原本需要进行的大规模评审拆分成各个小规模的评审，降低了需求返工的风险，提高了评审的质量。例如：可以在形成目标性需求后进行一次评审，在形成系统的初次概要需求后进行一次评审，应当对概要需求细分成几个部分，对每个部分进行各个评审，最终再对整体的需求进行评审。

(4) 精心挑选评审员。

需求评审可能涉及的人员包括：需方的高层管理人员、中层管理人员、具体操作人员、IT 主管、采购主管；供方的市场人员、需求分析人员、设计人员、测试人员、质量保证人员、实施人员、项目经理以及第三方的领域专家等。由于人员中大家所处的立场不同，对同一个问题的看法是不相同的，有些观点和系统的目标有关系，有些则关系不大，不同的观点可能形成互补的关系。为了保证评审的质量和效率，需要精心挑选评审员。首先，要保证不同类型的人员都参与进来，否则很可能会漏掉很重要的需求。其次，在不同类型的人员中要选择那些真正和系统相关的、对系统有足够了解的人员参与进来，否则很可能使评审的效率降低或者最终不切实际地修改了系统的范围。

(5) 对评审员进行培训。

很多情况下，评审员是领域专家而不是进行评审活动的专家，他们没有掌握进行评审的方法、技巧、过程等，因此需要对评审员进行培训。同样，对于主持评审的管理者也需要进行培训，以便于参与评审的人员能够紧紧围绕评审的目标来进行，这样能够控制评审活动的节奏，提高评审效率。对评审员的培训可以区分为简单培训和详细培训两种。简单培训可能需要十几分钟或者几十分钟，可在评审过程中把需要把握的基本原则、需要注意的常见问题说清楚。详细培训则可能需要对评审的方法、技巧、过程进行正式的培训，需要花费较长的时间，是一个独立的活动。

(6) 充分利用需求评审检查单。

需求检查单是很好的评审工具，需求检查单可以分成两类：需求形式的检查单和需求内容的检查单。需求形式的检查可以由质量保证人员负责，主要是针对需求文档的格式是否符合质量标准提出的。需求内容的检查由评审员负责，主要检查需求内容是否达到了系统目标、是否有遗漏、是否有错误等，这是需求评审的重点。检查单可以帮助评审员系统全面地发现需求中的问题，它也是随着工程经验的积累逐渐丰富和优化的。

(7) 建立标准的评审流程。

正规的需求评审会需要建立正规的需求评审流程，按照流程中定义的活动进行规范的评审过程。例如：在评审流程定义中可能规定评审的进入条件、评审需要提交的资料、每次评审会议的人员职责分配、评审的具体步骤、评审通过的条件等。

(8) 做好评审后的跟踪工作。

需求评审后，需要根据评审人员提出的问题进行评估，以确定哪些问题是必须纠正的，哪些可以不纠正，并给出充分的客观理由与证据。当确定需要纠正的问题后，要形成书面的需求变更申请，进入需求变更管理流程，并确保变更的执行。在变更完成后，要进行复审。切忌评审完毕后，没有对问题进行跟踪，而无法保证评审结果的落实，使前期的评审努力付

诸东流。

(9) 充分准备评审。

评审质量的好坏很大程度上取决于在评审会议前的准备活动。经常出现的问题是,需求文档在评审会议前并没有提前分发给参与评审会议的人员,没有留出更多更充分的时间让参与评审的人员阅读需求文档。更有甚者,没有执行需求评审的进入条件,在评审文档中存在大量的低级错误或者没有在评审前进行沟通,或者文档中存在方向性的错误,从而导致评审的效率很低,质量很差。对评审的准备工作,也应当定义一个检查单,在评审之前对照检查单落实每项准备工作。

3. 软件需求规格说明的评审细则

对需求规格说明的评审,可遵循如下评审细则。

(1) 是否清晰地定义了引用文档?

(2) 是否以软件配置项为单位分别对各个软件配置项进行了需求分析?

(3) 是否对每个软件配置项的外部接口进行了清晰、完整的说明?

(4) 是否对每个软件配置项所包含的功能及其性能进行了适当的了解?

(5) 是否对每个软件功能的输入、输出进行了细致的说明?

(6) 是否对每个软件功能所对应的处理模型或处理流程(还包括容错处理模型、异常处理模型等)进行了翔实的说明?

(7) 对于安全关键软件,是否清晰地表示了软件必须处理的安全关键事件或危险事件?

(8) 软件配置项的功能是否满足软件研制任务书的要求或系统设计文档的要求?

(9) 软件需求分析的方法、使用的工具是否得当?

(10) 是否明确了对软件的非功能性需求?

(11) 是否明确提出了软件的安全性、可靠性需求?

(12) 是否明确提出了软件的易用性需求?

(13) 是否明确提出了软件的维护性需求?

(14) 是否明确提出了软件的可移植性需求?

(15) 是否明确了对软件的数据保密性、完整性需求?

(16) 软件需求是否是可测试、可度量和可验证的?

(17) 是否具有需求追踪表,向上可追溯到“软件研制任务书”或“系统/子系统设计文档”?

(18) 所有需求是否都进行了明确定义?用例图是否覆盖了主要的用户功能需求?

(19) 用例图是否清楚地说明了功能、角色、主事件流、异常事件流、前置条件、后置条件和非功能需求等内容?

(20) 对于业务流程,是否有详细的描述,包括处理机制、算法等?

(21) 文档编写是否规范?

(22) 文档描述是否正确、完整、一致?

3.2.2 概要设计评审

1. 概要设计评审概述

软件概要设计结束后必须进行概要设计评审,以评价软件设计说明书中所描述的软件概要设计在总体结构、外部接口、主要部件功能分配、全局数据结构以及各主要部件之间的

接口等方面的合适性。一般应检查以下几个方面。

(1) 概要设计说明书是否与软件需求说明书的要求一致?

(2) 概要设计说明书是否正确、完整、一致?

(3) 系统的模块划分是否合理?

(4) 接口定义是否明确?

(5) 文档是否符合有关标准规定?

2. 概要设计说明的评审细则

对软件概要设计说明的评审,可遵循如下评审细则。

(1) 是否做到了以软件部件为基础进行软件体系结构的设计,即体系结构中的组成部分必须为实体部件?

(2) 软件体系结构是否优化、合理、稳健?

(3) 是否将软件需求规格说明中定义的功能、性能等全部都分配了具体的软件部件?

(4) 为各个软件部件分配的功能、性能是否合理?

(5) 是否清晰、合理地定义了各个软件部件的接口?

(6) 软件部件的扇入、扇出数是否符合要求(一般应控制在 7 以下)?

(7) 对于安全关键功能,是否采用了必要的设计策略?

(8) 是否说明了软件可靠性设计的具体措施?

(9) 是否清晰、合理地设计了软件部件之间的协同关系(如同步、互斥等)?

(10) 是否清晰、完整地列出了所有要求监督的软件工作过程,如软件需求分析、软件设计、配置项测试、配置管理等?

(11) 是否具体地策划了软件质量监督的监督节点?

(12) 是否建立了软件设计与软件需求的追踪表,审查分配给每个 CSC(计算机软件部件)的功能或任务是否可追溯到“软件需求规格说明”或“接口需求和设计文档”?

(13) 逻辑视图对 CSCI(计算机软件配置项)的层次分解是否合理? 分解到最底层的包或类的粒度是否合适? 即该包/类的后续实现者是否不必了解全系统的情况? 中途换人是否不影响工作进度?

(14) 分解的包/类是否涵盖了所有的功能需求?

(15) 主要用例的主流程是否用分解的设计元素进行描述?

(16) 是否为完成需求的功能增加了必要的包/类,使得层次分解的结果是一个完整的设计?

(17) 进程视图是否描述了所有主要的进程/线程,进程的生命期、功能、同步、通信等机制描述是否完备?

(18) 实现视图是否描述了 CSCI 的实现组成,每个组件是否分配了合适的需求功能,组件的表现形式(如 exe、dll 等)是否合理?

(19) 部署视图是否描述了 CSCI 的安装运行情况,能否对未来的运行景象形成明确概念?

(20) 文档编写是否规范?

(21) 文档描述是否正确、一致、完整?

3.2.3 详细设计评审

1. 详细设计评审概述

软件详细设计阶段结束后必须进行详细设计评审，以评价软件验证与确认计划中所规定的验证与确认方法的合适性与完整性。一般应检查以下几个方面。

(1) 详细设计说明书是否与概要设计说明书的要求一致？

(2) 模块内部逻辑结构是否合理，模块之间的接口是否清晰？

(3) 数据库设计说明书是否完全？是否正确反映详细设计说明书的要求？

(4) 测试是否全面、合理？

(5) 文档是否符合有关标准规定？

2. 详细设计说明的评审细则

对软件详细设计说明的评审，可遵循如下评审细则。

(1) 是否将软件部件分解为软件单元？

(2) 是否对每个软件单元规定了程序设计语言所对应的处理流程？

(3) 对每个单元的入口、出口是否给予了清晰完整的设计？

(4) 软件单元之间的关系是否清晰完整？

(5) 文档编写是否规范？

(6) 文档描述是否正确、一致、完整？

3.2.4 数据库评审

1. 数据库设计评审概述

数据库设计阶段结束后必须进行数据库设计评审，以评价数据库的结构设计及运用设计的合适性。一般应检查以下几个方面。

(1) 概念结构设计。

(2) 逻辑结构设计。

(3) 物理结构设计。

(4) 数据字典设计。

(5) 安全保密设计。

2. 数据库设计说明的评审细则

对数据库设计说明的评审，可遵循如下评审细则。

(1) 是否进行了数据库系统模式设计、子模式设计以及物理设计？

(2) 数据的逻辑结构是否满足完备性要求？

(3) 数据的逻辑结构是否满足一致性要求？

(4) 数据的冗余度是否合理？

(5) 数据库的后备、恢复设计是否合理、有效？

(6) 对数据的存取控制是否满足数据的安全保密性要求？

(7) 对数据的存取控制是否满足实时性要求？

(8) 网络、通信设计是否合理、有效？

(9) 审计、控制设计是否合理？

(10) 屏幕设计、报表设计是否满足要求？

(11) 文件的组织方式和存取方法是否合理、有效?

(12) 数据安排是否合理、有效?

(13) 数据在存储介质上的分配是否合理、有效?

(14) 数据的压缩、分块是否合理、有效?

(15) 缓冲区的大小和管理是否满足要求?

(16) 文档编写是否规范?

(17) 文档描述是否正确、一致、完整?

3.2.5 测试评审

1. 软件测试需求规格说明的评审细则

对软件测试需求规格说明的评审,可遵循如下评审细则。

(1) 测试依据是否完整、有效?

(2) 是否标识了所有被测对象?

(3) 是否提出了对被测对象的评价方法?

(4) 是否对被测对象的测试内容进行了适当的分类,即标识了测试类型?

(5) 对于每个测试项,是否提出了测试的充分性要求?

(6) 对于每个测试项,是否清晰地给出了追踪关系?

(7) 是否明确提出了测试项目的终止条件?

(8) 是否对软件单元提出了圈复杂度的测试要求?

(9) 是否对软件单元提出了对源代码的注释行进行分析、检查和统计的测试要求(如代码的有效注释行比率不低于 20%)?

(10) 是否对软件单元提出了语句覆盖和分支覆盖的测试项要求(如语句覆盖和分支覆盖应达到 100%)?

(11) 是否对软件单元的每个特征都提出了测试要求(至少包括正常激励的测试要求和被认可的异常激励要求)?

(12) 是否提出了单元调用关系覆盖测试的要求?

(13) 测试内容(测试项)是否覆盖了每个部件的所有外部接口?

(14) 是否按照设计要求,对部件的功能、性能提出了强度测试要求?

(15) 对于安全关键部件,是否提出了安全性分析和安全性测试要求?

(16) 测试内容(测试项)是否覆盖了配置项的所有功能和性能?

(17) 测试内容(测试项)是否覆盖了配置项的所有外部接口?

(18) 是否按照软件需求规格说明的要求,对软件配置项的功能、性能提出了强度测试要求?

(19) 对于安全关键的配置项,是否提出了安全性分析和安全性测试要求?

(20) 测试内容(测试项)是否涵盖了系统的所有功能和性能?

(21) 测试内容(测试项)是否涵盖了系统的所有外部接口?

(22) 是否按照系统设计文档的要求,对系统的功能、性能提出了强度测试要求?

(23) 对于安全关键的系统,是否提出了安全性分析和安全性测试要求?

(24) 文档描述是否正确、一致、完整?

(25) 文档编写是否规范?

2. 软件测试计划的评审细则

对软件测试计划的评审,可遵循如下评审细则。

(1) 是否明确了测试组织与成员,并为每个成员合理地分配了责任?

(2) 测试人员是否具有相对的独立性?

(3) 测试人员的资质是否符合测试项目的要求?

(4) 测试依据是否完整、有效?

(5) 是否标识了所有被测对象?

(6) 是否提出了对被测对象的评价方法?

(7) 是否对被测对象的测试内容进行了适当的分类,即标识了测试类型?

(8) 对于每个测试项,是否提出了测试的充分性要求?

(9) 对于每个测试项,是否提出了测试的终止条件?

(10) 对于每个测试项,是否清晰地给出了追踪关系? 具体地,单元测试计划应追踪到详细设计文档,部件测试计划应追踪到设计文档或概要设计文档,配置项测试计划应追踪到软件需求规格说明,系统测试计划应追踪到系统设计说明。

(11) 是否明确提出了测试环境要求? 包括软件环境、硬件环境、测试工具等。

(12) 是否明确提出了测试项目的终止条件?

(13) 确定的测试进度是否合理、可行?

3. 软件测试说明的评审细则

对软件测试说明的评审,可遵循如下评审细则。

(1) 是否覆盖了测试计划中标识的所有被测试对象?

(2) 是否对测试项提出了测试方法要求?

(3) 是否对测试项进行了合理的测试进度安排?

(4) 是否对测试项进行了合理的测试过程准备?

(5) 测试用例的描述是否全面?

(6) 测试用例是否充分?

(7) 测试方法是否可行?

(8) 是否建立了清晰的测试用例与测试计划的追踪关系?

(9) 文档编写是否规范?

(10) 文档描述是否正确、一致、完整?

4. 软件测试报告的评审细则

对软件测试报告的评审,可遵循如下评审细则。

(1) 是否对测试过程进行了描述?

(2) 是否对测试用例的执行情况进行了描述?

(3) 是否对未执行的测试用例说明了未执行的原因?

(4) 测试报告结论是否客观?

(5) 文档编写是否规范?

(6) 文档描述是否正确、一致、完整?

5. 软件测试记录的评审细则

对软件测试记录的评审，可遵循如下评审细则。

(1) 是否有测试人员签名？

(2) 测试用例执行结果描述是否充分、明确？

(3) 测试用例执行过程描述是否充分、明确？

(4) 文档编写是否规范？

(5) 文档描述是否正确、一致、完整？

3.3 静 态 分 析

3.3.1 控制流分析

1. 控制流测试

由于非结构化的程序会给测试、排错和维护带来许多不必要的麻烦，所以人们有理由要求写出的程序是结构良好的。自 20 世纪 70 年代以来，结构化程序的概念逐渐被人们普遍接受。用于刻画程序结构的控制流图已有很长的历史。对用结构化程序语言书写的程序，则可以通过使用一系列规则从程序推导出其对应的控制流图。因此，控制流图和程序是一一对应的，而且控制流图更容易使人们理解程序。

定义 3-1 有向图 $G=(V,E)$，V 是顶点的集合，E 是有向边(本文简称边)的集合。$e=(T(e),H(e))\in E$ 是一对有序的邻接节点，$T(e)$是尾，$H(e)$是头。如果 $H(e)=T(e')$，则 e 和 e'是临界边。$H(e)$是 $T(e)$的后继节点，$T(e)$是 $H(e)$的前驱节点，indegree(n)和 outdegree(n)分别是节点 n 的入度和出度。

【例 3-1】 下面给出一段用 C 语言编写的计算最大公约数的程序。

```
1  void main()
2  {
3    int x,y;
4    scanf("%d%d",&x,&y);
5    while(x>0&&y>0) {
6      if(x>y)x=x-y;
7      else y=y-x;
8      printf("%d\n",x+y);
9    }
10 }
```

该程序的控制流图如图 3-2 所示。

将图 3-2 中的描述语句去掉，可以得到简化的控制流图，如图 3-3 所示。

定义 3-2 路径：如果 $P=e_1\ e_2\cdots e_q$，且满足 $T(e_i+1)=H(e_i)$，则称 P 为路径，q 为路径长度。

定义 3-3 完整路径：如果 P 是一条路径，且满足 $e_1=e_0$，$e_q=e_k$，e_0 是程序的源节点，e_k 是其汇节点，则 P 称为完整路径。如果存在输入数据使得程序按照该路径运行，这样的路径称为可行完整路径，否则称为不可行完整路径。

程序设计要尽量避免不可行完整路径，因为这样的路径往往隐含错误。

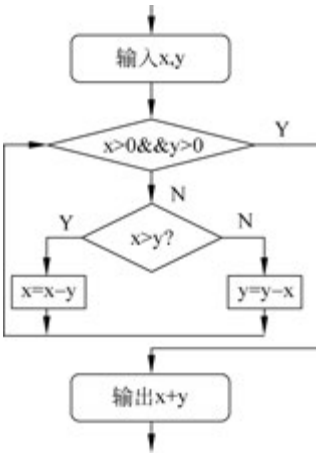

图 3-2　求最大公约数程序的控制流图

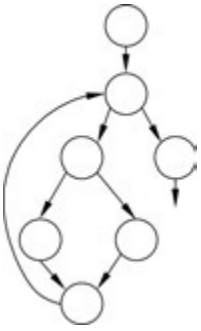

图 3-3　简化的控制流图

定义 3-4　可达：如果 e_i 到 e_j 存在一个路径，则称 e_i 到 e_j 是可达的。

定义 3-5　简单路径：路径上所有的节点都是不同的，称为简单路径。

定义 3-6　基本路径：任意有向边都在路径中最多出现一次的路径称为基本路径。

定义 3-7　子路径：路径 $A=e_u\ e_{u+1}\cdots e_t$ 是 $B==e_1\ e_2\cdots e_q$ 的子路径，如果满足 $1\leqslant u\leqslant t\leqslant q$。

定义 3-8　回路：路径 $P=e_u\ e_{u+1}\cdots e_q$ 满足 $T(e_u)=H(e_q)$，称为回路。除了第一个和最后一个节点外，其他节点都不同的回路称为简单回路。

定义 3-9　无回路路径：一条路径中不包含有回路子路径，称为无回路路径。

定义 3-10　A 连接 B：若 $A=e_u\ e_{u+1}\cdots e_t$，$B=e_v\ e_{v+1}\cdots e_q$ 为两条路径，如果 $T(e_t)=H(e_v)$ 且 $e_u\ e_{u+1}\cdots e_t e_v\ e_{v+1}\cdots e_q$ 为路径，则 A 连接 B，记为 $A*B$。

当一条路径是回路时，它可以和自己连接，即 $A^1=A$，$A^{k+1}=A*A^k$。

定义 3-11　路径 A 覆盖路径 B：如果路径 B 中所含的有向边均在路径 A 中出现，则称路径 A 覆盖路径 B。

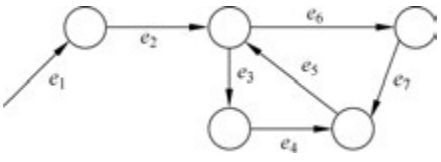

图 3-4　路径覆盖关系示例

注意：如果是没有回路的路径和子路径，则路径 A 覆盖路径 B，实际上 B 就是 A 的子路径。但如果存在回路，则情况未必如此。如图 3-4 所示，$A=e_1e_2e_3e_4e_5e_6e_7$ 是一个路径，$B=e_5e_3e_4$ 也是一个路径，显然 A 覆盖 B，但 B 并不是 A 的子路径。覆盖关系是一个比子路径的含义更广的概念。

2. 控制流覆盖准则

测试覆盖准则是指覆盖测试的标准。常见的测试覆盖准则包括以下几项。

(1) 语句覆盖准则。

语句覆盖测试是最简单的结构性测试方法之一。它要求在测试中，程序中的每条语句都得到运行。在控制流图中，要求所有语句都被运行的充分必要条件是覆盖图中的所有节点。因此，语句覆盖准则可以定义为：让 T 为程序 P 的一个测试数据集，G_P 为 P 的控制流图，L_T 为与 T 相对应的图中的完整路径的集合。

定义 3-12 测试数据集 T 称为语句覆盖充分的，当且仅当 L_T 覆盖了 G_P 中的所有节点。NODE(L_T)为路径集合 G_P 中所覆盖的中的节点的集合，N_G 是 G_P 中所有节点的集合，则语句覆盖测试的覆盖率定义为：

$$\text{语句覆盖测试的覆盖率} = \frac{\|\text{NODE}(L_T)\|}{\|N_G\|}$$

【例 3-2】 求解一元二次方程根的程序如下。对于该程序选择 3 个测试用例{2、5、3}，{1、2、1}，{4、2、1}就能覆盖所有的节点。也就是说，这 3 个测试用例对语句覆盖来说是充分的。该程序所对应简化的控制流图如图 3-5 所示。

```
 1  void main()
 2  {
 3    float a,b,c,x1,x2,mid;
 4    scanf(" % f, % f, % f"a,b,c);
 5    if(a!= 0) {
 6      mid = b * b - 4 * a * c;
 7      if(mid > 0) {
 8        x1 = ( - b + sqrt(mid))/2;
 9        x2 = ( - b - sqrt(mid))/2;
10        printf("two real roots\n");
11    }
12    else {
13      if(mid = 0) {
14        x1 = - b/2 * a;
15        printf("one real root\n");
16      }
17      else {
18        x1 = - b/(2 * a);
19        x2 = sqrt( - mid)/(2 * a);
20        printf("two complex roots\n");
21      }
22    }
23    printf("x1 = % f,x2 = % f\n"x1,x2);
24  }
```

(2) 分支覆盖准则。

在例 3-2 中，虽然 3 个测试用例能保证所有语句都被执行，但并不能保证 $a=0$ 这条分支被测试。分支测试要求在软件测试中，每个分支都至少获得一次“真”值和一次“假”值，也就是使程序中的每个取“真”分支和取“假”分支都至少经历一次。在控制流图中，分支表现为图中的一条有向边。因此，分支测试的覆盖准则定义如下。

定义 3-13 测试数据集 T 称为分支覆盖充分的，当且仅当 L_T 覆盖了 G_P 中的所有有向边。让 EDGE(L_T)为路径集合 L_T 中所覆盖的 G_P 中的有向边的集合，E_G 是 G_P 中所有边的集合，则分支覆盖测试的覆盖率定义为：

$$\text{分支覆盖测试的覆盖率} = \frac{\|\text{EDGE}(L_T)\|}{\|E_G\|} \times 100\%$$

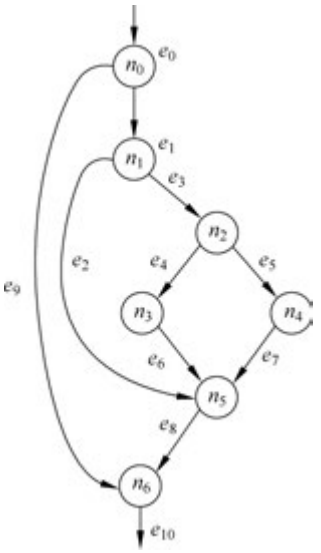

图 3-5 一元二次方程求根程序的控制流图

【例 3-3】 在图 3-5 中，选择 4 个测试用例{2、5、3}，{1、2、1}，{4、2、1}，{0、2、1}就能覆盖所有的分支，如表 3-1 所示。也就是说，这 4 个测试用例对分支覆盖来说是充分的。

表 3-1 分支覆盖测试用例

测试用例	a,b,c	$a!=0$	mid>0	mid==0
测试用例 1	2,5,3	真	真	假
测试用例 2	1,2,1	真	假	真
测试用例 3	4,2,1	真	假	假
测试用例 4	0,2,1	假	—	—

需要注意的是，上述测试用例在满足分支覆盖的同时，还满足了语句覆盖，因此分支覆盖要比语句覆盖更强一些，称**分支覆盖测试包含语句覆盖测试**。

(3) 谓词覆盖准则。

一个分支的条件是由谓词组成的。单个谓词称为原子谓词，例如在例 3-2 中，$a!=0$、mid 等都是原子谓词。而原子谓词通过逻辑运算符可以构成复合谓词，常见的逻辑运算符包括“与、或、非”等。对复合谓词而言，分支测试不是有效的。例如，对下列由复合谓词构成的语句

```
1  if(math>=9 || lang>=80 || poli>=75) x=1;
```

采用分支测试技术，只要原子谓词中的任何一个被满足，则该分支即为真，而不管其他的两个是否被满足。为此，原子谓词覆盖准则、分支-谓词覆盖准则和复合谓词覆盖准则被提出。

① 原子谓词覆盖准则。原子谓词测试要求在软件测试中，每个复合谓词所包含的每个原子谓词都至少获得一次“真”值和一次“假”值。定义如下。

定义 3-14 测试数据集 T 称为原子谓词覆盖充分的(原子谓词覆盖准则)，如果对任意一个分支中的任意一个原子谓词，T 中存在一个测试数据使其在运行时为真、为假至少各一次。设 ATOM(P)是程序 P 中所有原子谓词的集合，ATOMRUN(T)是对测试集 T 而言运行过程中原子谓词为真和为假的个数，则原子谓词的覆盖率定义为：

$$\text{原子谓词覆盖测试的覆盖率}=\frac{\|\text{ATOMRUN}(T)\|}{\|\text{ATOM}(P)\|}\times 100\%$$

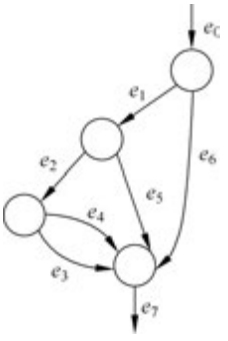

图 3-6 判断三角形类型的控制流图

【例 3-4】 程序实现下列需求：输入 3 个正整数，作为三角形的 3 边长，构成三角形，并指出三角形的类型。若是不等边三角形，使标志量置 1；若为等腰三角形，使标志量置 2；若为等边三角形，使标志量置 3；若无法构成三角形，使标志量置 4。控制流图如图 3-6 所示。

```
1  void main()
2  {
3     int i,j,k,match;
4     scanf("%d%d%d",&i,&j,&k);
5     if(i<=0 || j<=0 || k<=0 || i+j<=k || i+k<=j || j+k<=i) match=4;
6     else if(i==j && i==k && j==k) match=1;
```

```
    else if(i == j || i == k || j == k) match = 2;
    else match = 3;
    printf("match = %d\n", match);
}
```

对于该程序的测试，满足原子谓词覆盖准则的测试用例是：{－1 2 2}∪{2 －1 2}∪{2 2 －1}∪{1 1 2}∪{2 1 1}∪{1 2 1}∪{2 2 2}∪{2 2 3}∪{2 3 2}∪{3 2 2}∪{5 3 4}，其原子谓词的取值情况见表 3-2。

表 3-2　原子谓词覆盖测试用例

测试用例	变量	原子谓词								
	i,j,k	$i<=0$	$j<=0$	$k<=0$	$i+j<=k$	$i+k<=j$	$j+k<=i$	$i==j$	$i==k$	$j==k$
用例 1	－1,2,2	真	假	假	真	真	假	—	—	—
用例 2	2,－1,2	假	真	假	真	假	真	—	—	—
用例 3	2,2,－1	假	假	真	假	真	真	—	—	—
用例 4	1,1,2	假	假	假	真	假	假	—	—	—
用例 5	2,1,1	假	假	假	假	假	真	—	—	—
用例 6	1,2,1	假	假	假	假	真	假	—	—	—
用例 7	2,2,2	假	假	假	假	假	假	真	真	真
用例 8	2,2,3	假	假	假	假	假	假	真	假	假
用例 9	2,3,2	假	假	假	假	假	假	假	真	假
用例 10	3,2,2	假	假	假	假	假	假	假	假	真
用例 11	5,3,4	假	假	假	假	假	假	假	假	假

需要注意的是：原子谓词覆盖准则和语句覆盖准则相互之间没有包含关系，和分支覆盖准则相互之间也没有包含关系。因为原子谓词强调的是原子谓词的真假，而不考虑语句或分支是否被执行。例如，下面的语句：

```
if(x > 0 || y < 0) x++;
else x--;
```

当 $x=1$ 且 $y=1$ 和 $x=-1$ 且 $y=-1$，满足原子谓词覆盖准则，但不满足语句覆盖准则，因为语句 x--没被执行，同样也不满足分支覆盖准则。

② 分支-谓词覆盖准则。分支-谓词覆盖测试要求：不仅每个复合谓词所包含的每个原子谓词都至少获得一次“真”值和一次“假”值，而且每个复合谓词本身也至少获得一次”真”值和一次”假”值。定义如下。

定义 3-15　测试数据集 T 称为分支-谓词覆盖充分的(分支-谓词覆盖准则)，如果对任意一个分支和所包含的任意一个原子谓词，T 中存在一个测试数据在运行时为真、为假至少各一次。设 BRATOM(P)是程序 P 中所有分支及原子谓词的集合，BRATOMRUN(T)是对测试集 T 而言运行过程中分支和原子谓词为真和为假的个数，则分支-谓词的覆盖率定义为

$$\text{分支-谓词覆盖测试的覆盖率}=\frac{\|\text{BRATOMRUN}(T)\|}{\|2\times\text{BRATOM}(T)\|}\times 100\%$$

【例 3-5】　对于例 3-4 中程序的测试，可以观察到三个复合谓词本身也都取到了“真”值和“假”值，如表 3-3 所示。因此，上例中的测试用例同时也满足了分支-谓词覆盖准则。

表 3-3　分支-谓词覆盖测试用例

测试用例	变量	复合谓词		
	i,j,k	$i<=0\|\|j<=0\|\|k<=0\|\|i+j<=k\|\|i+k<=j\|\|j+k<=i$	$i==j\&\&i==k\&\&j==k$	$i==j\|\|i==k\|\|j==k$
用例 1	−1,2,2	真	—	—
用例 2	2,−1,2	真	—	—
用例 3	2,2,−1	真	—	—
用例 4	1,1,2	真	—	—
用例 5	2,1,1	真	—	—
用例 6	1,2,1	真	—	—
用例 7	2,2,2	假	真	—
用例 8	2,2,3	假	假	真
用例 9	2,3,2	假	假	真
用例 10	3,2,2	假	假	真
用例 11	5,3,4	假	假	假

从定义 3-15 和表 3-3 中可看出：分支-谓词覆盖准则包含语句覆盖准则、分支覆盖准则和原子谓词覆盖准则。

③ 复合谓词覆盖准则。在许多情况下，如果分支的条件比较复杂，则只有上述几种覆盖准则是不够的。于是，人们又提出了复合谓词覆盖准则。复合谓词测试要求在每个谓词中条件的各种可能都至少出现一次。定义如下。

定义 3-16　测试数据集 T 称为复合谓词覆盖充分的(复合谓词覆盖准则)，如果任意一个分支，对该分支所包含的谓词的任意一个可行的真假组合，T 中都存在一个测试数据使该组合谓词运行时，原子谓词的取值恰好为该真假值组合。设 COMATOM(P)是程序 P 中所有复合谓词的集合，令 COMATOMRUN(T)是对测试集 T 而言，运行过程中复合谓词为真和为假的个数，则复合谓词测试的覆盖率定义为

$$\text{复合谓词覆盖测试的覆盖率}=\frac{\|\text{COMATOMRUN}(T)\|}{\|2\times\text{COMATOM}(T)\|}\times 100\%$$

【例 3-6】　对于例 3-4 中程序的测试，满足复合谓词覆盖准则的测试用例是：

{−1 −1 −1}∪{−1 2 2}∪{2 −1 2}∪{2 2 −1}∪{1 1 2}∪{2 1 1}∪{1 2 1}∪{2 2 2}∪{2 2 3}∪{2 3 2}∪{3 2 2} ∪{5 3 4}其谓词的取值情况如表 3-4 所示。

表 3-4　复合谓词覆盖测试用例

测试用例	变量	原子谓词								
	i,j,k	$i<=0$	$j<=0$	$k<=0$	$i+j<=k$	$i+k<=j$	$j+k<=i$	$i==j$	$i==k$	$j==k$
用例 1	−1,−1,−1	真	真	真	真	真	真	—	—	—
用例 2	−1,−2,2	真	真	假	真	假	假	—	—	—
用例 3	2,−1,−2	假	真	真	假	假	真	—	—	—
用例 4	2,2,−1	假	假	真	假	真	真	—	—	—
用例 5	1,1,2	假	假	假	真	假	假	—	—	—
用例 6	2,1,1	假	假	假	假	假	真	—	—	—
用例 7	1,2,1	假	假	假	假	真	假	—	—	—

续表

测试用例	变量	原子谓词								
	i,j,k	$i<=0$	$j<=0$	$k<=0$	$i+j<=k$	$i+k<=j$	$j+k<=i$	$i==j$	$i==k$	$j==k$
用例 8	2,2,2	假	假	假	假	假	假	真	真	真
用例 9	2,2,3	假	假	假	假	假	假	真	假	假
用例 10	2,3,2	假	假	假	假	假	假	假	真	假
用例 11	3,2,2	假	假	假	假	假	假	假	假	真
用例 12	5,3,4	假	假	假	假	假	假	假	假	假

复合谓词覆盖准则包含语句覆盖准则、分支覆盖准则、原子谓词覆盖准则、分支-谓词覆盖准则。

上面各种覆盖准则之间的关系如图 3-7 所示，其中→表示准则之间的包含关系。

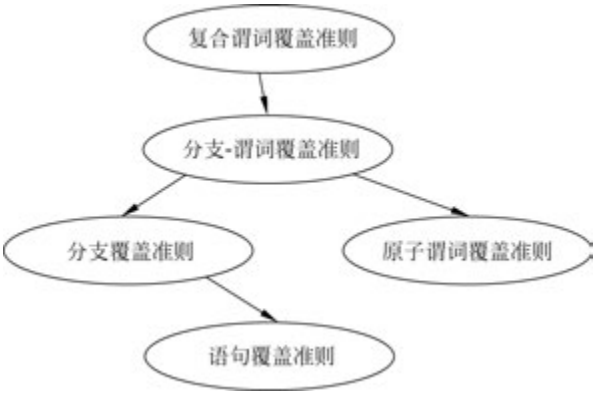

图 3-7　部分覆盖准则间的关系

(4) 路径覆盖准则。

对任意给定的程序，程序的执行过程是先给定输入，然后沿着不同的路径走向输出。因此，人们自然就想到了路径测试技术。路径测试技术要求观察程序运行的整个路径，要求程序的运行覆盖所有的完整路径。路径覆盖准则的定义如下：

定义 3-17　测试数据集 T 称为路径覆盖充分的，当且仅当 L_T 覆盖了 G_P 中的所有完整路径。令 $\mathrm{EP}(G_P)$ 为控制流图中的所有完整路径的集合，则路径覆盖测试的覆盖率定义为

$$\text{路径覆盖测试的覆盖率}=\frac{\|L_T\|}{\|\mathrm{EP}(G_P)\|}\times 100\%$$

【例 3-7】　对于例 3-4 中程序的测试，满足路径覆盖准则的测试用例是：{−1 2 2}∪{2 2 2}∪{2 2 3}∪{5 3 4}，其路径的覆盖情况如表 3-5 所示。

表 3-5　路径覆盖测试用例

测试用例	i,j,k	执行路径
用例 1	−1,2,2	$e_0e_6e_7$
用例 2	2,2,2	$e_0e_1e_5e_7$
用例 3	2,2,3	$e_0e_1e_2e_4e_7$
用例 4	5,3,4	$e_0e_1e_2e_3e_7$

路径覆盖准则包含了分支覆盖准则。但路径覆盖准则和原子谓词、分支-谓词和复合谓词之间没有包含关系。

路径覆盖准则需要测试的路径数目对许多程序来说往往是天文数字。例如，对于相互连接的 32 个双分支的程序，其分支的数目仅为 32 个，但路径的数目却是 2^{32} 个。因此，对一般的程序来说，要达到 100％的路径覆盖率几乎是不可能的。同时，判断不可行路径也是非常困难的，而程序中往往存在大量的不可行路径，这也是路径测试不太实用的主要原因。因此，实际使用时往往采取路径测试的一些简化测试技术，如简单路径覆盖准则和基本路径覆盖准则等。

考虑到一个有向图，其中的完整简单路径数目和完整基本路径数目往往也是很多的，完全的测试也是非常困难的。应用划分技术将大问题划成小问题，也是一个比较可行的方法。如图 3-8 所示，这是一个经常可以碰到的控制流图，节点每增加一个，路径数目则成倍增长。例如，当节点数目 $n=20$ 时，路径数目为 2^{20} 个，测试这么多的路径是不值得的。一是代价太大，二是也未必会发现更多的错误。如果限制路径的长度小于 10，则可以从中间将其划开，使每一部分包括的路径数目均为 2^{10} 个，这是可以接受的。

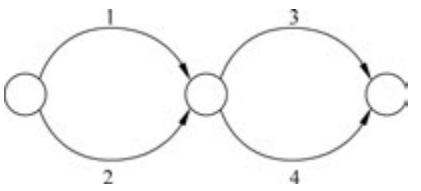

图 3-8　路径划分图

上面的讨论都没有涉及循环的概念。包括循环的控制流图的测试则更为复杂，因为考虑到循环的执行次数，这使得路径的数目会急剧增加。因此，一般在使用过程中，要限制循环的次数，例如让循环执行一次、两次等。如果循环中包括复杂的程序结构，例如分支、内循环等，由于每次循环的执行经过循环体时运行相同或不同的路径，则存在循环体内部的一些程序可能永远执行不了。所以在实际测试时，到底让循环执行多少次，要具体问题具体分析。

3.3.2　数据流分析

1. 数据流测试

控制流测试是面向程序的结构，控制流图和测试覆盖准则一旦给定，即可产生测试用例，它并不关心程序中每条语句是如何实现的。与控制流的测试思想不同，数据流测试是面向程序中的变量。

根据程序设计的理论，程序中的变量有两种不同的作用，一是将数据存储起来，二是将所存储的数据取出。这两种作用是通过变量在程序中所处的位置来决定。例如：当一个变量出现在赋值语句 $y=x_1+x_2$ 的左边时，它表示把赋值语句右边的计算结果存放在该变量所对应的存储空间内，也就是将数据与变量相绑定。当一个变量出现在赋值语句右边的表达式中时，表示该变量中所存储的数据被取出，参与计算，即与该变量相绑定的数据被引用。

定义 3-18　变量的定义性出现：若一个变量在程序中的某处出现使数据与该变量相绑定，则称该出现是定义性出现。

定义 3-19　变量的引用性出现：若一个变量在程序中的某处出现使与该变量相绑定的数据被引用，则称该出现是引用性出现。

一个变量被引用时，一般可以有两种用途：计算性引用和谓词性引用。计算性引用用

于计算新的数据，或为输出结果，或为中间计算结果等。谓词性引用用于计算判断控制转移方向的谓词，如出现在条件转移语句 If p then s than t 中的谓词 p 中。

为方便分析，可将前文介绍的控制流图加以改造，即把控制流图中的判断框去掉，把其中的谓词放在边上，这种图称为具有数据流信息的控制流图。

【例 3-8】 例 3-1 中计算最大公约数的程序对应的具有数据流信息的控制流图，如图 3-9 所示。

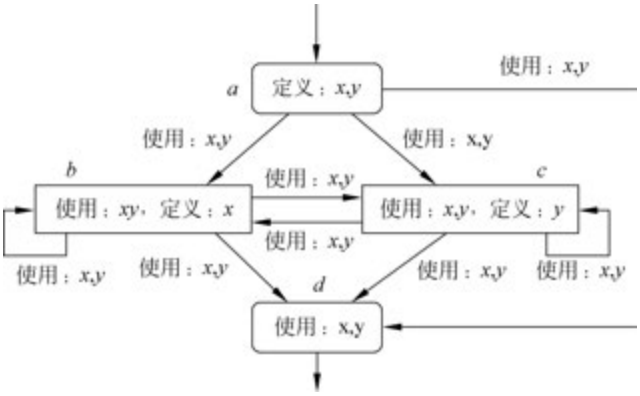

图 3-9　程序对应的具有数据流信息的控制流图

数据流测试的着眼点是测试程序中数据的定义与使用是否正确。它测试运行程序中从数据被绑定给一个变量之处到这个数据被引用之处的路径，通过它把一个变量的定义性出现传递到该定义的一个引用性出现。在具有数据流信息的控制流图中，一个变量的定义传递到该变量的一个引用可定义如下。

定义 3-20　让$<n_1,n_2,\cdots,n_k>$是具有数据流信息的控制流程图G_p中的一条路径，x是程序中的一个变量，如果节点$n_i(i=1,2,\cdots,k)$都不包含在x的定义性出现，则该路径称为对于x来说是无定义的。让n_0,n_k为G_p中的节点，n_0中包含变量x的定义性出现，n_k中包含变量x的计算性引用出现。从节点n_0到节点n_k的路径$<n_1,n_2,\cdots,n_k>$称为一条将x在n_0中的定义传递到n_k中的计算性引用的路径，如果$<n_1,n_2,\cdots,n_{k-1}>$是对于x来说无定义的，则称x在n_0中的定义可传递到n_k中的计算性引用。类似地，可以定义变量的一个定义性出现传递到一个谓词性引用的概念以及这样的传递路径。

2. 数据流覆盖准则

数据流测试方法着眼于测试每个数据的定义的正确性，通过考查每个定义的一个使用结果来判断该定义的正确性。

定义 3-21　定义覆盖测试准则：测试数据集T对测试程序P满足定义覆盖准则，如果对具有数据流信息的控制流图G_p中的每个变量x的每个定义性出现，若该定义性出现能够可行地传递到该变量的某个引用性出现，那么L_T中存在一条路径A，它包含一条子路径A'，使得A'将该定义出现传递到某一个引用性出现。

【例 3-9】 如图 3-10 所示，路径$A_1=<a,b,d,e>$，$A_2=<a,c,e>$覆盖了变量的所有定义性出现。因为对每个变量的每个定义性出现，都存在着一条子路径，通过它使该定义传递到它的某个引用。例如，变量a在节点b上的定义传递到有向边$<b,d>$上的谓词性引用，其传递路径是A_1的一条子路径。

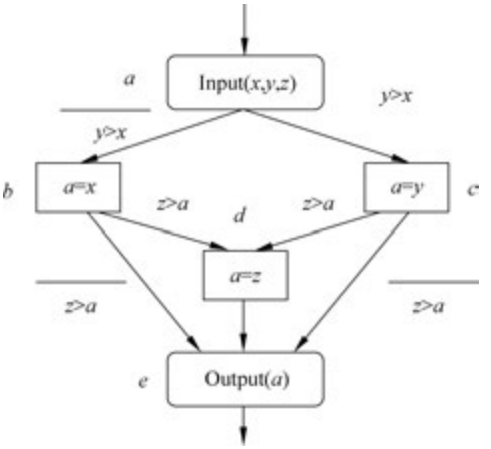

图 3-10　具有数据流信息的控制流图示例

定义 3-22　引用覆盖测试准则：测试数据集 T 对测试程序 P 满足引用覆盖准则，如果对具有数据流信息的控制流图中的每个变量 x 的每个定义 n，以及该定义的每个能够可行地传递到的引用 n'，L_T 中都存在一条路径 A，A 包含一条子路径 A'，使得 A' 将 n 传递到 n'。

【例 3-10】　对于图 3-9，有 3 个节点包含变量的定义性出现，这些节点是 a、b 和 c。

变量 x 在节点 a 上的定义性出现可传递到在节点 b、c 和 d 上的 x 的计算性引用，它还可以传递到有向边$<a,b>$,$<a,c>$,$<a,d>$,$<c,b>$,$<c,c>$上的谓词性引用。

变量 y 在节点 a 上的定义性出现可传递到在节点 b、c 和 d 上的 y 的计算性引用，它还可以传递到有向边$<a,b>$,$<b,b>$,$<a,c>$,$<a,d>$,$<b,c>$,$<b,d>$上的谓词性引用。

变量 x 在节点 b 上的定义性出现可传递到在节点 b、c 和 d 上的 x 的计算性引用，它还可以传递到有向边$<b,c>$,$<b,b>$,$<b,d>$,$<c,c>$,$<c,d>$上的谓词性引用。

变量 y 在节点 c 上的定义性出现可传递到在节点 b、c 和 d 上的 x 的计算性引用，它还可以传递到有向边$<c,b>$,$<b,b>$,$<b,c>$,$<c,c>$,$<c,d>$上的谓词性引用。

引用覆盖准则要求上述引用都被测试。下面是一个引用覆盖充分的测试路径集合：

$\{<a,b,b,c,c,b,b,d>,<a,d>,<a,c,c,b,b,c,d>\}$

引用覆盖准则的不足之处是路径集合中可能存在着回路，这样的路径可能是无穷的。因此，下列的定义-引用覆盖测试准则只检查无回路或只包含简单回路的路径。

定义 3-23　定义-引用覆盖测试准则：测试数据集 T 对测试程序 P 满足定义-引用覆盖测试准则，如果对具有数据流信息的控制流图 G_P 中的任意一条从定义传递到其引用的路径 A，若 A 是无回路的或者 A 只是开始节点和结束节点相同，那么 L_T 中存在一条路径 B，使得 A 是 B 的子路径。

实际使用时，定义-引用覆盖测试准则可以分为计算性引用的覆盖准则和谓词性引用的覆盖准则。这些覆盖准则之间的包含关系如下：定义-引用覆盖测试准则包含引用覆盖准则；引用覆盖准则包含定义覆盖准则。

3.3.3　程序插桩

程序插桩是一种基本的测试手段，是借助“向被测程序中插入操作（探针）来实现测试目的”的方法。在调试程序时，人们往往要在程序中插入一些打印语句，或者加一些断点来进

行程序调试。其目的在于,在程序执行时顺带打印出最关心的信息,并进一步通过这些信息了解执行过程中的一些动态特征。

1. 用于测试覆盖率和测试用例有效性度量的程序插桩

插桩技术是实现各种覆盖测试的必要手段,要统计各种成分的覆盖率,在一些点插入必要的信息是必不可少的。

【例 3-10】 对计算整数 x 和整数 y 的最大公约数的程序,其实现的流程图如图 3-11 所示。在该图中,虚线框表示插桩语句的位置。

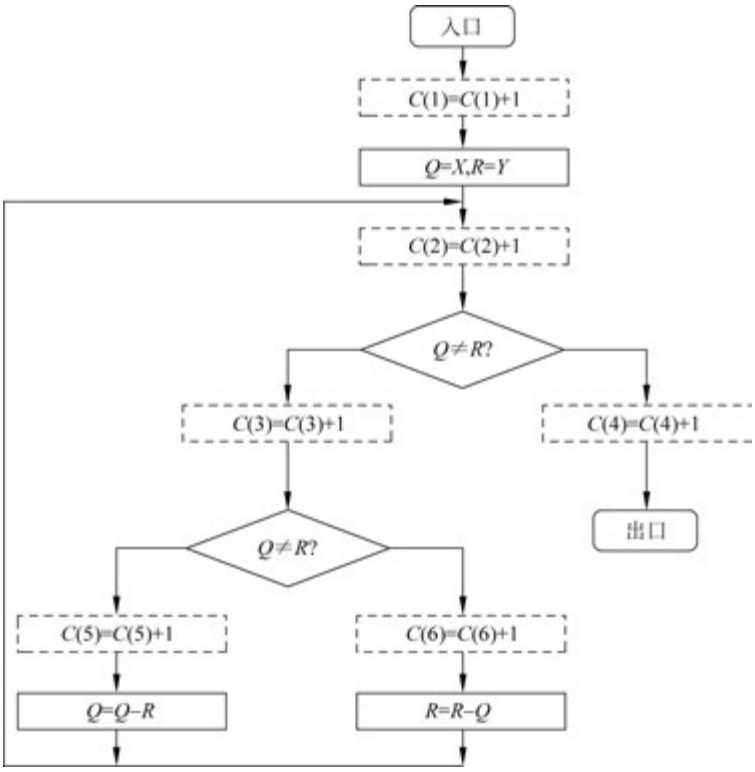

图 3-11 “求最大公约数程序”插桩后的流程图

程序从入口开始执行,到出口结束,所经历的插桩计数语句能够记录下程序各语句的执行次数。

程序插桩技术的研究涉及下列几个问题。

(1) 探测哪些信息?

这要根据具体的需求而定。例如,如果要统计各种覆盖准则下其对应元素的覆盖率,则用 $C(i)=C(i)+1$ 即可。如果在程序中间的某个位置要知道执行的结果是否正确,则要插入相应的打印语句。

(2) 在程序的什么位置设置探测点?

这也要根据具体的需求而定。例如,要统计各个分支执行比例,插入点的位置应放在每个分支之后。图 3-11 就是这种情况。

(3) 需要多少个探测点?

一般来讲,在满足需求的情况下,插入点的个数越少越好。当然,也不必特意去追求这

个最少值。对于许多软件测试工具或调试工具而言，加入插入点或断点都是一个基本的手段。而且，在测试需求已知的情况下，加入多少个插入点、在何处加入插入点都可以自动地计算，并自动地将相应的插入语句插入程序中的相应位置。

【例 3-11】 假设被调试的程序有 N 行，输出变量 x 的结果出错，则可以采用二分法找出程序的错误：在 $N/2$ 处加入一个打印变量 x 的语句，如果 x 在此处是正确的，则在[$N/2+1$，N]程序段内重复上述步骤。反之，则在[0，$N/2-1$]程序段内重复上述步骤，直到错误被定位为止。

2. 用于断言检测的程序插桩

在程序中的特定部位插入某些用以判断变量特性的语句，使得程序执行中这些语句得以实现，从而使程序的运行特性得以证实，把这些插入的语句称为**断言**(**Assertion**)。

断言语句也是程序插桩的一种，但与前面谈到的用于统计和评估的插入语句不同，断言语句的作用是当程序执行到这里时必须应该是什么，否则就会产生错误。断言语句对提高复杂程序的测试性是非常有用的。例如，在进行除法运算之前，加一条分母不为 0 的断言语句，可以有效地防止程序出错。

【例 3-12】 Divide 是计算两个非负数 NUM 和 DEN 的商的程序，假设 NUM＜DEN，采用＋、－、除 2 的运算，采用逐步逼近的方法达到求商的目的。程序中 E 是事先给定的精确度。

```
1  Divide(int NUM,int DEN,E,Q) {
2    float Q,E,A,B,W;
3    Q = 0; A = 0; B = DEN/2; W = 1;
4    While(W < E) {
5      if((NUM - A - B)> = 0) {
6        Q = Q + W/2;
7        A = A + B;
8      }
9      B = B/2;
10     W = W/2;
11   }
12 }
```

分析可知，在该程序的每一次执行迭代中，下面 4 个断言必定为真：

(1) W＝(k 是迭代次数)。

(2) A＝DEN * Q。

(3) B＝DEN * W/2。

(4) NUM/DEN－WQ NUM/DEN。

3.3.4 变异测试

程序变异(Program Mutation)测试是一种错误驱动测试，是针对某种类型的特定程序错误而提出来的。经过若干年的测试理论和软件测试的实践，人们逐渐发现要想找出程序的所有错误几乎是不可能的，比较现实的解决办法是将错误的搜索范围尽可能地缩小，以利于专门测试某类错误是否存在。这样做的好处是便于集中目标对付软件危害较大的可能错误，而暂时忽略危害较小的可能错误，从而取得较高的测试效率。程序变异测试有两种，即

程序强变异测试和程序弱变异测试。

1. 程序强变异测试

程序变异测试技术的基本思想是：对于给定的程序 P，先假定程序中存在一些小错误，每假设一个错误，程序 P 就变成 P'，如果假设了 n 个错误：$f_1, f_2, \cdots, f_n$，则对应有 n 个不同的程序：$P_1, P_2, \cdots, P_n$，这里 P_i 称为 P 的变异因子。

理论上，如果 P 是正确的，则 P_i 肯定是错误的，即存在测试数据 C_i，使得 P 和 P_i 的输出结果是不同的。因此，根据程序 P 和每个变异的程序，可以求得 $P_1, P_2, \cdots, P_n$ 的测试数据集 $C=\{C_1, C_2, \cdots, C_n\}$。

运行 C，如果对每个 C_i，P 都是正确的，而 P_i 都是错误的，这说明 P 的正确性较高。随着 n 的增大，P 的正确性也会越来越高。如果对某个 C_i，P 是错误的，而 P_i 是正确的，这说明 P 存在错误。

程序变异测试技术的关键是如何产生变异因子。常用的方法是通过对被测程序“应用变异算子”来产生变异因子。一个变异算子是一个程序转换规则，它把一种语法结构改变成另一种语法结构，保证转换后的程序的语法正确，但不保持语义的一致。

【例 3-13】 对程序中的一个表达式 $a<b$，可以用如下表达式之一来替换：

$a>b$，$a==b$，$a\neq b$，$a\geqslant b$，$a\leqslant b$，而产生一个变异因子。

变异运算是非常复杂的，它是根据程序中可能的错误而得出，可能有变量之间的替换、变量与常量之间的替换、算术运算符之间的替换、关系运算符之间的替换、关系运算符写得不全、逻辑运算符之间的替换等。可以说，对于一个一般的程序，其变异因子是非常庞大的，使用时要视具体问题而定。

【例 3-14】 对于下面的 C 语言程序：

```
1  void main()
2  {
3    int i;
4    float e,sum,term,x;
5    scanf("%f%f",&x,&e);
6    printf("x=%10.6fe=%10.6f\n",x,e);
7    term=x;
8    for(i=3; i<=100&&term>e; i=i+2) {
9      term=term*x*x/(i*(i-1));
10     if(i%2==0) sum=sum+term;
11     else sum=sum-term;
12   }
13   printf("sin(x)=%8.6f\n", sum);
14 }
```

下面是它的几个变体：

(1) term=0，产生一个变体。

(2) i>=100 && term>e，产生一个变体。

(3) i%2==1，产生一个变体。

在程序变异测试方法中，测试人员可以有选择地使用变异算子的一个子集来完成不同层次的测试分析，增加了测试的灵活性。同时，变异因子与原来程序的差别可以提供十分有用的信息，使测试人员较容易发现软件中的错误所在并排除之，这也成为变异测试的一个

优点。

变异测试的缺点是它需要大量的计算机资源来完成测试充分性分析。对于一个中等规模的软件，所需的存储空间也是巨大的；运行大量变异因子也导致了时间上巨大的开销，这些问题从某种程度上限制了变异测试方法的实际应用。

2. 程序弱变异测试

当变异因子比较多时，运用强变异测试技术需要花费大量的时间。弱变异测试方法的目标仍是要查出某类错误，但把注意力集中在程序中的一系列基本组成成分上，考虑在一个组成成分内部的错误是否可以在某个局部位置被发现。其主要思想如下。

设 P 是一个程序，C 是 P 的简单组成部分，若有一变异变换作用于 C 而生成 C'，如果 P' 是含有 C' 的 P 的变异因子，则在弱变异方法中，要求存在测试数据，当 P 在此测试数据下运行时，C 被执行，且至少在一次执行中，使 C 的产生值与 C' 不同。

从这里可以看出，弱变异和强变异有很多相似之处。其主要差别在于：弱变异强调的是变动程序的组成部分。根据弱变异准则，只要事先确定导致 C 与 C' 产生不同值的测试数据组，则可将程序在此测试数据组上运行，而并不实际产生其变异因子。

弱变异实现的关键的问题是确定程序的组成部分集合以及与其有关的变换。其中，组成部分可以是程序中的计算结构、变量定义与引用、算术表达式、关系表达式和布尔表达式等。

(1) 变量的定义与引用。

这种变异一般是采用变量替换。例如，语句 $A=B$；可以将 A 和 B 换成其他变量。

(2) 算术表达式。

设 exp 是一个表达式，一般将其变换成 exp+C 或者 C * exp。

(3) 关系表达式。

$\exp_1 R \exp_2$ 是关系表达式，这里 $\exp_1$ 和 $\exp_2$ 是算术表达式，R 是关系运算符。R 出错率最高而且难以发现。例如，<变换成<=、<=变换成<、>变换成>=、>=变换成>等。

(4) 布尔表达式。

可以采用随机测试的方式来进行变换。

弱变异测试方法的主要优点是开销较小，效率较高。然而，无论是强变异测试还是弱变异测试，在实际使用过程中对所变异的故障类型难以掌握。要么所涉及的故障太多，导致实际测试这些故障是不可能的；要么涉及的故障太少，对实际故障的检测起不了什么作用。因此，变异测试有很大的局限性。

3.3.5 编码标准一致性检查

代码检查是静态测试的主要方法，它包括代码走查、桌面检查、流程图审查等。

1. 代码检查

代码检查主要检查代码和流图设计的一致性、代码结构的合理性、代码编写的标准性、可读性、代码逻辑表达的正确性等方面。它包括变量检查、命名和类型审查、程序逻辑审查、程序语法检查和程序结构检查等内容。

最常见的静态测试是找出源代码的语法错误，这类测试可由编译器来完成。编译器可以逐行分析检验程序的语法，找出错误并报告。除此之外，测试人员需要采用人工方法来检

验程序，有些地方存在非语法方面的错误，只能通过人工检测的方法来判断。

2. 代码检查的目的

代码检查是为达到以下目的。

（1）检查程序是不是按照某种标准或规范编写的。

（2）发现程序缺陷。

（3）发现程序产生的错误。

（4）检查代码是不是流程图要求的。

（5）检查有没有遗漏的项目。

（6）使代码易于移植，因为代码经常需要在不同的硬件中运行，或者使用不同的编译器编译。

（7）使代码易于阅读、理解和维护。

3. 代码检查需要的文档

在进行代码检查前应准备好需求文档、程序设计文档、程序的源代码清单、代码编码标准、代码缺陷检查表和流程图等。

4. 代码检查的方式

代码检查的方式有以下 3 种。

（1）桌面检查。

桌面检查是程序员对源程序代码进行分析、检验，并补充相关的文档，发现程序中的错误的过程。由于程序员熟悉自己的程序，可以由程序员自己检查，这样可以节省很多时间，但要注意避免自己的主观判断。

（2）代码走查。

代码走查是程序员和测试员组成的审查小组通过逻辑运行程序，发现问题。小组成员要提前阅读设计规格书、程序文本等相关文档，利用测试用例，使程序逻辑运行。

走查可分为以下两个步骤：

① 小组负责人把材料发给每个组员，然后由小组成员提出发现的问题。

② 通过记录，小组成员对程序逻辑及功能提出自己的疑问，开会探讨发现的问题和解决方法。

（3）代码审查。

代码审查是程序员和测试员组成的审查小组通过阅读、讨论、分析技术对程序进行静态分析的过程。

代码审查可分为以下两个步骤。

① 小组负责人把程序文本、规范、相关要求、流程图及设计说明书发给每个成员。

② 每个成员将所发材料作为审查依据，但是由程序员讲解程序的结构、逻辑和源程序。在此过程中，小组成员可以提出自己的疑问；程序员在讲解自己的程序时，也能发现自己原来没有注意到的问题。

为了提高效率，小组在审查会议前可以准备一份常见错误清单，提供给参加成员对照检查。在实际应用中，代码检查能快速找到 20％～30％的编码缺陷和逻辑设计缺陷。代码检查看到的是问题本身而非问题的征兆。它需要消耗时间，而且需要知识和经验的积累。

5. 代码检查项目

代码检查项目包括以下内容。

(1) 目录文件组织。

目录文件组织要遵循以下原则：

① 所有的文件名简单明了，见名知意。

② 文件和模块分组清晰。

③ 每行代码在 80 个字符以内。

④ 每个文件只包含一个完整模块的代码。

(2) 检查函数。

检查函数要遵循以下原则。

① 函数头清晰地描述了函数的功能。

② 函数的名字清晰地定义了它所要做的事情。

③ 各个参数的定义和排序遵循特定的顺序。

④ 所有的参数都要是有用的。

⑤ 函数参数接口关系清晰明了。

⑥ 函数所使用的算法要有说明。

(3) 数据类型及变量。

数据类型及变量要遵循以下原则。

① 每个数据类型都有其解释。

② 每个数据类型都有正确的取值。

③ 数据结构尽量简单，降低复杂性。

④ 每一个变量的命名都明确地表示了其代表什么。

⑤ 所有的变量都要被使用。

⑥ 全部变量的描述要清晰。

(4) 检查条件判断语句。

检查条件判断语句要遵循以下原则。

① 条件检查和代码在程序中清晰表露。

② if-else 语句使用正确。

③ 数字、字符和指针判断明确。

④ 最常见的情况优先判断。

(5) 检查循环体制。

检查循环体制要遵循以下原则。

① 任何循环不得为空。

② 循环体系清晰易懂。

③ 当有明确的多次循环操作时使用循环。

④ 循环命名要有意义。

⑤ 循环终止条件清晰。

(6) 检查代码注释。

检查代码注释时要遵循以下原则。

① 有一个简单的关于代码结构的说明。

② 每个文件和模块都要有相应的解释。

③ 源代码能够解释，清晰易懂。

④ 每个代码的解释说明要明确地表达出代码的意义。

⑤ 所有注释要具体、清晰。

⑥ 所有无用的代码及注释要删除。

(7) 桌面检查。

进行桌面检查时要注意以下问题。

① 检查代码和设计的一致性。

② 代码对标准的遵循、可读性。

③ 代码逻辑表达的正确性。

④ 代码结构的合理性。

⑤ 程序编写与编写标准的符合性。

⑥ 程序中不安全、不明确和模糊的部分。

⑦ 编程风格问题等。

(8) 其他检查。

其他检查包括如下内容。

① 软件的扩展字符、编码、兼容性、警告/提示信息。

② 检查变量的交叉引用表：检查未说明的变量和违反了类型规定的变量，以及变量的引用和使用情况。

③ 检查标号的交叉引用表：验证所有标号的正确性。

④ 检查子程序、宏、函数：验证每次调用与所调用位置是否正确，调用的子程序、宏、函数是否存在，参数是否一致。

⑤ 等价性检查：检查全部等价变量的类型的一致性。

⑥ 常量检查：确认常量的取值和数制、数据类型。

⑦ 标准检查：检查程序中是否有违反标准的问题。

⑧ 风格检查：检查程序的设计风格。

⑨ 比较控制流：比较设计控制流图和实际程序生成的控制流图的差异。

⑩ 选择、激活路径：在设计控制流图中选择某条路径，然后在实际的程序中激活这条路径，如果不能激活，则程序可能有错。

⑪ 补充文档：根据以上检查项目，可以编制代码规则、规范和检查表等作为测试用例。

⑫ 对照程序的规格说明，详细阅读源代码，比较实际的代码，从差异中发现程序的问题和错误。

⑬ 检查必须遵守规定代码的语法格式和规则(如排版、注释、标识符命名、可读性、变量、函数、过程、可测性、程序效率、质量保证、代码编辑、编译、审查、代码测试、维护、宏)等各方面的编码要求。

6. 代码走查表

在进行人工代码检查时，可以制作代码走查缺陷表。在该缺陷检查表中，存在工作中遇到的典型错误，如下所示。

(1) 格式部分。

① 嵌套的 if 是否正确地缩进。

② 注释是否准确并有意义。

③ 使用的符号是否有意义。

④ 代码基本上是否与开始时的模块模式统一和一致。

⑤ 是否遵循了全套的编程标准。

(2) 入口和出口的连接。

① 初始入口和最终出口是否正确。

② 被传送的参数值是否正确地设置。

③ 对关键的被调用的模块的意外情况是否有所处理(如丢失等)。

④ 对另一个模块的每一次调用,全部所需的参数是否传送给每个被调用的模块。

(3) 存储器问题。

① 每个域在第一次使用前是否正确地初始化。

② 规定的域是否正确。

③ 每个域是否有正确的变量类型声明。

(4) 判断及转移。

① 用于判断的变量是否正确。

② 是否判断了正确的条件。

③ 每个转移目标是否正确并且至少执行了一次。

(5) 性能。

性能是否最佳。

(6) 可维护性。

① 清单格式是否适用于提高可读性。

② 各个程序块之间是否符合代码的逻辑意义。

(7) 逻辑。

① 全部设计是否已经实现。

② 代码所做的是否是设计规定的内容。

③ 每个循环是否执行了正确的次数。

(8) 可靠性。

对从外部接口采集的数据是否确认过。

(9) 内存设计。

① 数组或指针的下标是否越界。

② 是否修改了指向常量的指针的内容。

③ 是否有效地处理了内存耗尽的问题。

④ 是否出现了不规范指针使用(如指针变量没有被初始化、释放内存之后未将指针设置为空等)。

⑤ 是否忘记为数组和动态内存赋初值。

⑥ 申请内存之后,是否立即检查指针值是否为空。

(10) 关于类的高级特性。

是否违背了继承和组合的规则。

3.4 静态测试实践

3.4.1 指导原则

1. 提高评审有效性

(1) 技术因素。

① 保证遵循评审的基本过程,特别是针对审查这样的正式评审类型。

② 尽早参与软件工作产品的评审,以提前识别其中的缺陷,防止缺陷的雪崩效应。

③ 合理定义评审的入口准则以提高评审的效率和有效性。

④ 针对不同的软件工作产品选择不同的评审类型,并应用不同的检查列表。

⑤ 鼓励发现最重要的缺陷,注重内容而非形式。

⑥ 持续改进评审过程。

(2) 组织因素。

① 管理层即使在时间和成本压力下也能大力支持评审活动。

② 管理层给予足够的时间修改评审过程中发现的缺陷。

③ 不要将评审过程中收集的度量数据用于个人绩效评估。

④ 为评审员提供评审技能方面的培训,特别是审查这样的评审类型。

⑤ 选择最重要的软件工作产品进行评审。

⑥ 管理层认可评审过程中所取得的改进。

⑦ 组织内营造“无责备”的氛围,从而乐于接受识别的缺陷。

(3) 人员因素。

① 确保能有合适的评审员参与不同类型的评审,且在技术和行业背景知识方面达到良好的平衡。

② 评审可以发现缺陷并改进软件工作产品的质量,且在利益相关者之间达成共识。

③ 确保评审对每个参与人员而言都是一次正面积极的经历。

④ 确保评审员的评审意见具有建设性、有益性和客观性。

⑤ 鼓励评审员深层次思考评审对象中最重要的内容。

⑥ 在作者不同意或者不愿意的情况下,不进行评审。

2. 发现程序的错误

(1) 使用了错误的局部变量或全局变量。

(2) 不匹配的参数。

(3) 未定义的变量。

(4) 遗漏的标号或代码。

(5) 不适当的循环嵌套或分支嵌套。

(6) 不适当的处理顺序。

(7) 无终止的死循环。

(8) 不允许的递归。

(9) 调用并不存在的子程序。

(10) 不适当的连接。

3. 寻找潜伏问题的原因

(1) 未使用过的变量。

(2) 不会执行到的冗余代码。

(3) 可疑的计算。

(4) 潜在的死循环。

4. 提供程序的信息

(1) 每一类型语句出现的次数。

(2) 所用变量和常量的交叉引用表。

(3) 标识符的使用方式。

(4) 过程的调用层次。

(5) 程序代码违背编码规则。

5. 选择测试用例

软件测试本质是对被测试内容确定一组测试用例。测试用例应包含如下信息。

(1) 输入。

输入实际上有两种类型,即前提(在测试用例执行之前已经存在的环境)和由某种测试方法所标识的实际输入。

(2) 输出。

输出也有两类,即预期输出和实际输出。

(3) 测试活动要建立必要的前提条件。

提供测试用例输入,观察输出,并且将实际输出与预期输出进行比较,以确定该测试是否通过。

(4) 良好的测试用例信息也支持测试管理。

测试用例信息应该能够记录测试用例的执行历史,包括:测试用例是什么时候由谁运行的、每次执行的通过/失败记录、测试的是哪个软件版本等。测试用例同样也需要被开发、评审、使用、管理和保存。

6. 为查错做准备

静态分析可为查错做准备,在静态分析过程中计算机并不运行被测试的程序,这是静态分析与人工测试的根本区别。

3.4.2 实践案例

【例 3-15】 C 语言程序的静态分析。

```
1  max(float x,float y) {
2    return z = x > y ? x : y;
3  }
4  main() {
5    float a,b; int c;
6    scanf(" % f, % f",&a,&b);
7    c = max(a,b);
8    printf("MAX IS % f\n",c);
9  }
```

静态测试发现的问题如下。

(1) 必须修改的：程序没有注释；子函数没有返回值类型；精度丢失。

(2) 建议修改的：main 函数没有返回值类型和参数列表；一行代码只定义一个变量；程序适当加空行。

通过静态测试后对程序修改如下：

```
/*
   程序名称：求两个实数中的较大值
   作者：×××
   版本：1.0
   创建时间：×年×月×日
 */

float max(float x,float y) { //返回两个单精度数中的大数
  return z = x > y ? x : y;
}
void main() {
  float a,b;
  int c;
  scanf("%f,%f",&a,&b);
  c = max(a,b);
  printf("MAX IS %f\n",c);
}
```

3.5 小　　结

静态测试的特点包括：①不必动态地执行程序，不必进行测试用例设计和结果判读等工作；②可以由人手工方式进行，充分发挥人的优势，行之有效；③充分利用思维互补的情形：软件设计者的思维及交流障碍而造成的逻辑错误，由测试人员通过逻辑思维去解决，行之有效且检验出错误的水平较高；④静态测试实施不需要特别条件，容易开展。

本章首先介绍了评审的作用、原则、过程和人员分工，然后详细介绍了五种评审类型(需求、概要设计、详细设计、数据库和测试)和各自测试内容，接着从控制流分析、数据流分析、程序插桩、变异测试和编码标准一致性检查等方面详细介绍了常用的静态分析方法，最后归纳出静态测试实践的指导原则。

3.6 习　　题

1. 软件评审的作用(目的)是什么？
2. 软件评审的基本原则是什么？
3. 软件评审的过程包括哪些步骤？
4. 如何做好需求评审？
5. 请简述软件概要设计说明的评审细则。
6. 请简述软件详细设计说明的评审细则。

7. 请简述数据库设计说明的评审细则。
8. 请简述软件测试需求规格说明的评审细则。
9. 请简述软件测试计划的评审细则。
10. 请简述软件测试说明、软件测试报告和软件测试记录的评审细则。
11. 控制流覆盖准则包含哪些？
12. 数据流覆盖准则包含哪些？
13. 什么是程序插桩？常用的插桩技术有哪些？
14. 什么是变异测试？请简述程序强变异测试的思想。
15. 请简述程序弱变异测试和程序强变异测试的异同点。
16. 什么是代码检查？代码检查的目的是什么？
17. 常见的代码检查方式有哪些？它们各自的特点、实施步骤是什么？
18. 请简述代码检查项目的内容。
19. 请简述代码走查表包括的内容。
20. 请简述静态测试实践的指导原则。

习题 3