哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

软件工程

第四章 软件测试

4-1 软件测试基础

刘铭

2024年5月21日

主要内容

1. 软件质量
2. 代码复审：静态程序分析
3. 性能分析：动态程序分析
4. 软件测试基础
5. 测试过程
6. 测试方法分类
7. 测试用例

1. 软件质量

什么是软件质量？

- 消费者观点
  - 适合使用：产品或服务就应该是被期望的那样
  - 设计质量：设计的质量特性应包含在产品或服务中
  - 用户满意度=合格的产品 + 好的质量 + 按预算和进度安排交付
- 生产者观点
  - 质量的一致性：确保产品或服务是根据设计制造的
  - 利润：确保用最少的成本投入获取最大利益
- 质量：某一事物的特征或属性。（美国传统字典）
  - 可测量的特征——与已知标准可以进行比较，如长度、速度等
  - 软件是一种知识实体，其特征的定义远比物理对象要困难的多

软件质量

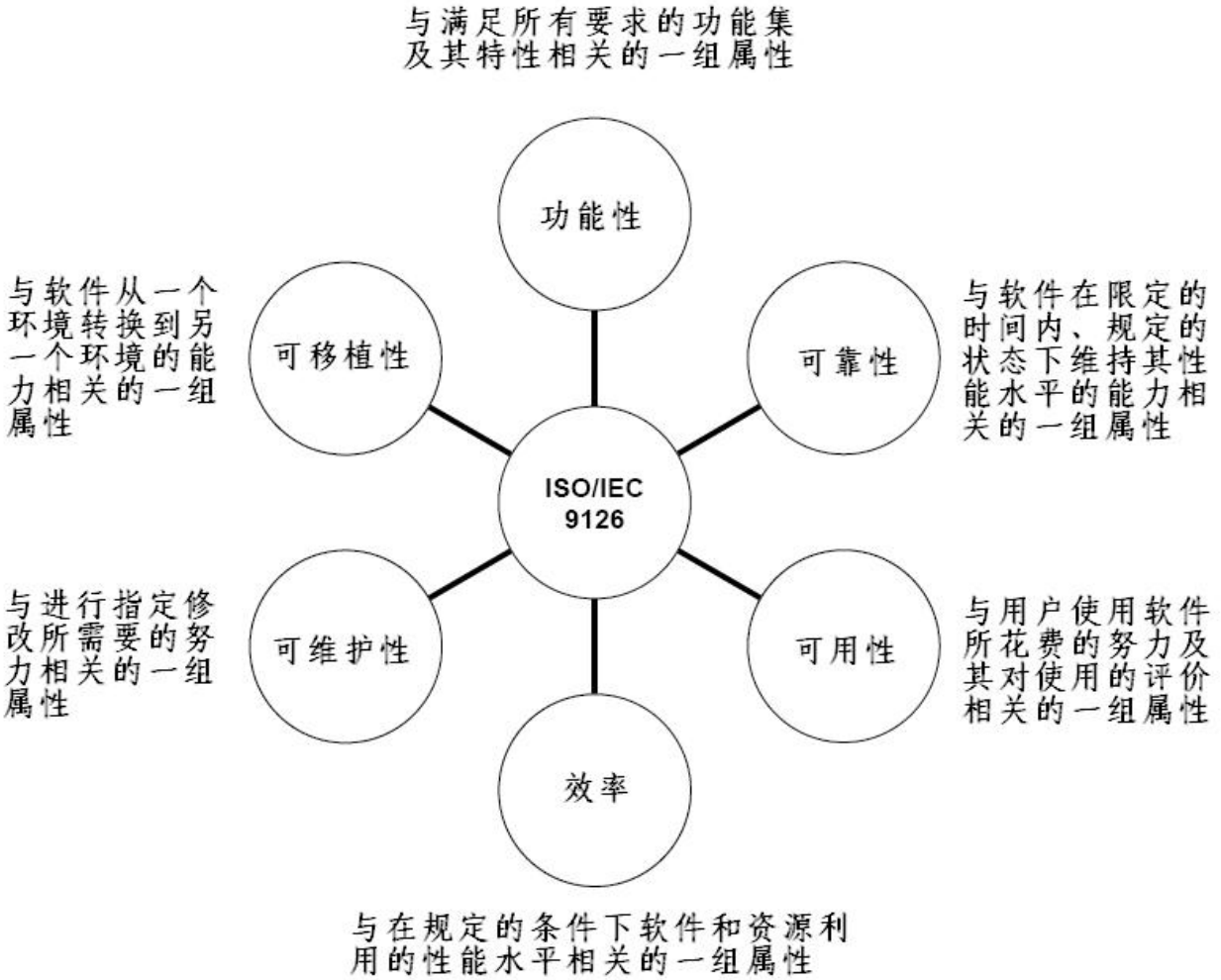
- **软件质量是对明确陈述的功能和性能需求、明确记录的开发标准以及对所有专业化软件开发应具备的隐含特性的符合度。**
  - 软件需求是质量测量的基础，不符合需求就是没有质量
  - 特定标准定义了一组用以指导软件开发方式的准则。若未能遵守准则，则肯定质量成问题
  - 有一组未被提及的隐式需求（如：对易用性的期望）。若软件符合显示需求，但未能满足其隐式需求，则软件质量仍然值得怀疑

软件质量

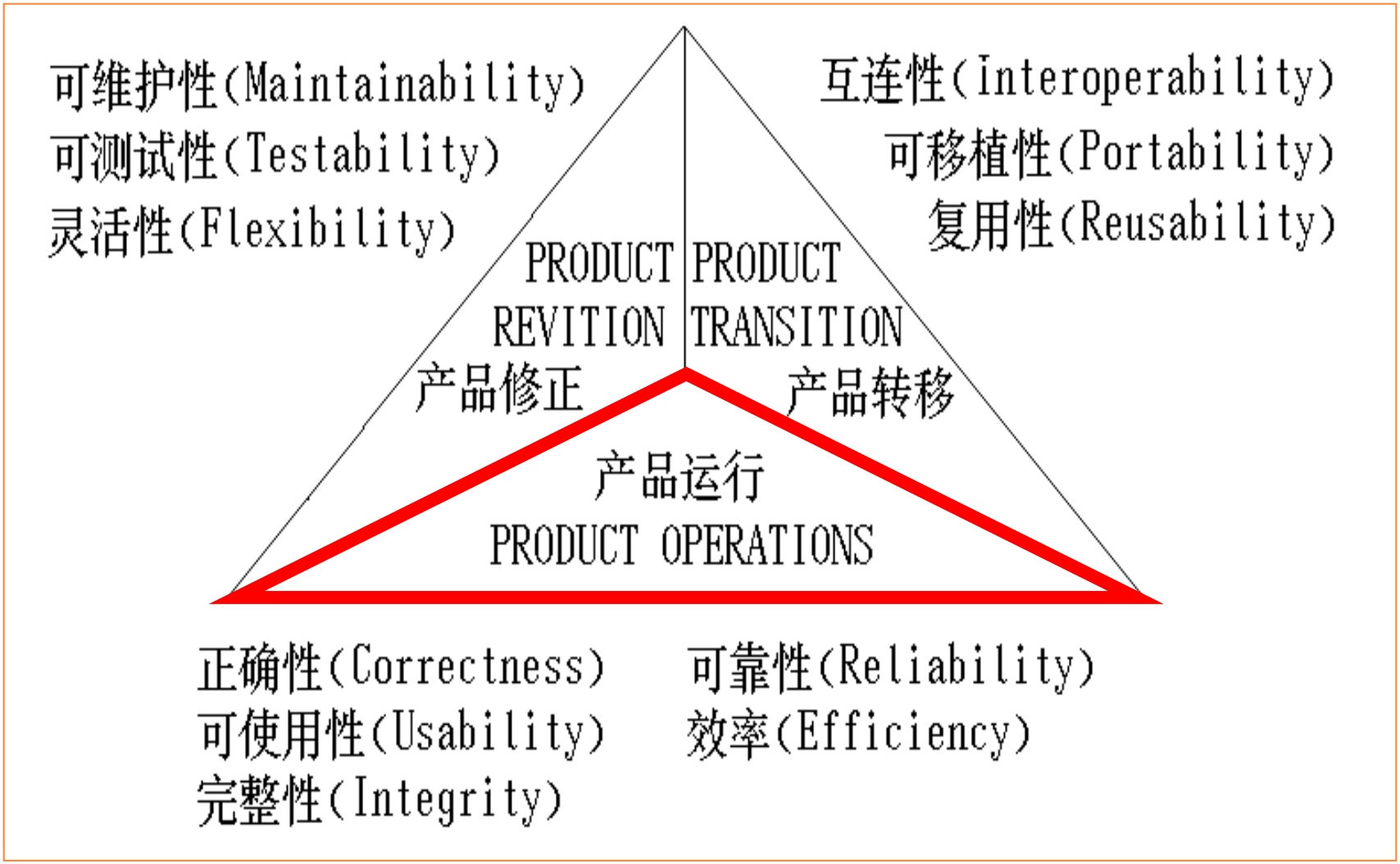
- 软件质量是许多质量属性的综合体现，各种质量属性反映了软件质量的方方面面。人们通过改善软件的各种质量属性，从而提高软件的整体质量（否则无从下手）。
  - 软件的质量属性众多：正确性、精确性，健壮性、可靠性、容错性、性能、易用性、安全性、可扩展性、可复用性、兼容性、可移植性、可测试性、可维护性、灵活性等。
  - 质量属性可分为两大类：“功能性”与“非功能性”，后者有时也称为“能力”（Capability）。

软件质量属性

国际软件质量标准：ISO/IEC 9126



软件质量属性


可维护性(Maintainability)
可测试性(Testability)
灵活性(Flexibility)
互连性(Interoperability)
可移植性(Portability)
复用性(Reusability)
PRODUCT REVITION
产品修正
PRODUCT TRANSITION
产品转移
产品运行
PRODUCT OPERATIONS
正确性(Correctness)
可使用性(Usability)
完整性(Integrity)
可靠性(Reliability)
效率(Efficiency)

软件质量要素

- **软件质量要素**
  - 什么是软件质量要素？
    - 从**技术角度**讲，对软件整体质量影响最大的那些质量属性才是质量要素
    - 从**商业角度**讲，客户最关心的、能成为卖点的质量属性才是质量要素
  - 对于一个特定的软件而言，我们首先判断什么是质量要素，才能给出提高质量的具体措施，而不是一股脑地想把所有的质量属性都做好，否则不仅做不好，还可能得不偿失。
  - 如果某些质量属性并不能产生显著的经济效益，我们可以忽略它们，把精力用在对经济效益贡献最大的质量要素上。简而言之，**只有质量要素才值得开发人员下功夫去改善。**

商业目标决定质量目标！

- **教科书的片面观点**
  - 大凡软件工程教科书为了强调质量的重要性，总是要举一些历史上发生过的重大软件质量事故，例如航天飞机爆炸、核电站失事、爱国者导弹发生故障等等。这些事故的确不是危言耸听，给人们敲响了质量的警钟。
  - 学术界总是喜欢宣扬质量至上的理念，而**忽视企业的商业利益，将质量目标凌驾于商业目标之上**。
  - **重视软件质量是应该的，但是“质量越高越好”并不是普适的真理。只有极少数软件应该追求“零缺陷”，对绝大多数软件而言，商业目标决定了质量目标，而不该把质量目标凌驾于商业目标之上。**

- **严格系统对质量的要求**
  - 航空、航天等系统对质量要求极高，任何缺陷都有可能导致机毁人亡，所以人们不惜一切代价去消除缺陷。在发射航天器之前，只要发现任何异常，就会立即取消发射指令，直到异常被消除为止。

商业目标决定质量目标！

- **商业目标决定质量目标**
  - 严格系统毕竟是少数，绝大多数普通软件的缺陷并不会造成机毁人亡的重大损失；日常工作中用到的软件几乎都是有缺陷的，即便是Microsoft的软件产品也经常出错甚至导致死机，人们抱怨后往往会继续使用有缺陷的软件。
  - 企业的根本目标是为了获取尽可能多的利润，而不是生产完美无缺的产品。如果企业销售出去的软件的质量比较差，轻则挨骂，重则被退货甚至被索赔，因此为了提高用户对产品的满意度，企业必须提高产品的质量。但是企业不可能为了追求完美的质量而不惜一切代价，当企业为提高质量所付出的代价超过销售收益时，这个产品已经没有商业价值了，还不如不开发。
  - 企业必须权衡质量、效率和成本，产品质量太低了或者太高了，都不利于企业获取利润。企业理想的质量目标不是“零缺陷”，而是恰好让广大用户满意，并且将提高质量所付出的代价控制在预算之内。
  - 生产“足够好”的软件即可

质量管理活动

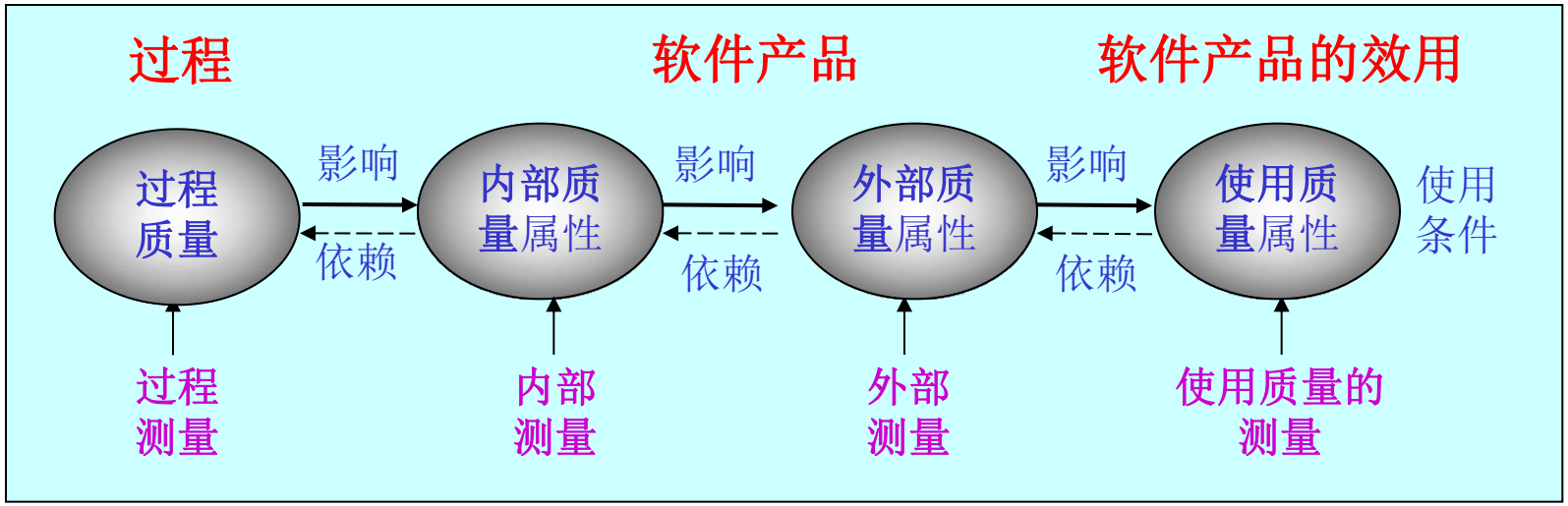
- **质量计划**
  - 为特定的项目选择合适的程序和标准，并按要求进行修改、调整
- **质量控制（QC）**
  - 为了保证每一件工作产品都能满足对它的需求而在整个软件过程中所运用的一系列审查、评审和测试。
  - **检验产品的质量，保证产品符合客户的需求；是产品质量检查者；**
- **质量保证（QA）**
  - 为了保证软件高质量而必需的“有计划的、系统化的行动模式”。
  - **审计过程的质量，保证过程被正确执行；是过程质量审计者；**
- **质量管理应与项目管理分离，以确保其独立性**

质量成本的构成

- **预防成本**
  - 质量计划
  - 正式技术评审
  - 测试设施
  - 培训
- **鉴定成本**
  - 审查
  - 设备校准和维护
  - 测试
- **失效成本**
  - 内部失效成本
    - 返工
    - 修复
    - 错误分析
  - 外部失效成本
    - 投诉解决
    - 退换产品
    - 服务支持
    - 保修工作

全面质量管理

ISO-9126的软件质量模型框架



过程质量 有助于提高 产品质量
产品质量 有助于提高 使用质量

全面质量管理

- 事先预防的思想，“缺陷越早发现越早修改越经济”的原则
- 重视质量保证（SQA）工作，检查开发和管理活动是否与已定的过程策略、标准和流程一致，检查工作产品是否遵循模板规定的内容和格式

1. 软件质量保证活动

- 为项目准备SQA计划
- 参与开发项目的软件过程描述
- 评审各项软件工程活动，以验证其是否符合定义的软件过程
- 审核指定的软件工作产品，以验证其是否符合定义的软件过程中的相应部分
- 确保软件工作及工作产品中出现的偏差已文档化，并且按照文档化的规程进行了处理
- 记录所有不符合的部分，并报告给高层管理者

软件质量保证的要素

- 标准
- 评审和审核
- 测试
- 错误/缺陷的收集和分析
- 变更管理
- 教育
- 供应商管理
- 安全管理
- 安全
- 风险管理

2. 全面软件质量管理：质量管理计划

- **质量管理计划**
  - 质量管理计划就是为了实现质量目标的计划。而质量目标则是由商业目标决定的。开发软件产品的最终目的是为了赚钱，所以人们为提高软件质量所付出的代价是有上限的，项目负责人当然希望代价越低越好。**质量管理计划是全面质量管理的行动纲领。**
  - 谁制定质量管理计划？由项目核心成员和质量人员共同协商制定，主要由质量人员起草，由项目经理审批即可。
  - 质量管理计划的主要内容：
    - **1. 质量要素分析**
    - **2. 质量目标**
    - **3. 人员与职责**
    - **4. 过程检查计划**
    - **5. 技术评审计划**
    - **6. 软件测试计划**
    - **7. 缺陷跟踪工具**
    - **8. 审批意见**

3. 全面软件质量管理：软件评审

- **软件评审是软件过程中的“过滤器”**
- **目的是尽早地发现工作成果中的缺陷，并帮助开发人员及时消除缺陷，从而有效地提高产品的质量**
- **软件评审包括“正式技术评审”、“走查”、“审查”、“轮查”等**
  - 发现软件的任何一种表示形式中的功能、逻辑或实现上的错误
  - 验证评审中的软件是否满足其需求
  - 保证软件的表示符合预先定义的标准
  - 得到以统一的方式开发的软件
  - 使项目更易于管理
- **技术评审的主要好处：**
  - 通过消除工作成果的缺陷而提高产品的质量；
  - 技术评审可以在任何开发阶段执行，不必等到软件可以运行之际，越早消除缺陷就越能降低开发成本；
  - 开发人员能够及时地得到同行专家的帮助和指导，无疑会加深对工作成果的理解，更好地预防缺陷，一定程度上提高了开发生产率。

软件评审与测试区别

- **观点**
  - 软件评审和软件测试的目的都是为了消除软件的缺陷，两者主要区别是
    - 前者无需运行软件，评审人员和作者把工作成果摆放在桌面上讨论；
    - 后者一定要运行软件来查找缺陷。软件评审在软件测试之前执行，尤其是在需求开发和系统设计阶段。
    - 相比而言，软件测试的工作量通常比技术评审的大，发现的缺陷也更多。
  - 在制定质量计划的时候，已经确定了本项目的主要测试活动、时间和负责人，之后再考虑软件测试的详细计划和测试用例。
  - **强调：质量人员一定要参与软件测试，只有这样他才能深入地了解软件的质量问题，而且给予开发小组强有力地帮助。**

4. 全面软件质量管理：质量保证，过程检查

- 观点
  - CMM(能力成熟度模型)和ISO9001所述的软件质量保证，实质就是过程检查，即检查软件项目的“工作过程和工作成果”是否符合既定的规范。
  - **符合规范的工作成果不见得就是高质量的，但是明显不符合规范的工作成果十有八九是质量不合格的。**
    - 例如版本控制检查
    - 再例如，机构制定了重要工作成果的文档模板（例如需求规格说明书、设计报告等），要求开发人员写的文档尽可能符合模板。如果质量人员发现开发人员写的文档与机构的模板差异非常大，那么就要搞清楚究竟是模板不合适？还是开发人员偷工减料？
  - **过程检查的要点是：找出明显不符合规范的工作过程和工作成果，及时指导开发人员纠正问题，切勿吹毛求疵或者在无关痛痒的地方查来查去。**
    - 不少机构的质量人员并没有真正理解过程检查的意义，老是对照规范，查找错别字、标点符号、排版格式等问题，迷失了方向，这样只有疲劳没有功劳，而且让开发人员很厌烦。
    - 对于中小型项目而言，过程检查工作由质量人员一个人负责就够了，约占其20%的工作量，让质量人员抽出更多的时间从事技术评审和软件测试工作。

4. 全面软件质量管理：过程检查

- **流程**
  - 过程检查计划的要点是确定主要检查项和检查时间（或频度）。
  - 质量人员在执行过程检查的时候，如果发现问题，应该立即记录下来。过程问题也是缺陷，因此最好使用缺陷跟踪工具，有助于提高过程检查的效率。
  - 质量人员首先设法在项目内部解决已经发现的质量问题，与项目成员们协商，给出解决措施。在项目内难以解决的质量问题，由上级领导给出解决措施。

5. 全面软件质量管理：缺陷跟踪工具

- **概念**
  - 如果没有缺陷跟踪工具的话，人们只好用纸张或文件去记录缺陷，不仅变更缺陷信息很麻烦，而且难以共享信息。缺陷跟踪工具就是帮助项目成员记录和跟踪缺陷用的，一般都有数据库支持，可以在局域网内运行。
  - Internet上有许多缺陷跟踪工具，大家可以免费下载使用。由于缺陷跟踪工具仅仅是一种辅助性的工具，我们没有必要太在乎该软件的功能，只要用起来方便就行。
  - 缺陷的主要属性：缺陷ID, 缺陷类型,缺陷状态,缺陷描述,相关文件,严重性,优先级,报告者,报告日期,接受者,解决方案（建议）,解决日期 。
  - 缺陷跟踪工具的常见功能：查询缺陷,添加缺陷,修改缺陷,删除,缺陷分类图,缺陷趋势图, Email

6. 全面软件质量管理：人员

- **角色职责**
  - **谁对软件质量负责？**是全员负责。任何与软件开发、管理工作相关的人员都对质量产生影响，都要对质量负责。所以人们不要把质量问题全部推出质量人员或测试人员。
  - **谁对软件质量负最大的责任？**谁的权利越大，他所负的质量责任就越大。质量人员是成天与质量打交道的人，但他个人并不对产品质量产生最大的影响，所以也不负最大的责任。
  - 质量人员的主要职责：
    - 负责制定质量计划（很重要但是工作量比较少）；
    - 负责过程检查（类似于CMM中的质量保证），约占个人工作量的20%；
    - 参与技术评审，约占个人工作量的30%；
    - 参与软件测试，约占个人工作量的30%；
    - 参与软件过程改进（面向整个机构），约占个人工作量的20%；

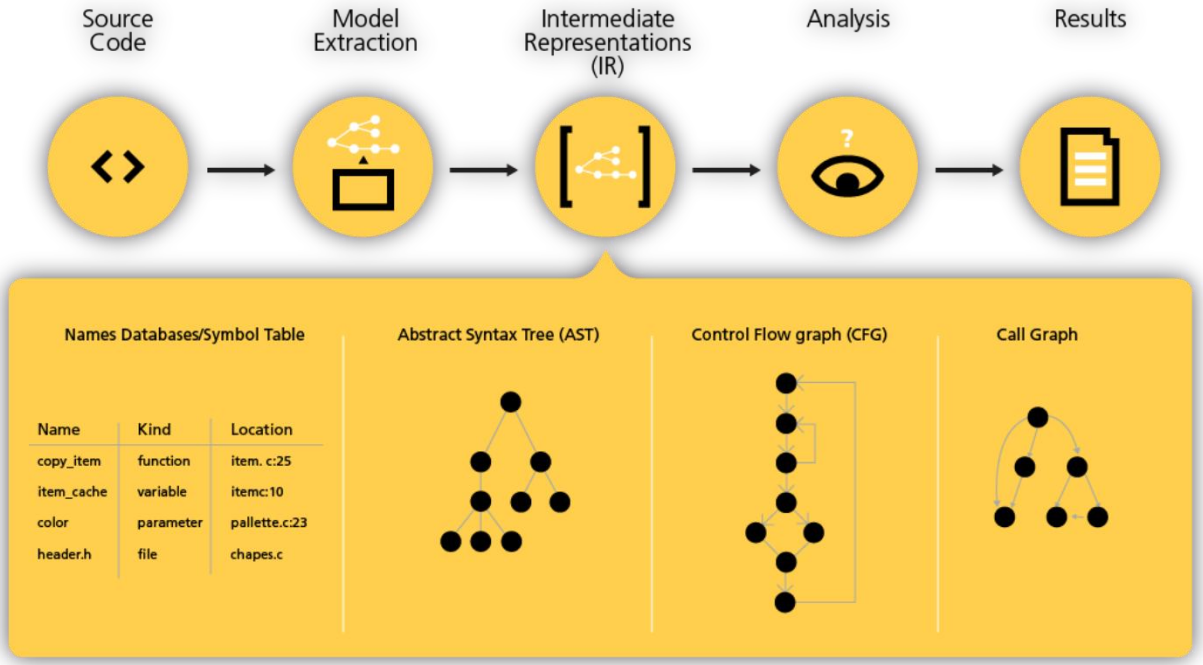
      *上述工作量的比例仅供参考，在实际应用时必须根据项目的特征而定。

2. 代码复审：静态程序分析

代码复审(Review)

- **代码复审/复查是对源代码的系统检查（同行评审）**
  - 旨在发现初始开发阶段被忽视的错误，提高整体质量；
  - 审查的形式多种多样，如结对编程、走查、正式评审。

- 结对编程
- 走查
- 正式评审会议
- 自动化评审



代码复审(Review)

- **代码复审：代码是否在“代码规范”的框架内正确地解决了问题**

名 称	形 式	目 的
自我复审	自己 vs. 自己	用同伴复审的标准来要求自己。不一定最有效，因为开发者对自己总是过于自信。如果能持之以恒，则对个人有很大好处
同伴复审	复审者 vs. 开发者	简便易行
团队复审	团队 vs. 开发者	有比较严格的规定和流程，用于关键的代码，以及复审后不再更新的代码。 覆盖率高——有很多双眼睛盯着程序。但是有可能效率不高（全体人员都要到会）

From 《现代软件工程讲义》

代码复审的目的

- **找出代码的错误：**
  - 编码错误，比如一些能碰巧骗过编译器的错误。
  - 不符合项目组的代码规范的地方。
- **发现逻辑错误，程序可以编译通过，但是代码的逻辑是错的。**
- **发现算法错误，比如使用的算法不够优化。**
- **发现潜在的错误和回归性错误——当前的修改导致以前修复的缺陷又重新出现。**
- **发现可能改进的地方。**
- **教育(互相教育)开发人员，传授经验，让更多的成员熟悉项目各部分的代码，同时熟悉和应用领域相关的实际知识。**
- **代码复审中的提问与回应能帮助团队成员互相了解，就像练武之人互相观摩点评一样。在一个新成员加入一个团队的时候，代码复审能非常有效地帮助新成员了解团队的开发策略、编程风格及工作流程。**

From 《现代软件工程讲义》

代码复审的步骤

- **在复审前——**
  - 代码必须成功地编译。
  - 程序员必须测试过代码。
- **“你这样修改了之后，有没有别的功能会受影响？”**
- **“项目中还有别的地方需要类似的修改么？”**
- **“有没有留下足够的说明，让将来维护代码时不会出现问题？”**
- **“对于这样的修改，有没有别的成员需要告知？”**
- **“导致问题的根本原因是什么？我们以后如何能自动避免这样的情况再次出现？”**

From 《现代软件工程讲义》

代码复审的核查表：以子程序为例

- **总体问题：**
  - 创建子程序的理由充分吗？
  - 如果把一个子程序中的某些部分独立成另一个子程序会更好的话，你这样做了吗？
  - 是否用了明显而清楚的动宾词组对过程进行命名？是否是用返回值的描述来命名函数？
  - 子程序的名称是否描述了它做的所有工作？
  - 子程序的内聚性是不是很强的功能内聚性？它只做一件工作并做得很好吗？
  - 子程序的耦合是不是松散的？两个子程序之间的联系是不是小规模、密切、可见和灵活的？
  - 子程序的长度是不是它的功能和逻辑自然地决定的：而不是由人为标准决定的？

代码复审的核查表：以子程序为例

- **参数传递问题**
  - 形式参数与实际参数匹配吗？
  - 子程序中参数的排列合理吗？与相似子程序中的参数排列顺序匹配吗？
  - 接口假设说明了吗？
  - 子程序中参数个数是不是7 个或者更少，
  - 是否只传递了结构化变量中另一个子程序用得到的部分？
  - 是否用到了每一个输入参数？
  - 是否用到了每一个输出参数?
  - 如果子程序是一函数，是否在所有情况下它都会返回一个值？

代码复审的核查表：以子程序为例

- **防错性编程**
  - 断言是否用于验证假设？
  - 子程序对于非法输入数据进行防护了吗？
  - 子程序是否能很好地进行程序终止？
  - 子程序是否能很好地处理修改情况？
  - 是否不用很麻烦地启用或去掉调试帮助？
  - 是否信息隐蔽、松散耦合，以及使用“防火墙”数据检查，以使得它不影响子程序之外的代码？
  - 子程序是否检查返回值？
  - 产品代码中的防错性代码是否帮助用户，而不是程序员？

走查(Walkthrough)

- **走查(Walkthrough)**
  - 由设计人员或编程人员组成一个走查小组，通过阅读一段文档或代码，并进行提问和讨论，从而发现可能存在的缺陷、遗漏和矛盾的地方。
  - 类型：设计走查、代码走查。
- **走查过程**
  - 与评审过程类似，即先把材料先发给走查小组每个成员，让他们认真研究程序，然后再开会；
  - 与评审的区别：评审通常是简单地读程序或对照错误检查表进行检查；走查则是按照所提交的测试用例，人工模仿计算机运行一遍，并记录跟踪情况。

走查(Walkthrough)

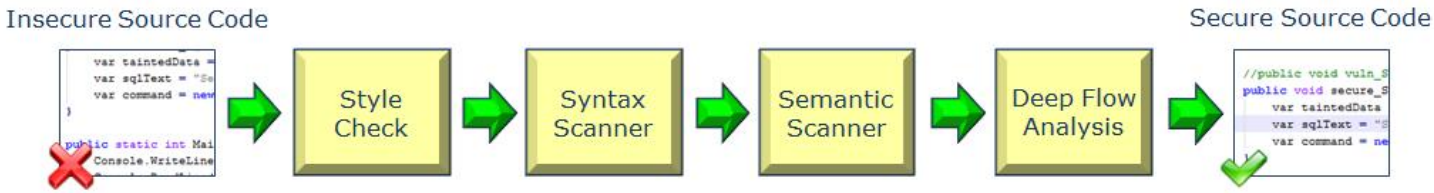
- **走查是开发者的一次友好的会议，需要仔细规划，并有明确的目的、日程、持续时间和参与人员，许多小组以星期为单位走查。**
  - 走查前几天：召集人将收集的一些要在会上审查的材料(模型、文档、程序代码等)分发给参与者，参与者研究这些材料并在会议之前提交意见。
  - 会议期间：召集人提出大家的意见并对每一项进行讨论。
- **会议时间比较短，一般2-3 小时。**
- **会议的目的是查明问题，而不是干扰开发者。**
- **会议的思想是确认问题的存在，甚至不必去谋求问题的解决。**
  - 会后：将问题分发给相应人员进行解决。

Lightweight code review 轻量级的代码评审

- 轻量级代码评审通常比正式的代码检查要求更少的开销，但如果正确完成，可以同样有效。
- 轻量级评审通常作为正常开发过程的一部分进行：
  - *Over-the-shoulder* – one developer looks over the author's shoulder as the latter walks through the code. 实时评审
  - *Email pass-around* – source code management system emails code reviewers automatically after checkin is made. 电子邮件
  - *Pair programming* – two authors develop code together at the same workstation, as it is common in Extreme Programming. 结对编程
  - *Tool-assisted code review* – authors and reviewers use software tools, informal ones such as pastebins and IRC, or specialized tools designed for peer code review. 工具辅助

使用静态代码分析发现潜在bug

- **静态代码分析(static code analysis)：在代码构建过程中帮助开发人员快速、有效的定位代码缺陷；**
  - 无需运行被测代码，仅通过分析或检查源程序的语法、结构、过程、接口等来检查程序的正确性，找出代码隐藏的错误和缺陷，如参数不匹配，有歧义的嵌套语句，错误的递归，非法计算，可能出现的空指针引用等等。
- **“30~70%的代码逻辑设计和编码缺陷是可以通过静态代码分析来发现和修复的”。**
- **主要技术：缺陷模式匹配、类型推断、模型检查、数据流分析等。**
- **常用的Java静态代码分析工具：**
  - Checkstyle
  - SpotBugs
  - SonarQube



3. 性能分析：动态程序分析

程序性能分析

- 性能分析(performance analysis，也称为profiling)：以收集程序运行时信息为手段研究程序行为的分析方法，是一种动态程序分析的方法。
- 这种方法与静态代码分析相对，主要测量程序的空间或时间复杂度、特定指令的使用情形、函数调用的频率及运行时间等。
- 性能分析的目的在于决定程序的哪个部分应该被优化，从而提高程序的速度或者内存使用效率。
- 性能分析可以由程序的源代码或是可执行文件进行，一般会使用称为性能分析工具(profiler)的工具进行。

程序性能分析的两种方法

- **抽样(Sampling)**
  - 当程序运行时，分析工具时不时看一看这个程序运行在哪一个函数内，并记录下来，程序结束后，工具就会得出一个关于程序运行时间分布的大致的印象。这种方法的优点是不需要改动程序，运行较快，可以很快地找到瓶颈。但是不能得出精确的数据，代码中的调用关系(CallTree)也不能准确表示。
- **代码注入(Instrumentation)**
  - 将检测的代码加入到每一个函数中，这样程序的一举一动都被记录在案，程序的各个效能数据都可以被精准地测量。这一方法的缺点是程序的运行时间会大大加长，还会产生很大的数据文件，数据分析的时间也相应增加。同时，注入的代码也影响了程序真实的运行情况。
- **一般的做法是：先用抽样的方法找到效能瓶颈所在，然后对特定的模块用代码注入的方法进行详细分析。**

From 《现代软件工程讲义》

关于性能分析

- 性能测试→分析→改进→性能测试
- 先Debug，再性能分析
- 先保证正确性，再提高效能

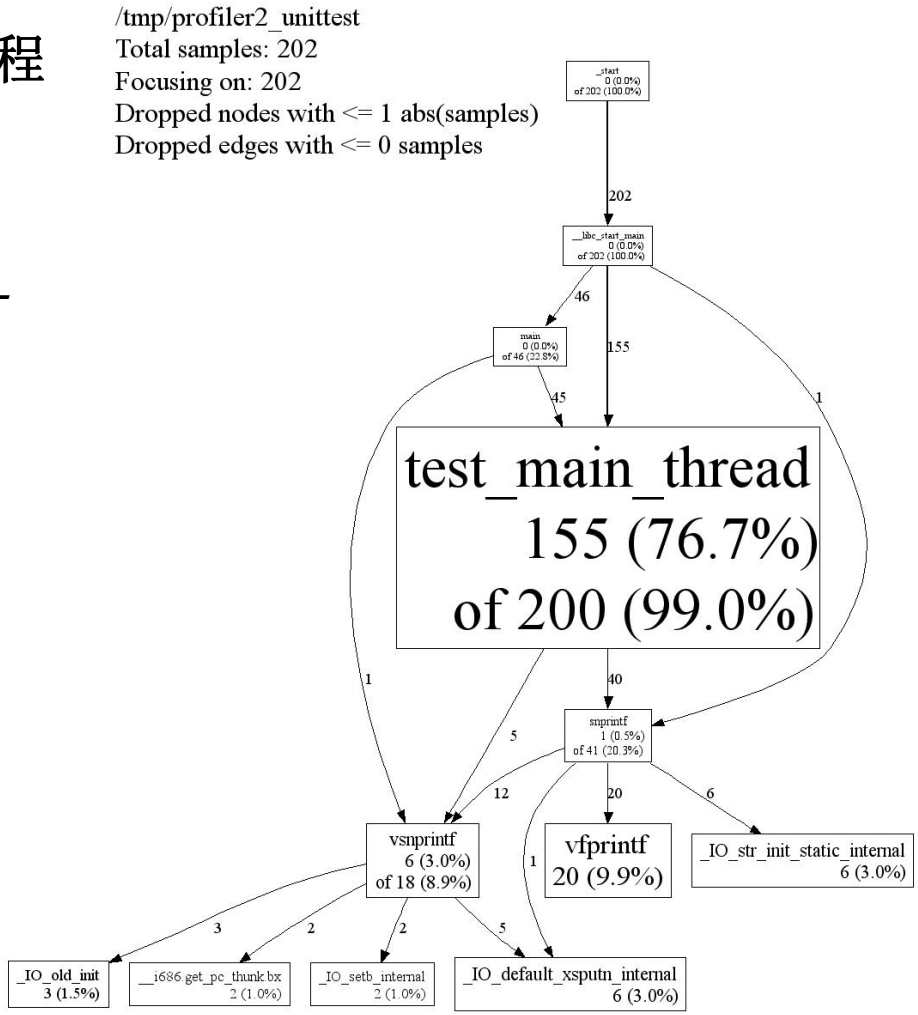
- 是否会用性能分析工具来提高程序质量是一个优秀程序员的标志之一

性能分析的工具

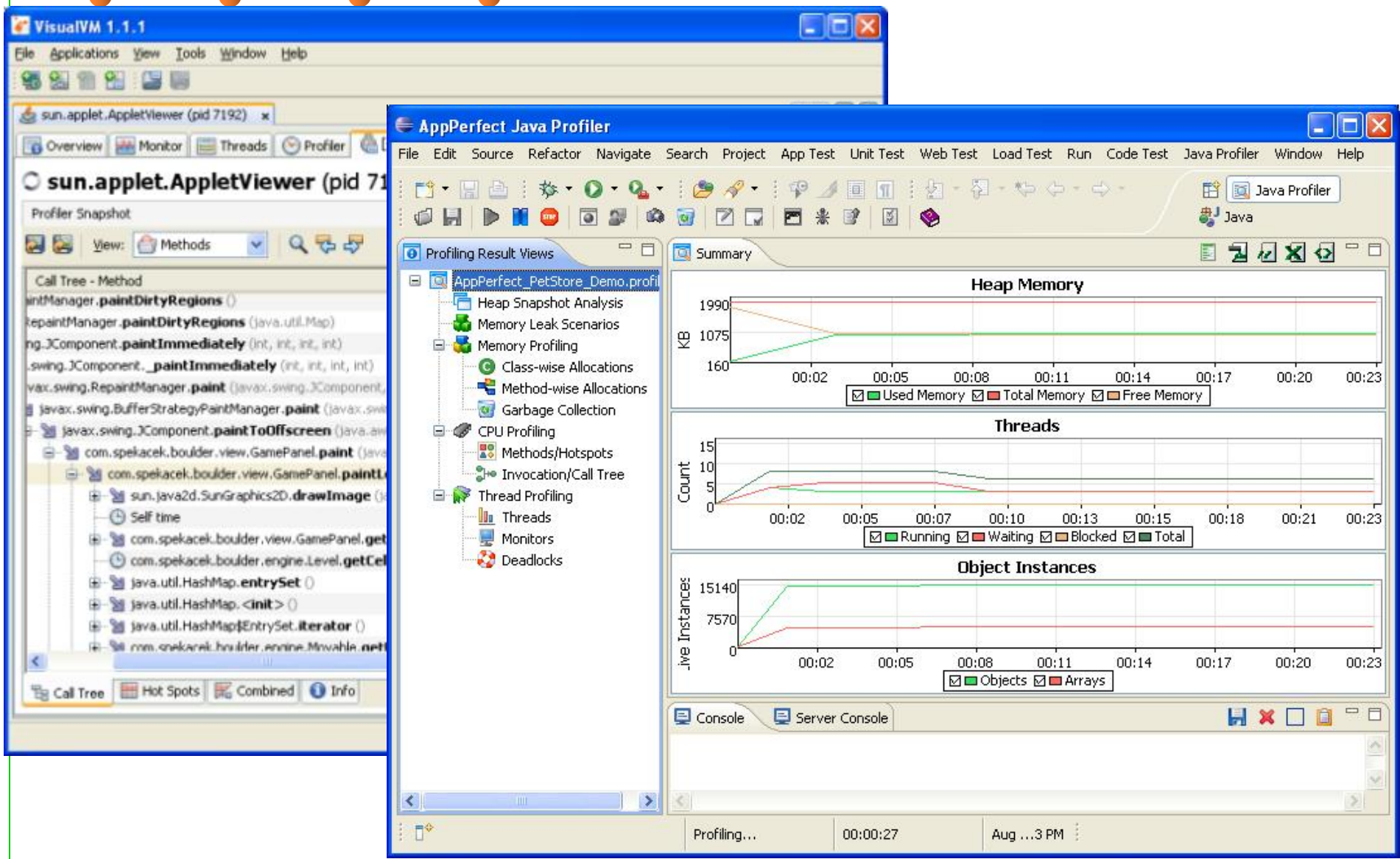
- 不同的编程语言有不同的性能分析工具；
- http://en.wikipedia.org/wiki/List_of_performance_analysis_tools进行了一个很好的汇集；
- 常用：
  - Google提供的gperftools，面向C/C++等
  - Visual Studio Team System Profiler
  - Test and Performance Tools Platform (TPTP)，Memory Analyzer (MAT)，Eclipse/Java
  - cProfile, profile和pstats，　Python

以gperftools（C/C++）为例

- http://code.google.com/p/gperftools
- **Google提供的一套工具，用于分析程序性能，找到程序的性能瓶颈。**
  - malloc free内存分配器tcmalloc；
  - 堆内存检测和内存泄漏分析工具heap-profiler和heap-checker；
  - CPU性能监测工具CPU-profiler。



Dynamic code analysis / profiling



4. 软件测试基础

1. 软件测试的概念

- 传统：测试是一种旨在评估一个程序或系统的属性或能力，确定它是否符合其所需结果的活动。

- **Myers**：测试是为了发现错误而执行一个程序或系统的过程。

  明确提出了“在程序中寻找错误”是测试的目的。

- **IEEE**：测试是使用人工和自动手段来运行或检测某个系统的过程，其目的在于检验系统是否满足规定的需求或弄清预期结果与实际结果之间的差别。

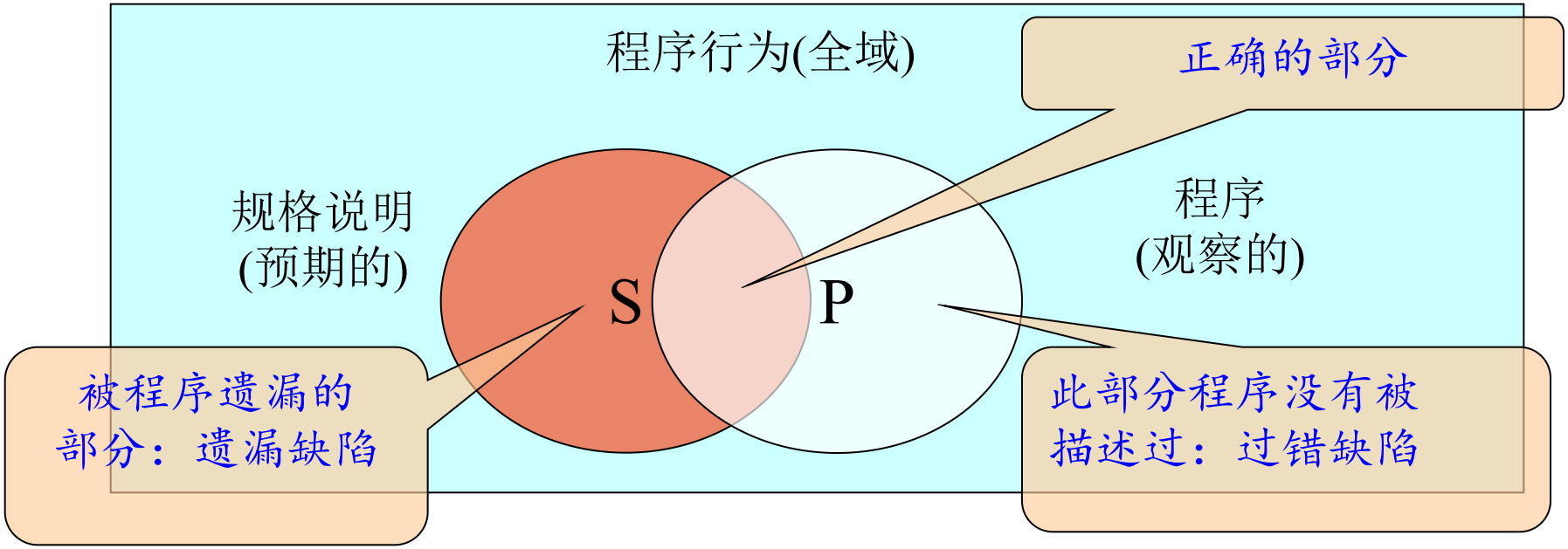
  该定义明确提出了软件测试以“检验是否满足需求”为目标。

1. 软件测试的概念

- **软件测试是更为广泛主题验证与确认(Verification and Validation, V&V)的一部分**
  - 验证：我们在正确地构造产品吗？ 强调采用正确的手段和过程
  - 确认：我们在构造正确的产品吗？ 强调结果正确

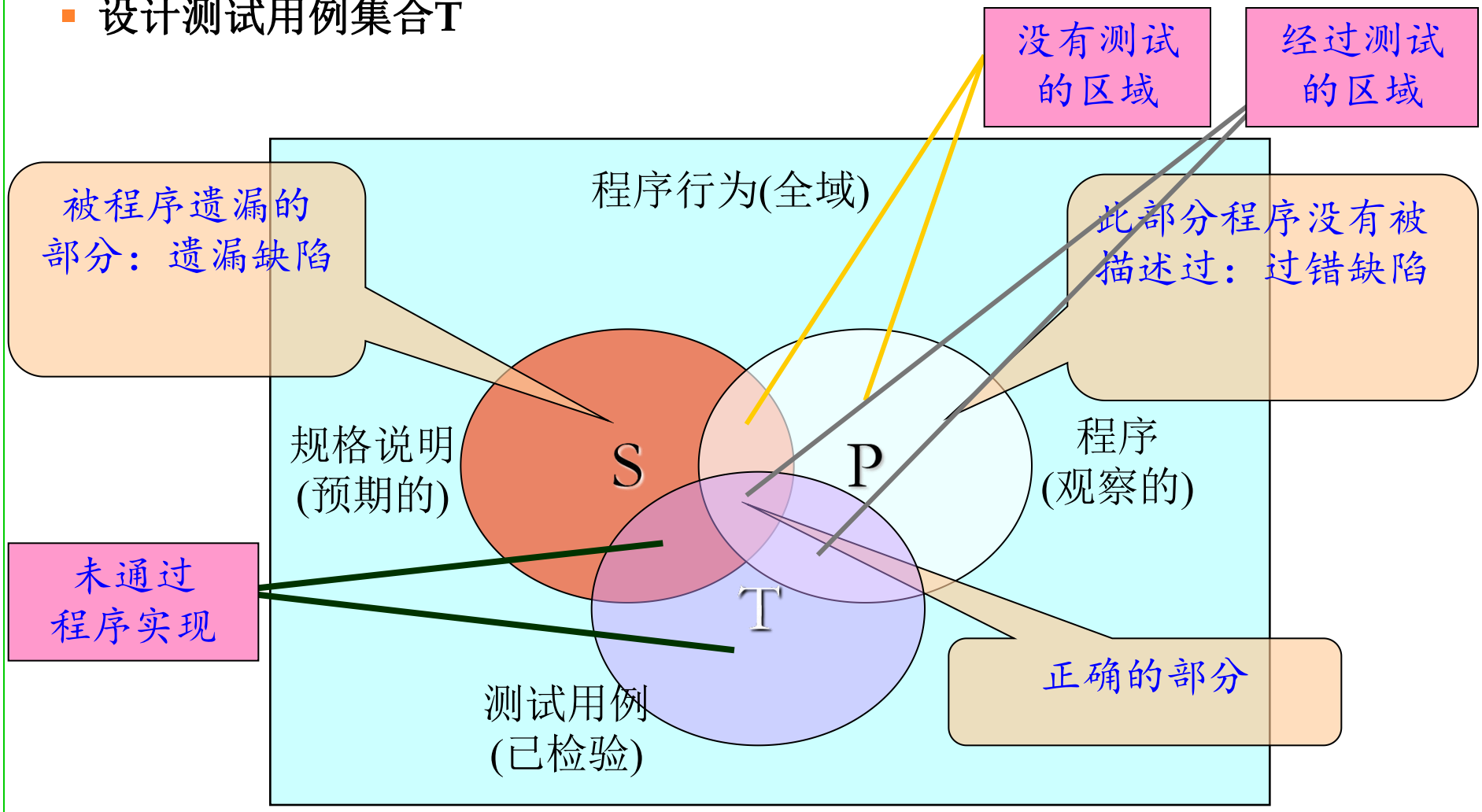
用Venn Diagram来理解测试

- **考虑一个程序行为全域，给定一段程序及其规格说明**
  - 集合S是所描述的行为；
  - 集合P是用程序实现的行为；



用Venn Diagram来理解测试

- 设计测试用例集合T



软件测试的目标

- 在程序交付测试之前，大多数程序员可以找到和纠正超过**99%**的错误；
- 在交付测试的程序中，每**100**条可执行语句的平均错误数量是**1-3**个；
- 软件测试的目的就是找出剩下的**1%**的错误。

- **Glen Myers关于软件测试目的提出以下观点：**
  - 测试是为了**发现错误**而执行程序的过程
  - 测试是为了证明“程序有错”，而**无法证明**“程序正确”
  - 一个好的测试用例在于**能够发现**至今未发现的错误
  - 一个成功的测试是**发现了**至今未发现的错误的测试

2. 软件测试的原则

- 应当把“**尽早的和不断的测试**”作为软件开发者的座右铭
- **程序员应避免检查自己的程序**
- 测试**从小规模开始，逐渐扩到大规模**
- 设计**测试用例**时，**应包括合理的输入和不合理的输入，以及各种边界条件**，特殊情况下要制造极端状态和意外状态
- 充分**注意测试中的聚集现象**：测试中发现的80%的错误，可能由程序的20%功能所造成的
- 对测试错误结果一定**要有一个确认过程**
- **制定严格的测试计划**，排除测试的随意性
- **注意回归测试的关联性**，往往修改一个错误会引起更多错误
- **妥善保存一切测试过程文档**，测试重现往往要靠测试文档

3. 软件测试文档

- **测试计划(Test Plan)**
  - 测试计划是测试工作的指导性文档，规定测试活动的范围、方法、资源和进度；明确正在测试的项目、要测试的特性、要执行的测试任务、每个任务的负责人，以及与计划相关的风险。
  - 主要内容：测试目标、测试方法、测试范围、测试资源、测试环境和工具、测试体系结构、测试进度表
- **测试规范(Test Specification)**
- **测试用例(Test Case)**
- **缺陷报告(Bug Report)**

3. 软件测试文档

- 测试计划(Test Plan)
- 测试规范(Test Specification)
  - 测试规范是从整体上规定测试案例的运行环境、测试方法、生成步骤、执行步骤以及调试和验证的步骤。
  - 主要内容：系统运行环境、总体测试方法、测试用例的生成步骤、测试用例的执行步骤、调试和验证
- 测试用例(Test Case)
- 缺陷报告(Bug Report)

3. 软件测试文档

- 测试计划(Test Plan)
- 测试规范(Test Specification)
- 测试用例(Test Case)
  - 测试用例是数据输入和期望结果组成的对，其中“输入”是对被测软件接收外界数据的描述，“期望结果”是对于相应输入软件应该出现的输出结果的描述，测试用例还应明确指出使用具体测试案例产生的测试程序的任何限制。
  - 测试用例可以被组织成一个测试系列，即为实现某个特定的测试目的而设计的一组测试用例。例如，一部分测试用例用来测试系统的兼容性，另一部分是用来测试系统在特定的环境中，系统的典型应用是否能够很好地运作。
- 缺陷报告(Bug Report)

3. 软件测试文档

- 测试计划(Test Plan)
- 测试规范(Test Specification)
- 测试用例(Test Case)
- 缺陷报告(Bug Report)
  - 缺陷报告是编写在需要调查研究的测试过程期间发生的任何事件，简而言之，就是记录软件缺陷。
  - 主要内容：缺陷编号、题目、状态、提出、解决、所属项目、测试环境、缺限报告步骤、期待结果、附件
  - 在报告缺陷时，一般要讲明缺陷的严重性和优先级。
    - 严重性表示缺陷的恶劣程度，反映其对产品和用户的影响。
    - 优先级表示修复缺陷的重要程度和应该何时修复。

4. 软件测试组织

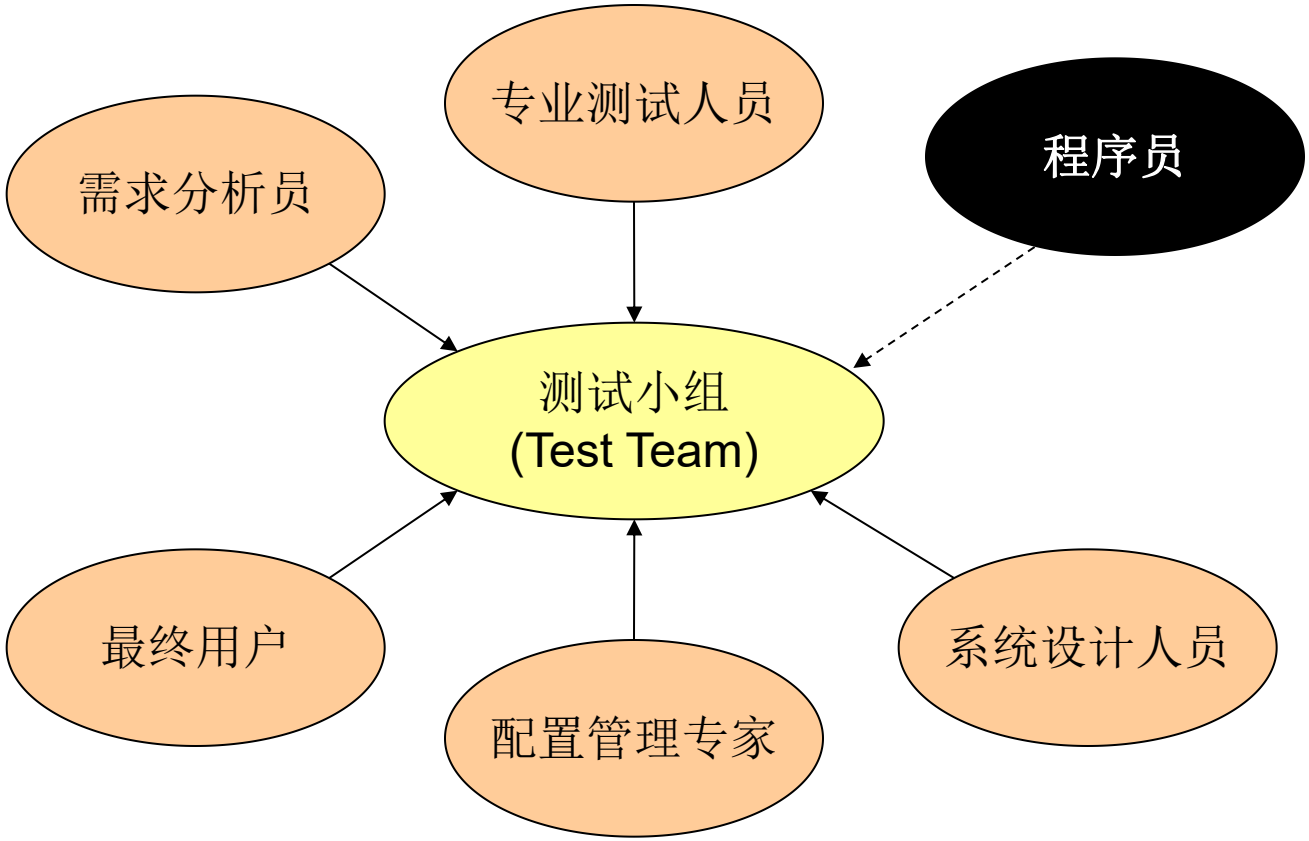
开发者

- 系统的构建者，理解系统
- 测试是“温和”的，试图证明正确性
- 发布驱动

独立测试组

- 必须学会系统
- 测试是破坏性的，试图证明错误存在
- 质量驱动

4. 软件测试组织



软件测试人员的素质要求

- **沟通能力**
  - 理想的测试人员必须能与测试涉及到的所有人进行沟通，具有与技术人员(开发者)和非技术人员(客户、管理人员)的交流能力。
- **移情能力**
  - 和系统开发有关的所有人员(用户、开发者、管理者)都处于一种既关心又担心的状态中。测试人员必须和每一类人打交道，因此需要对每一类人都具有足够的理解和同情，从而将测试人员与相关人员之间的冲突和对抗减少到最低程度。
- **技术能力**
  - 一个测试人员必须既明白被测软件系统的概念又要会使用工程中的那些工具，最好有几年以上的编程经验，从而有助于对软件开发过程的较深入理解。

软件测试人员的素质要求

- **自信心**
  - 开发人员指责测试人员出了错是常有的事，测试人员必须对自己的观点有足够的自信心。
- **外交能力**
  - 当你告诉某人他出了错时，就必须使用一些外交方法，机智老练和外交手法有助于维护与开发人员之间的协作关系。
- **幽默感**
  - 在遇到狡辩的情况下，一个幽默的批评将是很有帮助的。
- **很强的记忆力**
  - 理想的测试人员应该有能力将以前曾经遇到过的类似的错误从记忆深处挖掘出来，这一能力在测试过程中的价值是无法衡量的。

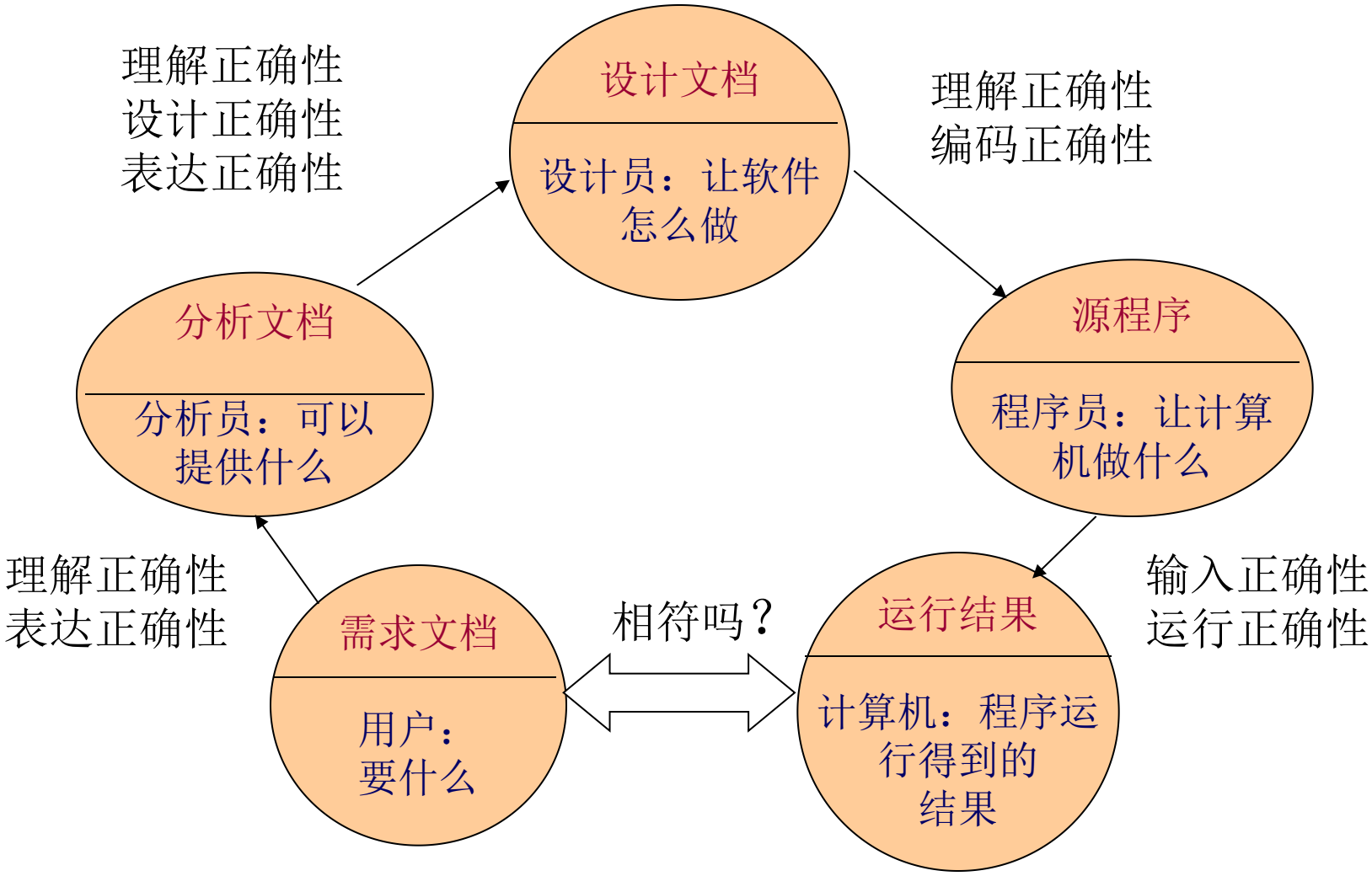
软件测试人员的素质要求

- **耐心**
  - 一些质量保证工作需要难以置信的耐心，有时需要花费惊人的时间去分离、识别一个错误。
- **怀疑精神**
  - 开发人员会尽他们最大的努力将所有的错误解释过去，测试人员必须听每个人的说明，但他必须保持怀疑直到他自己看过以后。
- **自我督促**
  - 干测试工作很容易变得懒散，只有那些具有自我督促能力的人才能够使自己每天正常地工作。
- **洞察力**
  - 一个好的测试人员具有“测试是为了破坏”的观点、捕获用户观点的能力、强烈的质量追求、对细节的关注能力。

5. 软件测试的对象

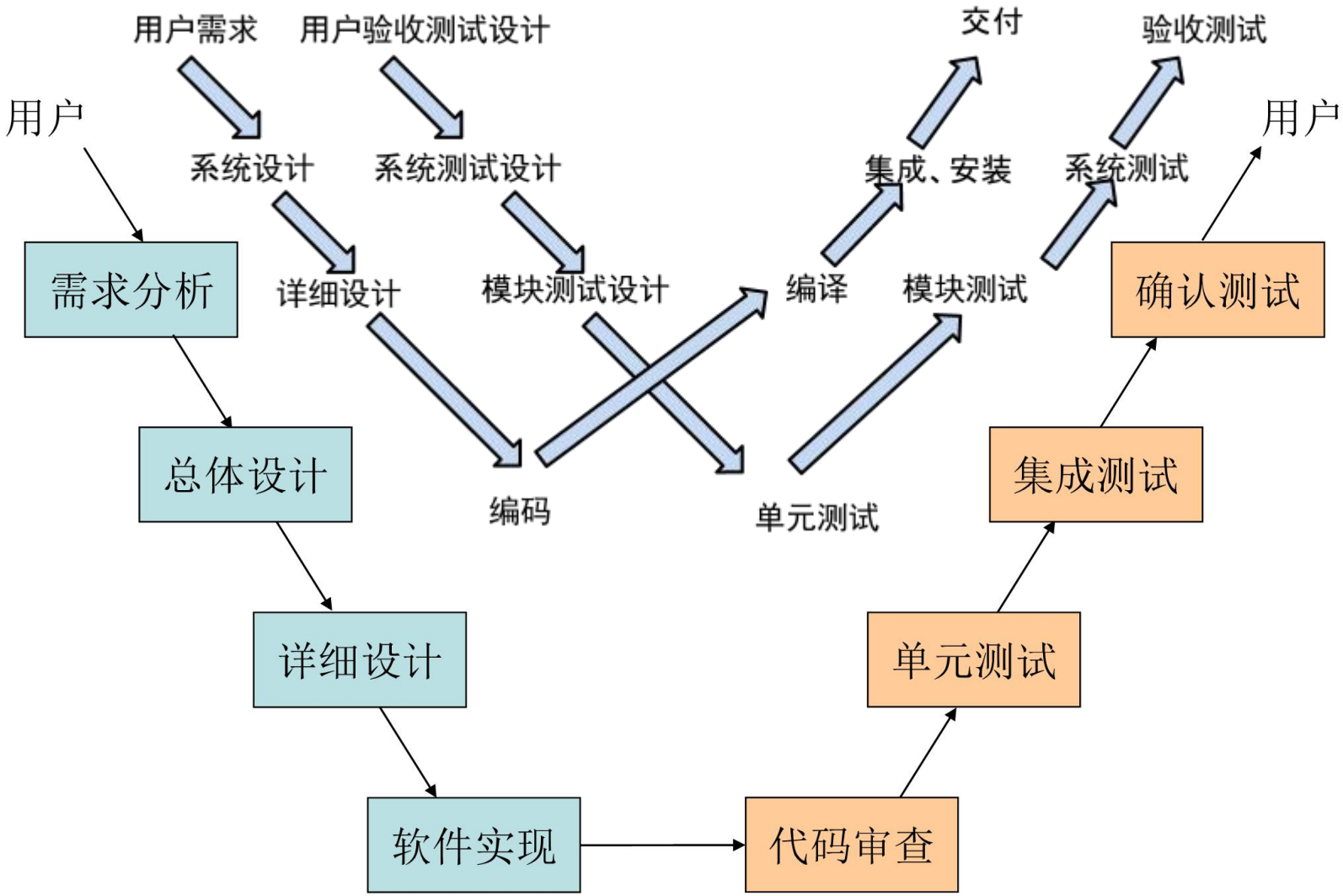
- 软件测试并不等于程序测试，应贯穿于软件定义与开发的各个阶段。
- 测试对象包括：
  - 需求规格说明
  - 设计规格说明
  - 源程序

软件测试对象之间关系

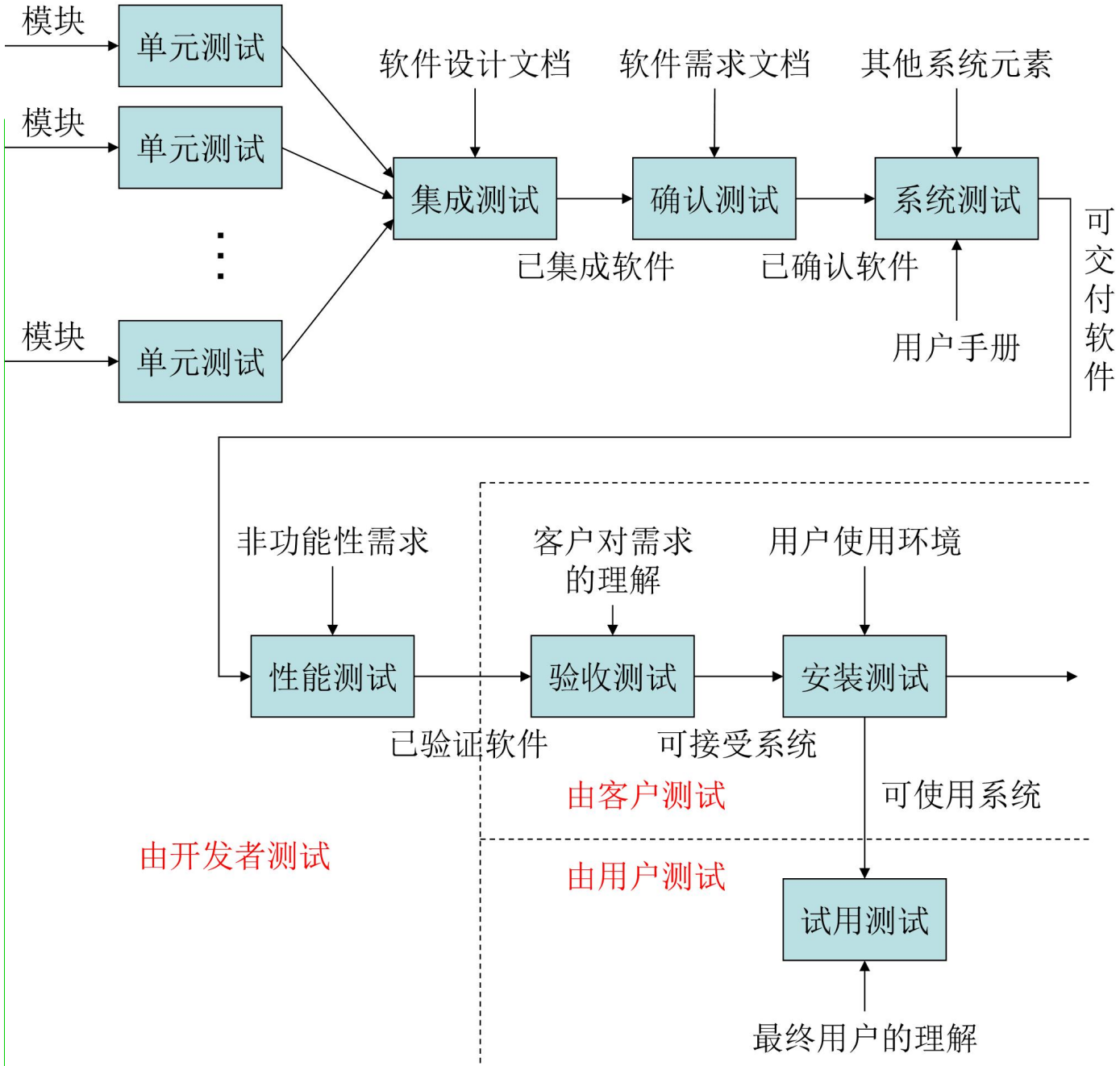

理解正确性
设计正确性
表达正确性
设计文档
设计员：让软件怎么做
理解正确性
编码正确性
分析文档
分析员：可以提供什么
源程序
程序员：让计算机做什么
理解正确性
表达正确性
需求文档
用户：要什么
相符吗？
运行结果
计算机：程序运行得到的结果
输入正确性
运行正确性

5. 软件测试过程

软件测试的V模型


用户需求
用户验收测试设计
交付
验收测试
用户
系统设计
系统测试设计
集成、安装
系统测试
用户
需求分析
详细设计
模块测试设计
编译
模块测试
确认测试
总体设计
编码
单元测试
集成测试
详细设计
单元测试
软件实现
代码审查

软件测试活动


模块
单元测试
模块
单元测试
模块
单元测试
软件设计文档
软件需求文档
其他系统元素
集成测试
确认测试
系统测试
已集成软件
已确认软件
用户手册
可交付软件
非功能性需求
客户对需求的理解
用户使用环境
性能测试
验收测试
安装测试
已验证软件
可接受系统
由客户测试
可使用系统
由开发者测试
由用户测试
试用测试
最终用户的理解

6. 软件测试方法的分类

软件测试方法分类

- 按实施步骤分：
  - 单元测试　Unit Testing
  - 集成测试　Integration Testing
  - 确认测试　Validation Testing
  - 系统测试　System Testing

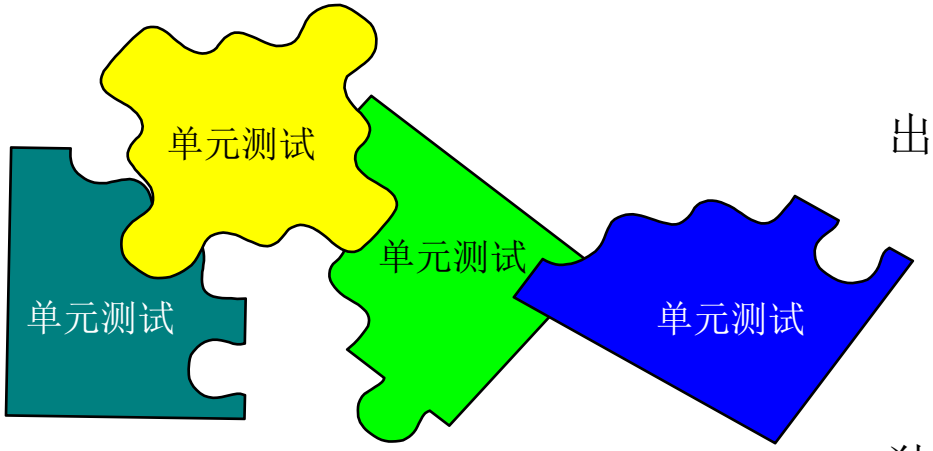
- 按使用的测试技术分：
  - 静态测试：走查/评审
  - 动态测试：白盒/黑盒
- 按软件组装策略分:
  - 非增量测试：整体集成
  - 增量测试：自顶向下、自底向上、三明治

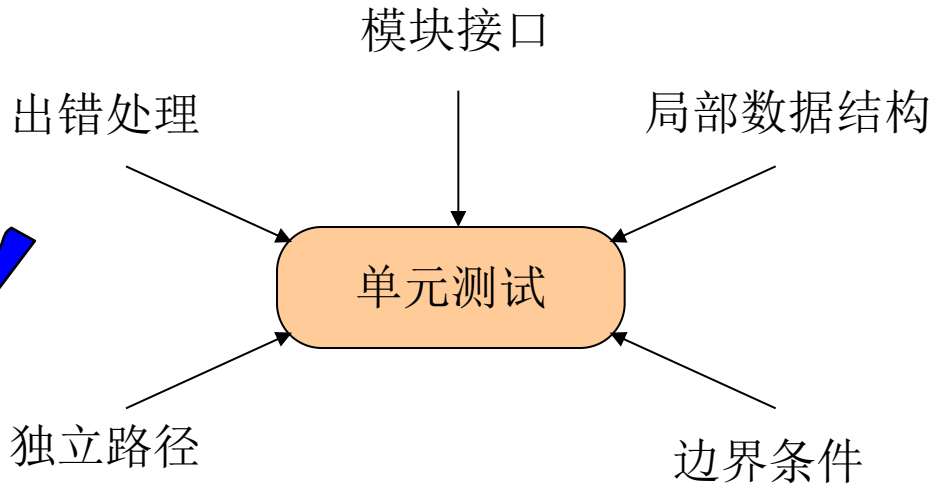
1. 单元测试(Unit Testing)

- **单元测试目的**
  - 验证开发人员所书写的代码是否可以按照其所设想的方式执行而产出符合预期值的结果，确保产生符合需求的可靠程序单元。
- **单元测试范围**
  - 单元测试是对软件**基本组成单元（构件或模块）**进行的测试
  - 单元测试侧重于构件中的内部处理逻辑和数据结构（依据详细设计）
  - **结构化程序单元是模块，面向对象程序单元是类**
- **单元测试规程**
  - 单元测试通常被认为是编码阶段的附属工作，单元测试可以在编码开始之前或源代码生成之后完成
  - 单元测试一般由编写该单元代码的开发人员执行，该人员负责设计和运行一系列的测试以确保该单元符合需求。

单元测试

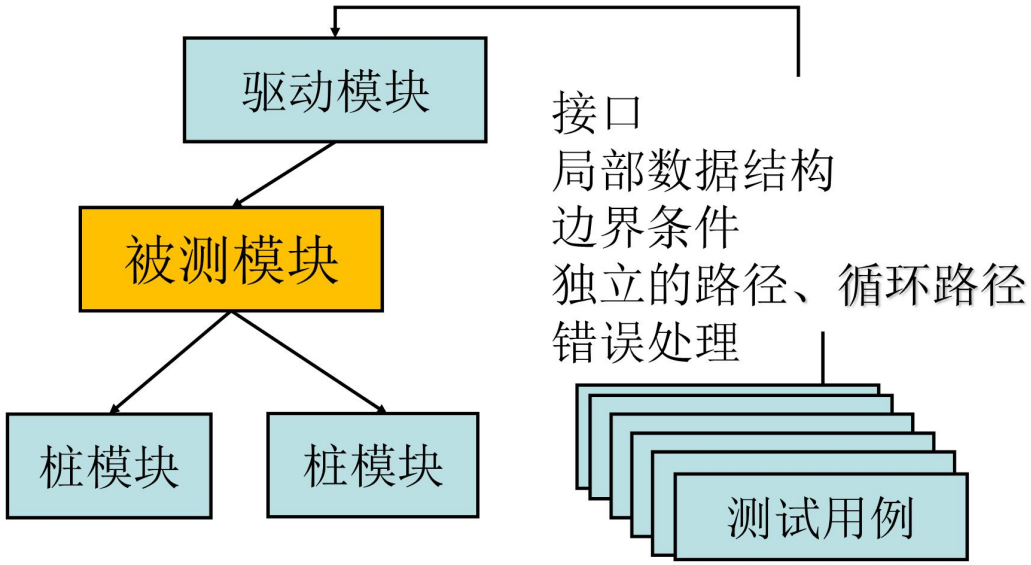
- **单元测试内容**
  - 接口
  - 局部数据结构
  - 独立路径
  - 边界条件
  - 错误处理路径



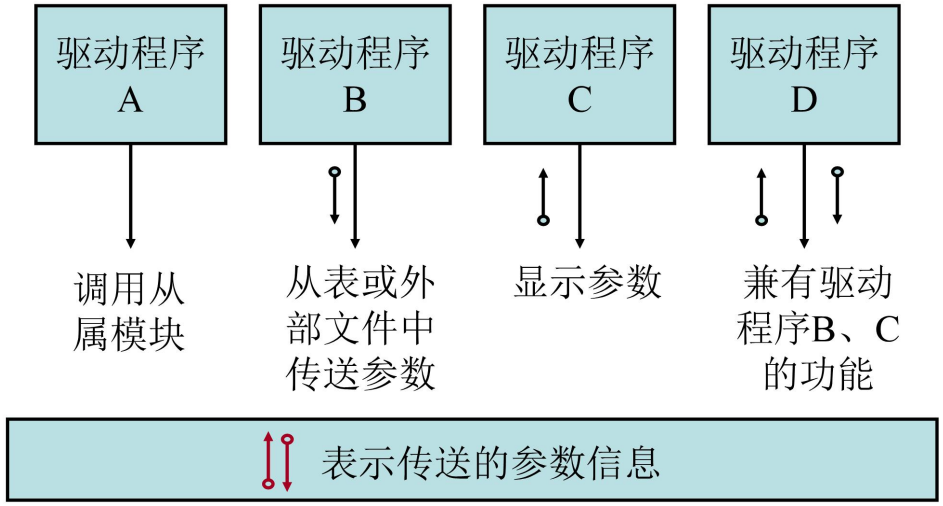


单元测试的环境

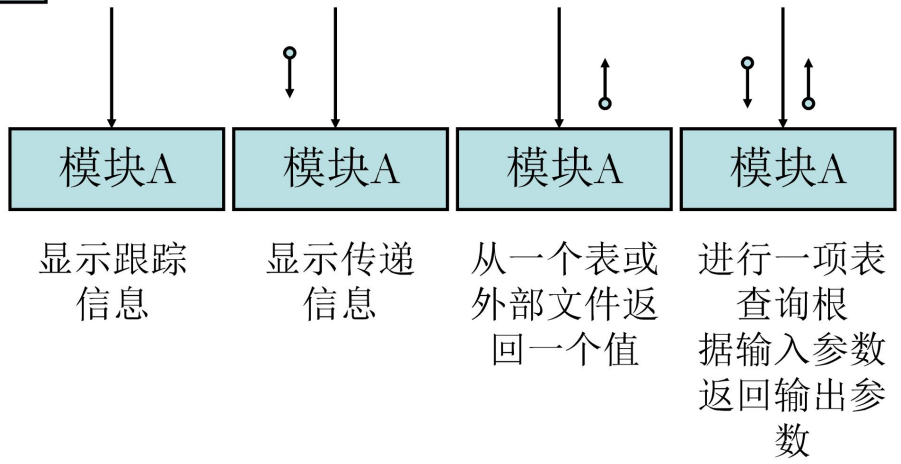
- **单元测试环境**
  - **驱动模块(driver)：** 模拟被测模块的上一级模块，接收测试数据，把这些数据传送给所测模块，最后再输出实际测试结果；
  - **桩模块(stub)：** 模拟被测单元需调用的其他函数接口，模拟实现子函数的某些功能。



单元测试的驱动模块/桩模块

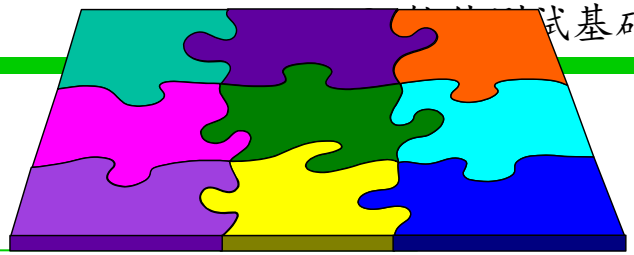


驱动模块：为被测模块提供数据



桩模块：只做少量的数据操作

2. 集成测试



- **每个模块都能单独工作，但是集成在一起却不能工作？ Why？**

  ——模块通过接口相互调用时会引入很多新问题...

- **集成测试(Integration Testing)**
  - 在单元测试的基础上，将所有模块按照总体设计的要求组装成为子系统或系统进行的测试。
  - 集成测试是构造软件体系结构的系统化技术，同时也是进行一些旨在发现与接口相关的错误的测试。
  - **结构化集成测试针对调用关系测试，面向对象针对依赖关系测试**

集成测试策略

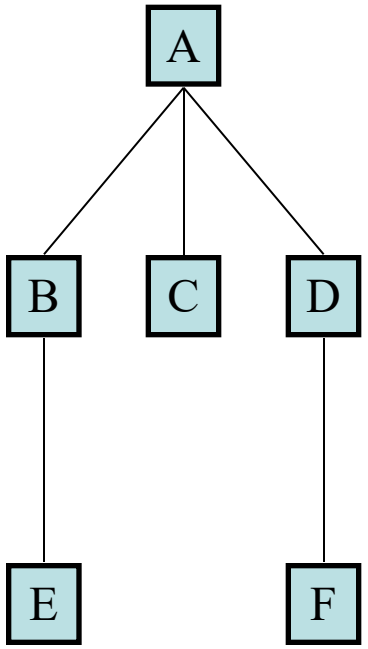
- **集成测试策略**
  - **基于层次的集成**：自顶向下与自底向上
  - **基于功能的集成**：按照功能的优先级逐步将模块加入系统中
  - **基于进度的集成**：把最早可获得的代码进行集成
  - **基于使用的集成**：通过类的使用关系进行集成

集成测试的目标

- **集成测试考虑的问题：**
  - 模块接口的数据是否会丢失
  - 组合后的子功能，能否达到预期要求的父功能
  - 模块的功能是否会相互产生不利的影响
  - 全局数据结构是否有问题
  - 模块的误差累积是否会放大
  - 单个模块的错误是否会导致数据库错误

集成测试的方法：整体集成

- **整体集成方式(非增量式集成)**
  - 把所有模块按设计要求一次全部组装起来，然后进行整体测试
  - “一步到位”的集成方式



- **例如：**
  - Test A (with stubs for B, C, D)
  - Test B (with driver for A and stub for E)
  - Test C (with driver for A)
  - Test D (with driver for A and stub for F)
  - Test E (with driver for B)
  - Test F (with driver for D)
  - Test (A, B, C, D, E, F)

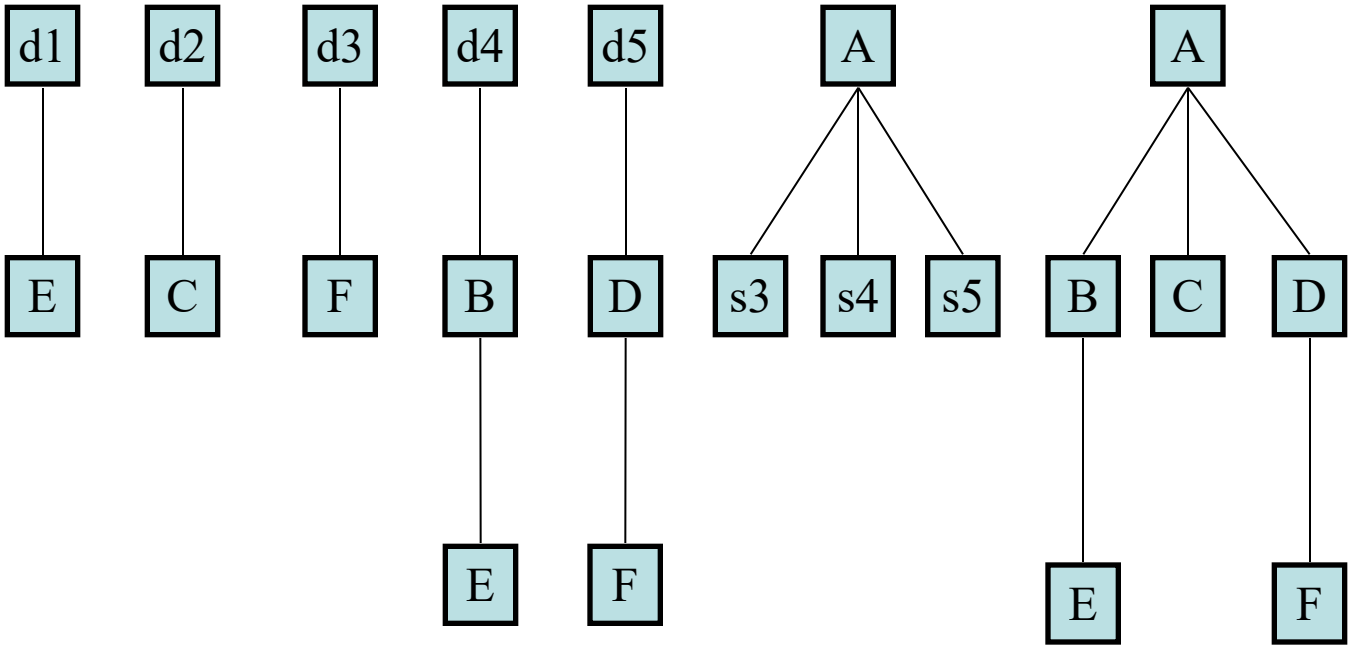
集成测试的方法：整体集成

- **优点：**
  - 效率高，所需人力资源少；
  - 测试用例数目少，工作量低；
  - 简单，易行；

- **缺点：**
  - 可能发现大量的错误，难以进行错误定位和修改；
  - 即使测试通过，也会遗漏很多错误；
  - 测试和修改过程中，新旧错误混杂，带来调试困难；

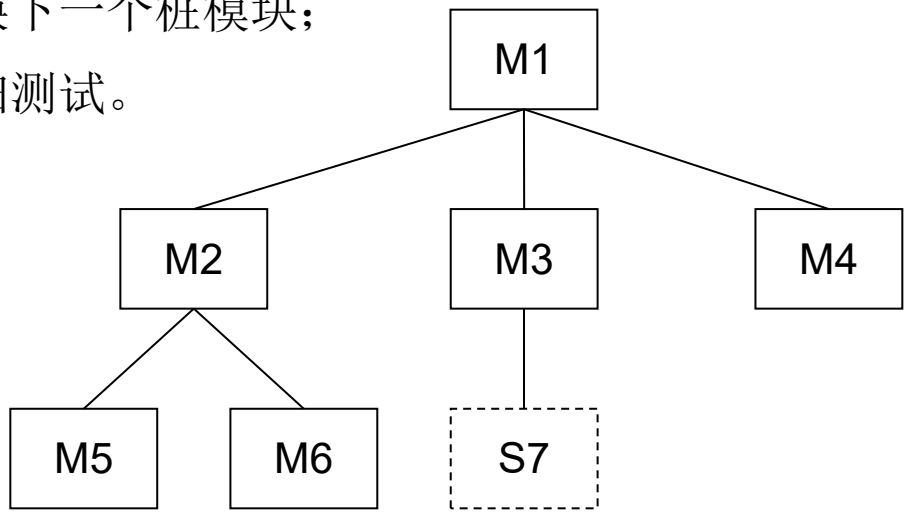
集成测试的方法：增量式集成

- **增量式集成测试方法：**
  - 逐步将新模块加入并测试

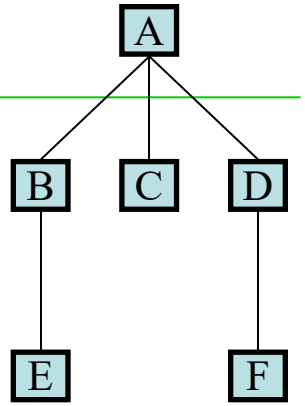


(1) 自顶向下的增量集成

- **自顶向下的集成测试：从主控模块开始，按软件的控制层次结构，以深度优先或广度优先的策略，逐步把各个模块集成在一起。**
- **具体步骤：**
  - 1：以主控模块作为测试驱动模块，把对主控模块进行单元测试时所引入的所有桩模块用实际模块代替；
  - 2：依据所选的集成策略(深度优先、广度优先)，每次只替代一个桩模块；
  - 3：每集成一个模块立即测试一遍；
  - 4：只有每组测试完成后，才着手替换下一个桩模块；
  - 5：为避免引入新错误，不断进行回归测试。



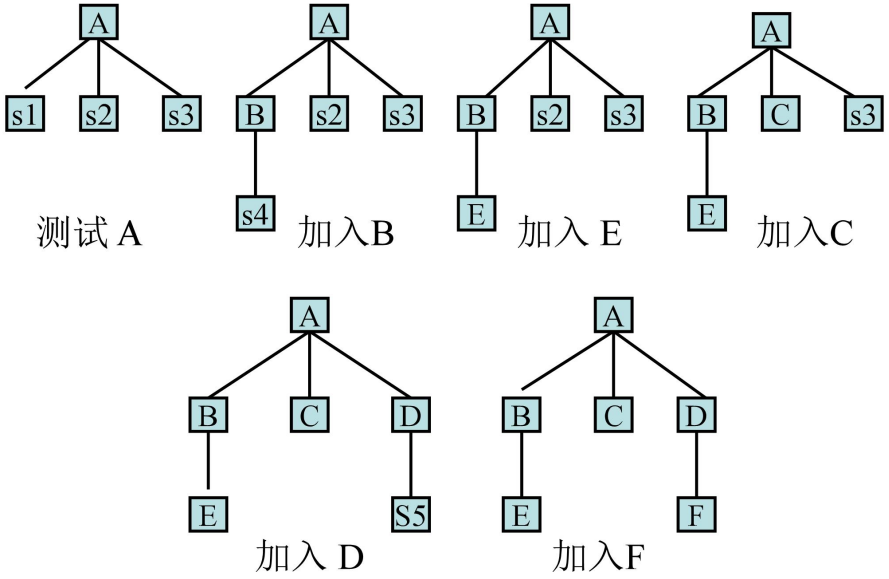
(1) 自顶向下的增量集成



- 自顶向下集成
  - 深度优先：A、B、E、C、D、F
  - 广度优先：A、B、C、D、E、F

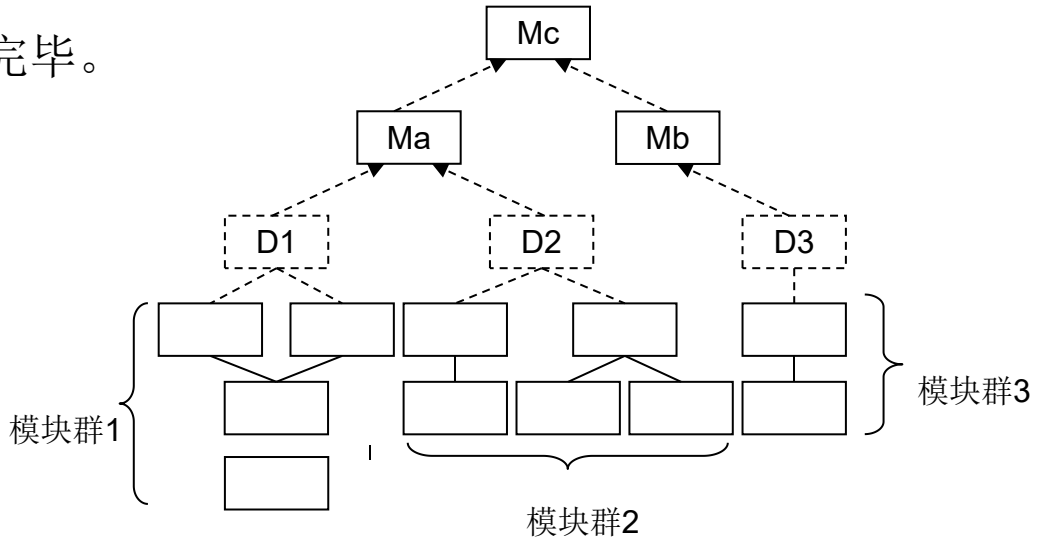
- 广度优先的测试过程：
  - Test A (with stubs for B,C,D)
  - Test A;B (with stubs for E,C,D)
  - Test A;B;C (with stubs for E,D)
  - Test A;B;C;D (with stubs for E,F)
  - Test A;B;C;D;E (with stubs for F)
  - Test A;B;C;D;E;F

深度优先的测试过程



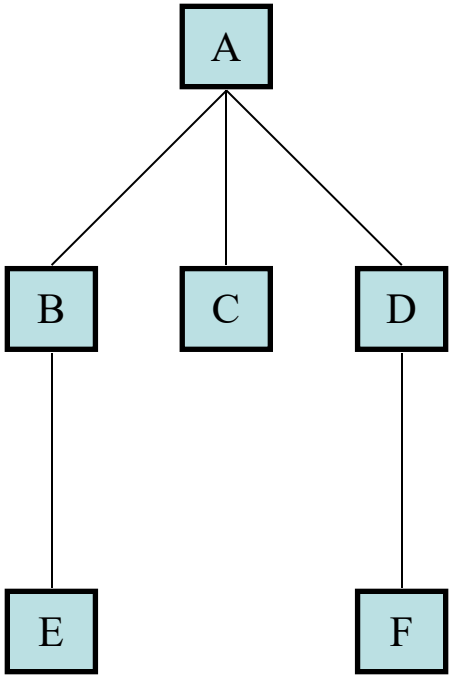
(2) 自底向上的增量集成

- **自底向上的集成测试：从软件结构最底层的模块开始组装测试。**
- **具体步骤：**
  - 1：把底层模块组织成实现某个子功能的模块群(cluster)；
  - 2：开发一个测试驱动模块，控制测试数据的输入和测试结果的输出；
  - 3：对每个模块群进行测试；
  - 4：删除测试使用的驱动模块，用较高层模块把模块去组织成为完成更大功能的新模块；
  - 5：循环，直到整个程序测试完毕。



(2) 自底向上的增量集成

- 过程：
  - Test E (with driver for B)
  - Test C (with driver for A)
  - Test F (with driver for D)
  - Test B;E (with driver for A)
  - Test D;F (with driver for A)
  - Test (A;B;C;D;E;F)



两种集成测试的优缺点

- **自顶向下集成：**
  - 优点：能尽早地对程序的主要控制和决策机制进行检验，因此可较早地发现错误；软件某个完整功能可以被实现和展示（先深集成策略）；较少需要驱动模块；
  - 缺点：所需的桩模块数量巨大；在测试较高层模块时，低层处理采用桩模块替代，不能反映真实情况，重要数据不能及时回送到上层模块，因此测试并不充分；

- **自底向上集成：**
  - 优点：不用桩模块，测试用例的设计亦相对简单；
  - 缺点：程序最后一个模块加入时才具有整体形象，难以尽早建立信心。

(3) 三明治集成

- **三明治集成：一种混合增量式集成策略，综合了自顶向下和自底向上两种方法的优点。**
- **具体步骤：**
  - S1：确定以哪一层为界来决定使用三明治集成策略；
  - S2：对该层次及其下面的所有各层使用自底向上的集成策略；
  - S3：对该层次之上的所有各层使用自顶向下的集成策略；
  - S4：把该层次各模块同相应的下层集成；
  - S5：对系统进行整体测试。

(3) 三明治集成

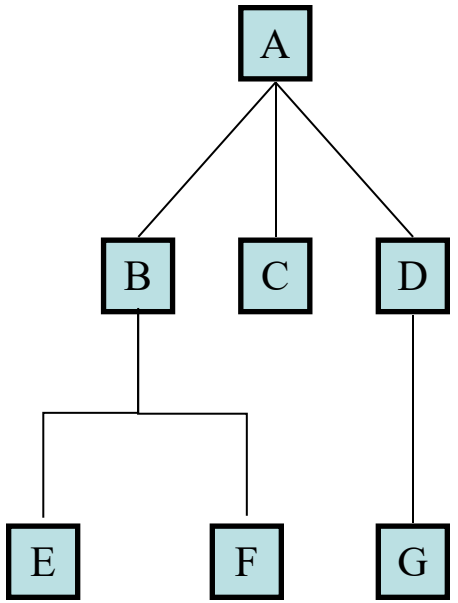
- **选定模块B所在的中间层，过程如下：**
  - Test A (with stubs for B,C,D)
  - Test B (with driver for A and stubs for E,F)
  - Test C (with driver for A)
  - Test D (with driver for A and stub for G)
  - Test A;B (with stubs for E,F,C,D)
  - Test A;B;C (with stubs for E,F,D)
  - Test A;B;C;D (with stubs for E,F,G)

  自顶向下

  - Test E (with driver for B)
  - Test F (with driver for B)
  - Test B;E;F (with driver for A)
  - Test G (with driver for D)
  - Test D;G (with driver for A)

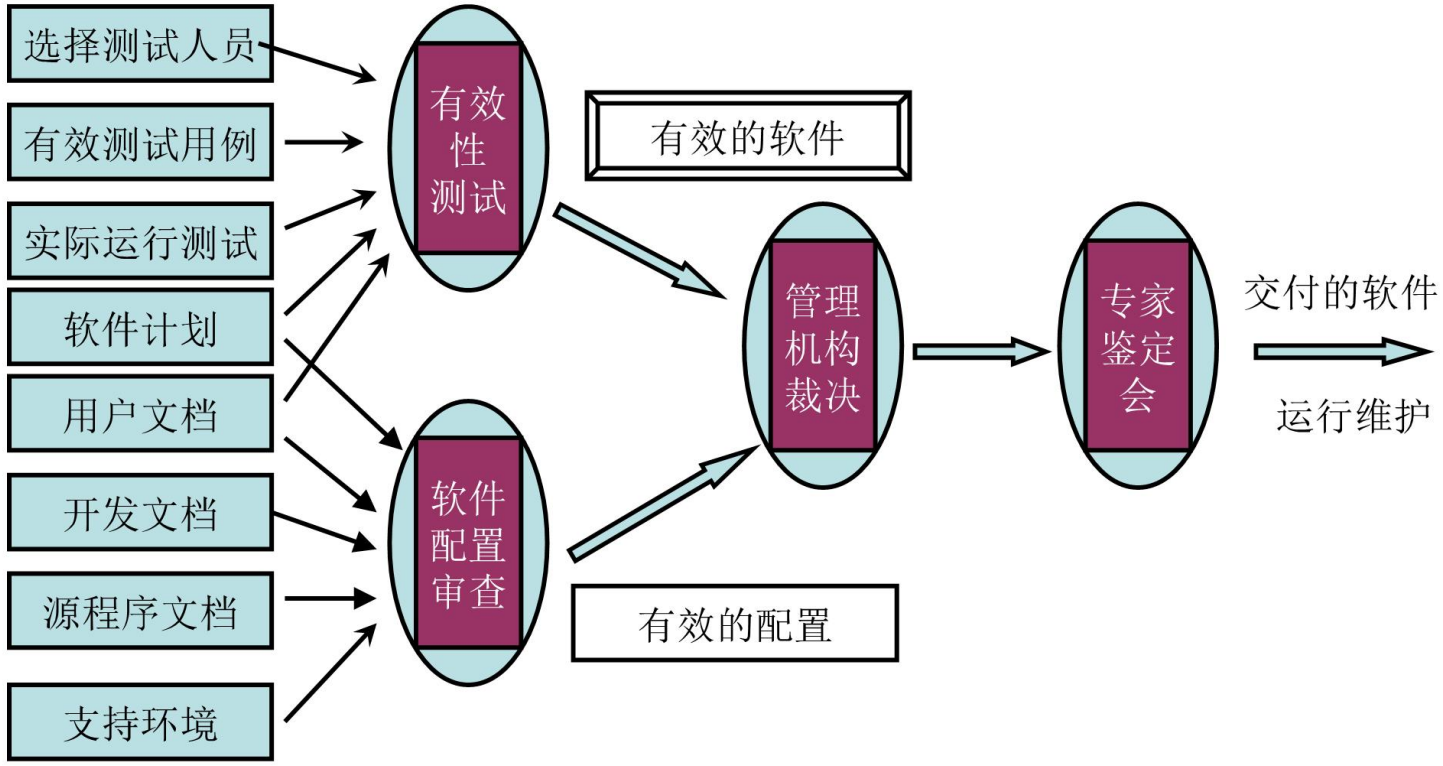
  自底向上

  - Test A;B;C;D;E;F;G



3. 确认测试

- 软件确认是通过一系列表明已符合软件需求的测试而获得的
- 确认测试(Validation Testing)：检查软件能否按合同要求进行工作，即是否满足软件需求说明书中的确认标准。



4. 系统测试

- **系统测试(System Testing)**
  - 系统测试是将已经集成好的软件系统作为一个元素，与计算机硬件、外设、某些支持软件、数据和人员等其他元素结合在一起，在实际运行环境下进行的一系列测试。
- **系统测试方法**
  - 功能测试、协议一致性测试
  - 性能测试、压力测试、容量测试、安全性测试、恢复性测试
  - 备份测试、GUI 测试、健壮性测试、兼容性测试、可用性测试
  - 可安装性测试、文档测试、在线帮助测试、数据转换测试

系统测试：功能测试

- **功能测试(Functional Testing)**
  - 功能测试是系统测试中最基本的测试，它不管软件内部的实现逻辑，主要根据软件需求规格说明和测试需求列表，验证产品的功能实现是否符合需求规格。
  - 功能测试主要发现以下错误：
    - 是否有不正确或遗漏的功能？
    - 功能实现是否满足用户需求和系统设计的隐藏需求？
    - 能否正确地接受输入？能否正确地输出结果？
  - 常用的测试技术
    - 黑盒测试方法：等价类划分、边界值测试

系统测试：压力测试

- **压力测试(Press Testing)**
  - 压力测试是检查系统在资源超负荷情况下的表现，特别是对系统的处理时间有什么影响。
  - 压力测试的例子
    - 对于一个固定输入速率(如每分钟120 个单词)的单词处理响应时间
    - 在一个非常短的时间内引入超负荷的数据容量
    - 成千上万的用户在同一时间从网上登录到系统
    - 引入需要大量内存资源的操作
  - 压力测试采用边界值和错误猜测方法，且需要工具的支持。

系统测试：安全性测试

- **安全性测试(Security Testing)**
  - 安全性测试检查系统对非法侵入的防范能力。
  - 安全性测试期间，测试人员假扮非法入侵者，采用各种办法试图突破防线。
  - 安全性测试的例子
    - 想方设法截取或破译口令
    - 专门定做软件破坏系统的保护机制
    - 故意导致系统失败，企图趁恢复之机非法进入
    - 试图通过浏览非保密数据，推导所需信息

系统测试：恢复测试

- **恢复测试(Recovery Testing)**
  - 恢复测试是检验系统从软件或者硬件失败中恢复的能力，即采用各种人工干预方式使软件出错，而不能正常工作，从而检验系统的恢复能力。
  - 恢复性测试的例子
    - 当供电出现问题时的恢复
    - 恢复程序的执行
    - 对选择的文件和数据进行恢复
    - 恢复处理日志方面的能力
    - 通过切换到一个并行系统来进行恢复

系统测试：GUI测试

- **GUI测试(Graphic User Interface Testing)**
  - GUI测试一是检查用户界面实现与设计的符合情况，二是确认用户界面处理的正确性。
  - GUI测试提倡界面与功能的设计分离，其重点关注在界面层和界面与功能接口层上。
  - GUI自动化测试工具
    - WinRunner，QARun，QARobot，Visual Test
  - 常用的测试技术
    - 等价类划分、边界值分析、基于状态图方法、错误猜测法

系统测试：安装测试

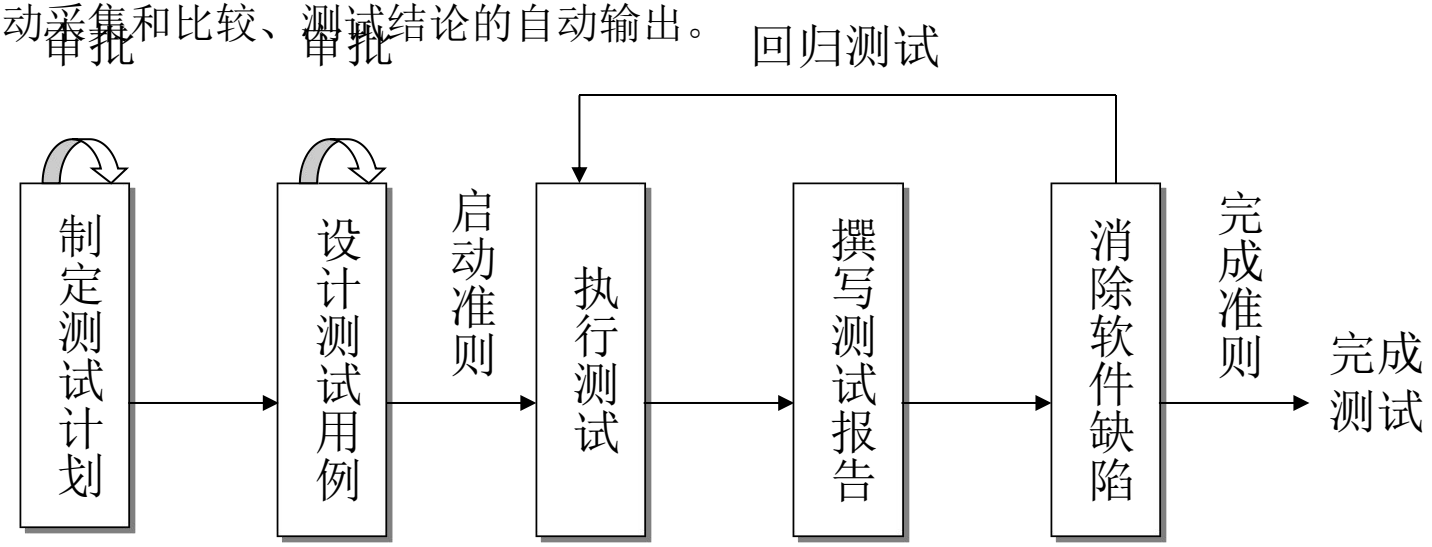
- **安装测试(Installation Testing)**
  - 系统验收之后，需要在目标环境中进行安装，其目的是保证应用程序能够被成功地安装。
  - 安装测试应考虑
    - 应用程序是否可以成功地安装在以前从未安装过的环境中？
    - 应用程序是否可以成功地安装在以前已有的环境中？
    - 配置信息定义正确吗？
    - 考虑到以前的配置信息吗？
    - 在线文档安装正确吗？
    - 安装应用程序是否会影响其他的应用程序吗？
    - 安装程序是否可以检测到资源的情况并做出适当的反应？

5. 验收测试

- **验收测试(Acceptance Testing)**
  - 验收测试是以用户为主的测试，一般使用用户环境中的实际数据进行测试。
  - 在测试过程中，除了考虑软件的功能和性能外，还应对软件的兼容性、可维护性、错误的恢复功能等进行确认。
- **α测试与β测试**
  - α测试与β测试是产品在正式发布前经常进行的两种测试；
    - α测试是由用户在开发环境下进行的测试；
    - β测试是由软件的多个用户在实际使用环境下进行的测试。

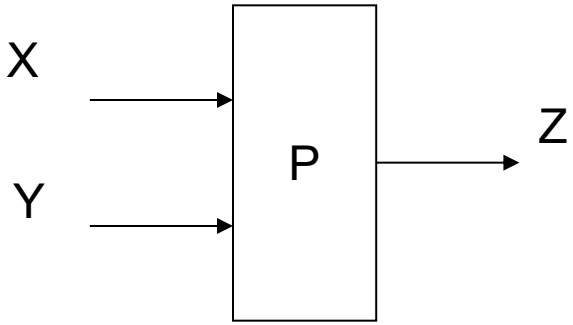
6. 回归测试

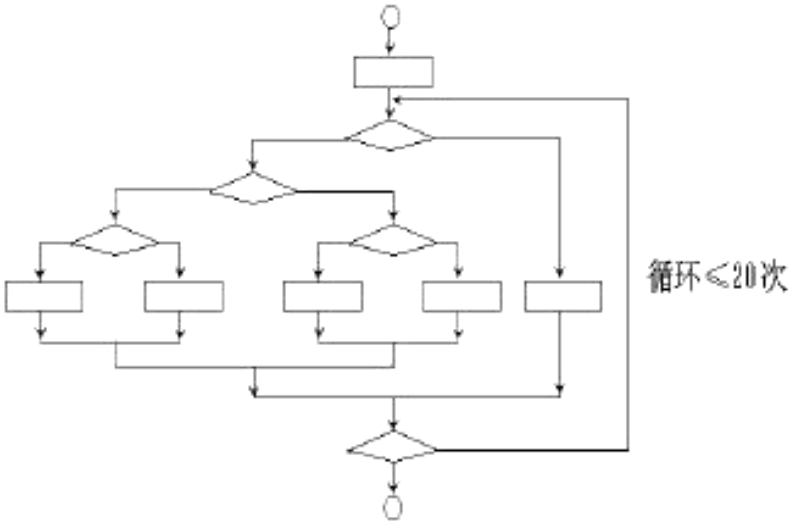
- **回归测试(Regression Testing)**
  - 回归测试是验证对系统的变更没有影响以前的功能，并且保证当前功能的变更是正确的。
  - 回归测试可以发生在软件测试的任何阶段，包括单元测试、集成测试和系统测试，其令人烦恼的原因在于频繁的重复性劳动。
  - 回归测试应考虑的因素
    - 范围：有选择地执行以前的测试用例；
    - 自动化：测试程序的自动执行和自动配置、测试用例的管理和自动输入、测试结果的自动采集和比较、测试结论的自动输出。



典型的软件测试技术

- **黑盒测试：又称“功能测试”或“数据驱动测试”**
  - 它将测试对象看做一个黑盒子，测试人员完全不考虑程序内部的逻辑结构和内部特性，只依据程序的需求规格说明书，检查程序的功能是否符合它的功能说明。**主要用于集成测试、确认测试和系统测试。**
- **白盒测试：又称“结构测试”或“逻辑驱动测试”**
  - 它把测试对象看做一个透明的盒子，它允许测试人员利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序所有逻辑路径进行测试。**主要用于单元测试和集成测试。**





7. 测试用例

测试用例的定义与特征

- **测试用例(testing case):**
  - 测试用例是为特定的目的而设计的一组测试输入、执行条件和预期的结果。
  - 测试用例是执行的最小测试实体。
  - 测试用例就是设计一个场景，使软件程序在这种场景下，必须能够正常运行并且达到程序所设计的执行结果。
- **测试用例的特征:**
  - 最有可能抓住错误的;
  - 不是重复的、多余的;
  - 一组相似测试用例中最有效的;
  - 既不是太简单，也不是太复杂。

测试用例的设计原则

- **测试用例的代表性：**
  - 能够代表并覆盖各种合理的和不合理的、合法的和非法的、边界的和越界的以及极限的输入数据、操作和环境设置等
- **测试结果的可判定性：**
  - 测试执行结果的正确性是可判定的，每一个测试用例都应有相应的期望结果。
- **测试结果的可再现性：**
  - 对同样的测试用例，系统的执行结果应当是相同的。

结束

2024年5月21日

全面质量管理

- **郎中治病的故事**
  - 质量的死对头是缺陷（defect,bug...），缺陷是混在产品中的人们不喜欢、不想要的东西，它对产品没有好处只有坏处。缺陷越多质量越低，缺陷越少质量越高，**提高软件质量的基本手段是消除软件缺陷**。
  - 中国郎中看病的故事

    在中国古代，有一家三兄弟全是郎中。其中老三是名医，人们问他：“你们兄弟三人谁的医术最高？”

    他回答说：“我常用猛药给病危者医治，偶尔有些病危者被我救活，于是我的医术远近闻名并成了名医。我二哥通常在人们刚刚生病的时候马上就治愈他们，临近村庄的人说他是好郎中。我大哥不外出治病，他深知人们生病的原因，所以能够预防家里人生病，他的医术只有我们家里才知道。”
  - 郎中三兄弟是三种治病方式的代言人。

上医治未病之病，中医治将病之病，下医治已病之病

全面质量管理

- **消除软件缺陷的三种方式**
  - 老大治病的方式最高明，如果人们能够预防生病的话，那么没病就用不着看医生了。
    - 提高软件质量最好的办法是：**在开发过程中有效地防止工作成果产生缺陷，将高质量内建于开发过程之中。主要措施是“不断地提高技术水平，不断地提高规范化水平”，其实就是练内功，通称为“软件过程改进”。**
  - 老二治病的方式就是医院的模式，病人越早看病，就越早治好，治病的代价就越低。
    - 同理，在开发软件的时候，即使人们的技术水平很高，并且严格遵守规范，但是人非机器，总是会犯错误的，因此无法完全避免软件中的缺陷。
    - **当工作成果刚刚产生时马上进行质量检查，及时找出并消除工作成果中的缺陷。这种方式效果比较好，人们一般都能学会。最常用的方法是技术评审、软件测试和过程检查，已经被企业广泛采用并取得了成效。**
  - 老三治病的方式代价最高，只能是不得已而为之。
    - 在现实中，大多数软件企业采用老三的方式来对付质量问题。典型现象是：在软件交付之前，没有及时消除缺陷。当软件交付给用户后，用着用着就出错了，赶紧请开发者来补救。可笑的是，当软件系统在用户那里出故障了，那些现场补救成功的人倒成了英雄，好心用户甚至还寄来感谢信。

最佳的软件开发实践（IBM）

- 在商业运作中已经证明这是一种能够解决软件开发过程中跟本问题的方法：
  - 迭代开发
  - 需求的管理
  - 应用基于构件的架构
  - 可视化软件建模
  - 持续质量验证
  - 控制软件变更