第三部分 质量管理

（15-21章）

学习用来管理和控制软件质量的原理、技术和概念。

涉及以下问题：

- 高质量软件的一般特征是什么？
- 什么是软件质量保证？
- 软件测试需要应用什么策略？
- 使用什么方法才能设计出有效的测试用例？
- 有没有保证软件正确性的可行方法？
- 如何管理和控制软件开发过程中经常发生的变更？
- 使用什么标准和尺度评估需求模型、设计模型、源代码以及测试用例的质量？

软件工程基础 —— 第17章 软件测试策略

计算机学院 孟宇龙

目录

关键概念

- α 测试
- β 测试
- 自底向上集成
- 类测试
- 簇测试
- 完成
- 配置审核
- 调试
- 部署测试
- 驱动
- 独立测试组
- 集成测试

什么是软件质量？

- 软件质量的定义不明确，就谈不上软件测试。

- 高质量的软件是一个重要目标，在一般意义上，软件质量可以这样定义：**在一定程度上，应用有效的软件过程，创造有用的产品，为生产者和使用者提供明显的价值。**

软件质量保证

- 延伸前述的质量定义，**软件质量保证就是为了保证软件高质量而必需的“有计划、系统化的行动模式”**。

- 软件质量保证（质量管理）是适用于整个软件过程的一种普适性活动，**而不仅仅是在编码完成之后才开始进行。**

软件质量保证

- 软件质量保证是由两个不同人群相联系的多种任务组成——分别是做技术工作的**软件工程师**和负有质量策划、监督、记录、分析和报告责任的**软件质量保证组**。
- 其中，软件工程师通过采用可靠的技术方法和措施，进行技术评审，并进行周密计划的软件测试来获得质量。
- 可见，**软件测试是软件质量保证活动中的一部分**。

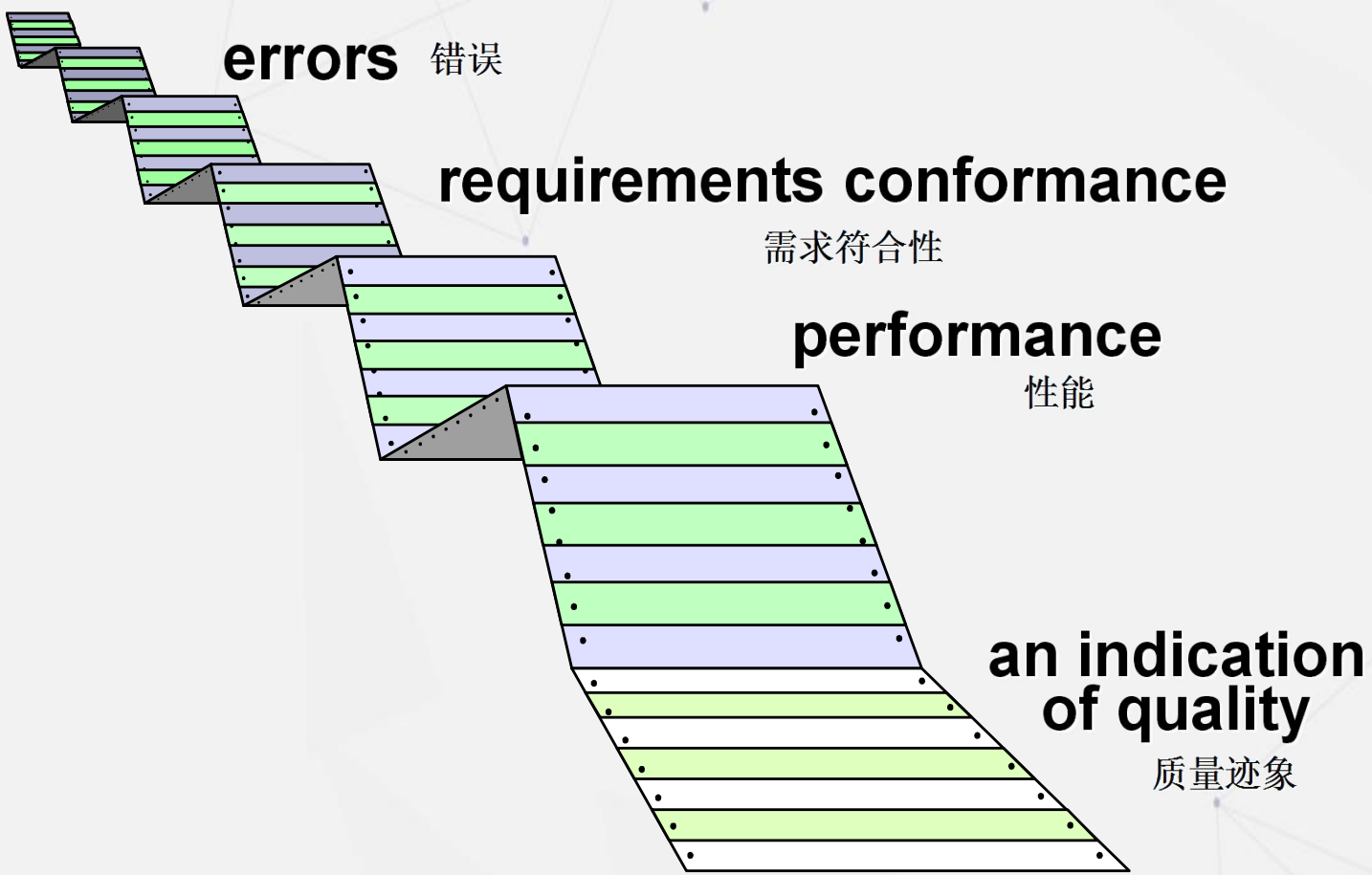
软件测试

- 测试是在交付产品给最终用户之前，带着特定的目的在运行程序的过程中发现错误。
- 测试的目的是为了发现测试对象的问题，而不是证明测试对象没有问题。

测试可以提高软件的质量吗？

- 软件公司一般都有自己的测试中心或测试部门，他们的职责和作用是什么呢？读者可能会不加思索地回答：“测试可以提高软件产品的质量！”
- 我们说：“回答错了”，为什么？因为测试只能发现软件产品的“不符合项”或错误(Bug)，不能改正软件产品的错误，所以不能直接提高软件产品的质量。这个问题就是软件测试的作用。
- 优秀的测试团队可在早期发现错误，使软件维护的费用降到最低点。

测试显示了什么


errors 错误
requirements conformance
需求符合性
performance
性能
an indication
of quality
质量迹象

软件测试策略

软件测试策略提供了一张路线图：

描述将要进行的测试步骤，包括这些步骤的计划和执行时机，以及需要的工作量、时间和资源。

任何测试策略都必须包括：

测试计划、测试用例设计、测试执行以及测试结果数据的收集与评估

17.1 软件测试的策略性方法

测试策略为软件开发人员提供了测试模板，且具备下述一般特征：

- 为完成有效的测试，应该进行有效的技术评审。通过评审，许多错误可以在测试开始之前排除。
- 测试开始于构件层，然后向外“延伸”到整个基于计算机系统的集成。
- 不同的测试技术适用于不同的软件工程的方法和不同的时间点。
- 测试由软件开发人员和（对大型项目而言）独立的测试组执行。
- 测试和调试是不同的活动，但任何测试策略都必须包括调试。

17.1.1 验证与确认

- 软件测试是通常所讲的更为广泛的主题——验证与确认的一部分。
- **验证**是指确保软件正确地实现某一特定功能的一系列活动。
- **确认**指的是确保开发的软件可追溯到客户需求的另外一系列活动。Boehm [Boe81] 用另一种方式说明了这两者的区别：
  - 验证：我们在正确地构造产品吗？
  - 确认：我们在构造正确的产品吗？

17.1.2 谁测试软件？

developer

开发者

理解系统，但是测试很“温和”，由“交付”驱动。

开发者的软件分析与设计连同编码是建设性的任务。其精心地设计和执行测试，试图证明其程序的正确性，而不是发现错误。

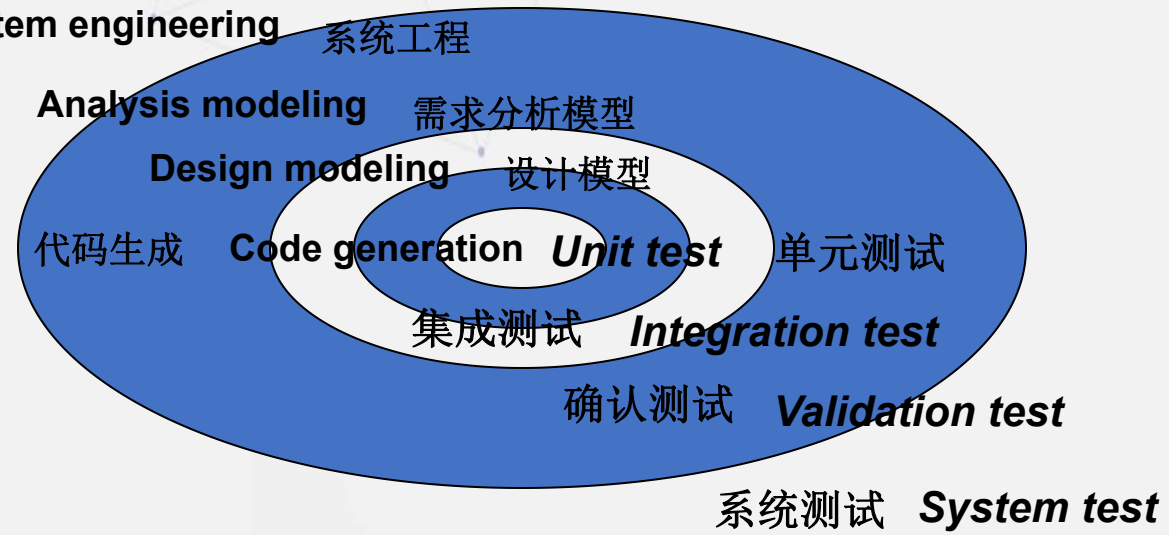
independent tester

独立测试者

必须了解系统，但试图打破它，由质量驱动。

以开发者的观点，可以认为测试是破坏性的。由一种微妙但确实存在的企图，试图摧毁软件工程师所建造的大厦。

17.1.3 测试策略


System engineering
系统工程
Analysis modeling
需求分析模型
Design modeling
设计模型
代码生成
Code generation
Unit test
单元测试
集成测试
Integration test
确认测试
Validation test
系统测试
System test

17.1.3 测试策略

- 我们首先以 ‘小的测试’ 开始，随后转向 ‘大的测试’
- 对于传统的软件
  - 我们最初关注模块（构件）
  - 随后关注集成模块
- 对于面向对象软件
  - 我们的关注点从单独模块“小的测试”（传统的观点）变化到一个包含属性和操作以及暗含沟通和协作的面向对象类。

17.1.4 测试完成的标准

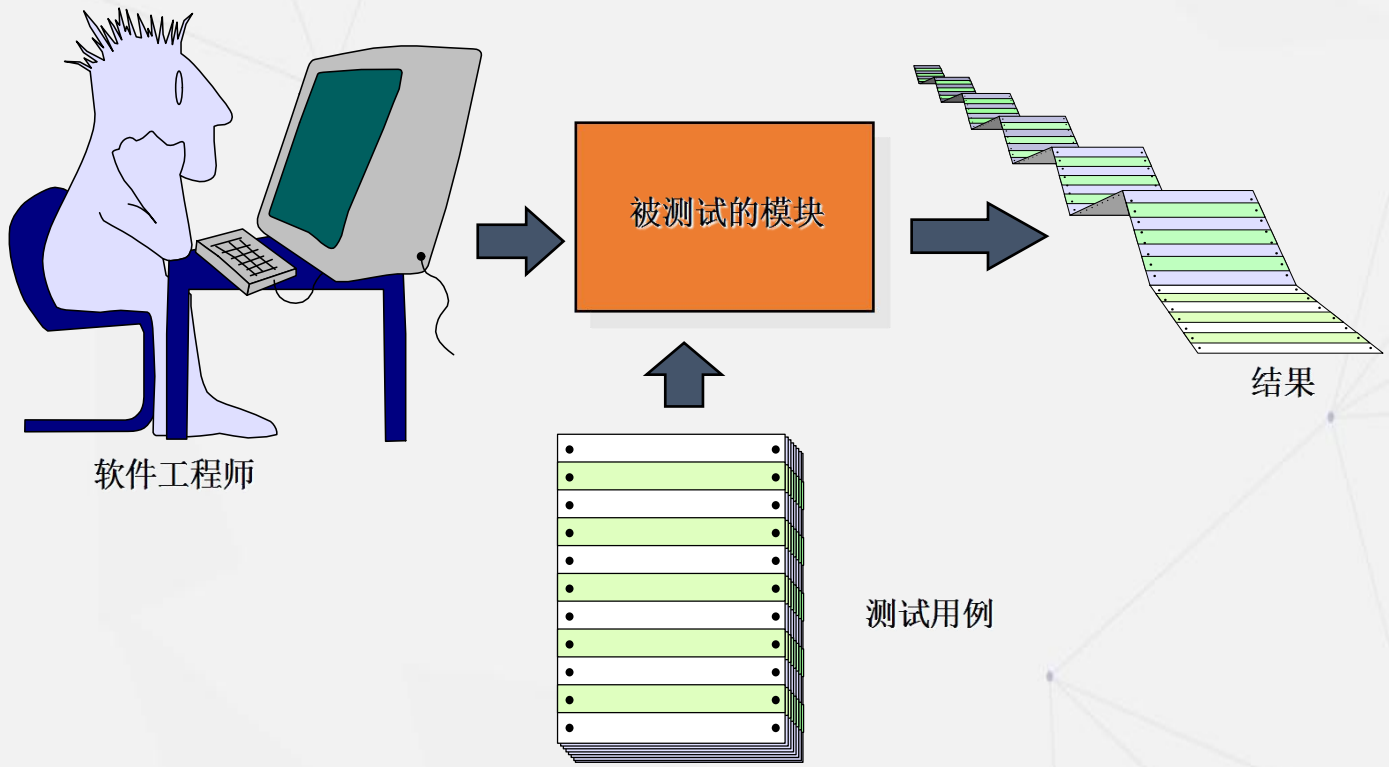
系统通过客户验收，顺利运行后！

17.2 策略问题

好的指导原则

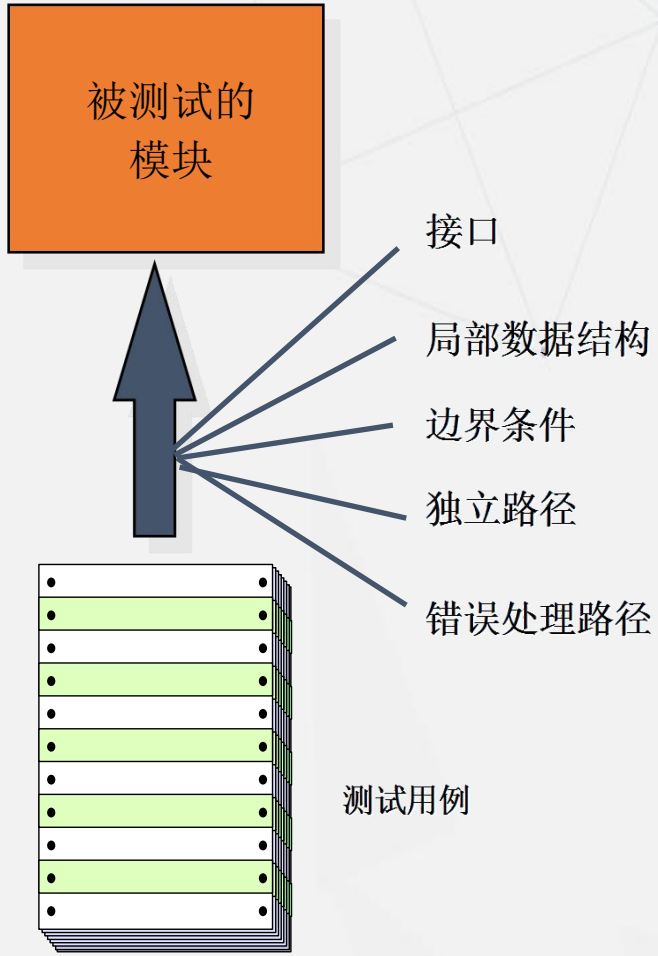
1. 早在测试之前，就要以量化的方式规定产品需求。
2. 明确地陈述测试目标。
3. 了解软件的用户并为每类用户建立用户描述。
4. 指定强调“快速周期测试”的测试计划。
5. 建立能够测试自身的“健壮”软件。
6. 测试之前，利用有效的正式技术评审作为过滤器。
7. 实施正式技术评审以评估测试策略和测试用例本身。
8. 为测试过程建立一种持续的改进方法。

- 单元测试是对软件的基本组成单元进行测试。
- 单元可以是：函数、子过程、类或类的方法、独立的过程和函数、一个菜单或显示界面



17.3.1 单元测试

17.3.1 单元测试



模块接口

- 这是对模块接口进行的测试，检查进出程序单元的数据流是否正确。
- 模块接口测试必须在任何其它测试之前进行。
- 若数据不能正确的输入输出，那么其他测试都是没有意义的。

局部数据结构

- 在模块工作过程中，必须测试模块内部的数据能否保持完整性，包括内部数据的内容、形式及相互关系不发生错误。
- 对于局部数据结构，应该在单元测试中注意发现以下几类错误：

（1）不正确的或不一致的类型说明。

（2）错误的初始化或默认值。

（3）错误的变量名，如拼写错误或书写错误。

（4）下溢、上溢或者地址错误。

独立路径测试

- 在单元测试中，最主要的测试是针对路径的测试。
- 测试用例必须能够发现由于计算错误、不正确的判定或不正常的控制流而产生的错误。
- 执行控制结构中的所有独立路径以确保模块中的所有语句至少能执行一次。

出错处理

- 测试出错处理的重点是模块在工作中发生了错误，其中的出错处理设施是否有效。好的设计要求能够预置出错条件并设置异常处理路径，以便当错误确实出现时重新确定路径或彻底中断处理。
- 检验程序中的出错处理可能面对的情况有：

（1）对运行发生的错误描述难以理解。

（2）所报告的错误与实际遇到的错误不一致。

（3）出错后，在错误处理之前就引起系统的干预。

（4）例外条件的处理不正确。

（5）提供的错误信息不足，以至于无法找到错误的原因。

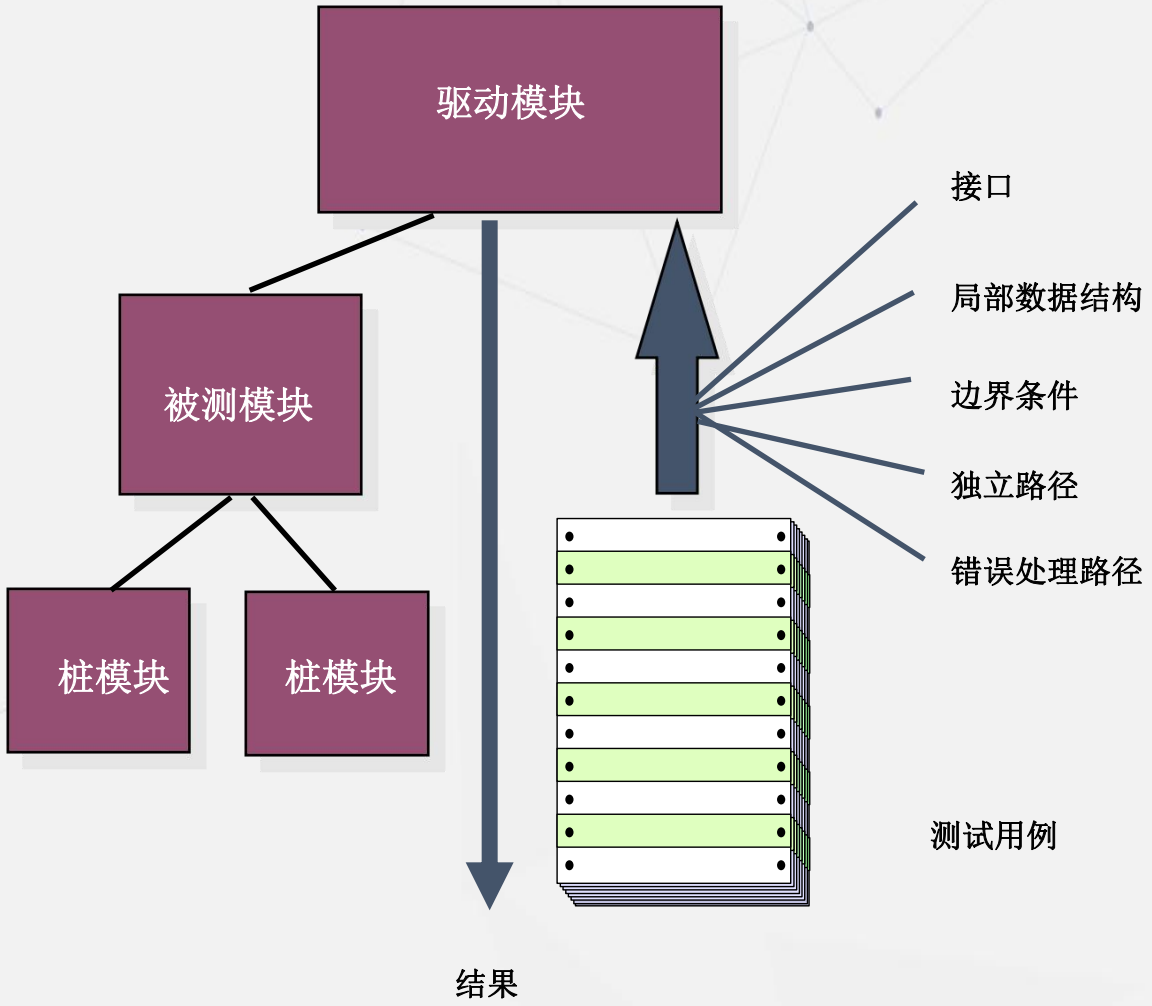
边界条件

- 边界测试是单元测试的最后一步，必须采用边界值分析方法来设计测试用例，测试边界条件，确保在达到边界值的极限或受限处理的情形下，看模块是否能够正常工作。
- 边界测试是最重要的单元测试任务之一。软件通常在边界出错。也就是说，错误行为往往出现在处理n维数组的第n个元素，或者第i次循环的第i次调用，或者遇到允许出现的最大、最小数值时。使用刚好小于、等于或者大于最大值或最小值的数据结构、控制流和数值作为测试用例就很有可能发现错误。

何时进行单元测试

- 单元测试常常是和代码编写工作同时进行的，在**完成了程序编写、复查和语法正确性验证后**，就应进行单元测试用例设计。
- 在单元测试时，如果模块不是独立的程序，需要设置一些**辅助测试模块**。辅助测试模块有两种：
  - （1）**驱动模块**(Drive)　用来模拟被测试模块的上一级模块，相当于被测模块的主程序。它接收数据，将相关数据传送给被测模块，启动被测模块，并打印出相应的结果。
  - （2）**桩模块**(Stub)　用来模拟被测模块工作过程中所调用的模块。它们一般只进行很少的数据处理。
- 驱动模块和桩模块都是**额外的开销**，虽然在单元测试中必须编写，但并不需要作为最终的产品提供给用户。

单元测试环境



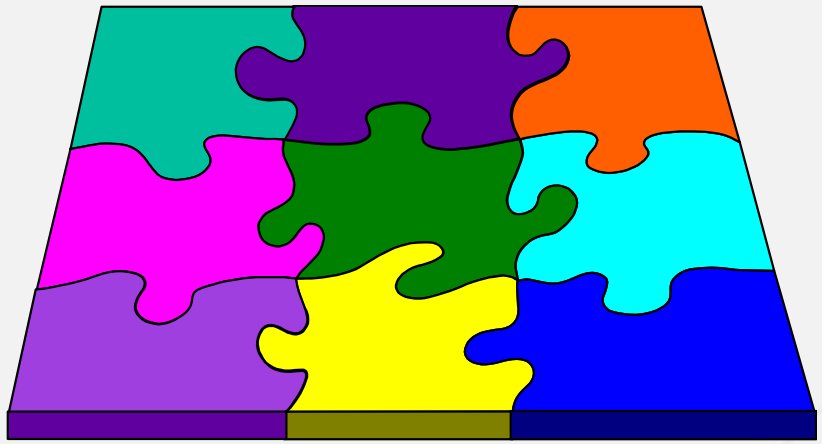
17.3.2 集成测试策略

单元测试之后……

集成测试是构造软件体系结构的系统化技术，同时也是进行一些已在发现与接口相关的错误的测试。

主要有两种集成测试策略：

- “大爆炸”方法，非增量的式的
- 增量构建策略



非增量式测试

- 非增量式测试是采用一步到位的方法来构造测试：

  ——对所有模块进行个别的单元测试后，按照程序结构图将各模块连接起来，把连接后的程序当作一个整体进行测试。

- 非增量式测试的缺点：

  ——当一次集成的模块较多时，非增量式测试容易出现混乱，因为测试时可能发现了许多故障，为每一个故障定位和纠正非常困难，并且在修正一个故障的同时，可能又引入了新的故障，新旧故障混杂，很难判定出错的具体原因和位置。

增量式测试

增量式测试的集成是逐步实现的：

——逐次将未曾集成测试的模块和已经集成测试的模块（或子系统）结合成程序包，再将这些模块集成为较大系统，在集成的过程中边连接边测试，以发现连接过程中产生的问题。

按照不同的实施次序，增量式集成测试又可以分为三种不同的方法：

（1）自顶向下增量式测试

（2）自底向上增量式测试

（3）三明治测试——混合增量测试

自顶向下增量式集成测试

- 自顶向下增量式测试表示逐步集成和逐步测试是按照结构图自上而下进行的，即**模块集成的顺序**是**首先集成主控模块（主程序），然后依照控制层次结构向下进行集成。**从属于主控模块的按深度优先方式（纵向）或者广度优先方式（横向）集成到结构中去。

深度优先方式的集成：

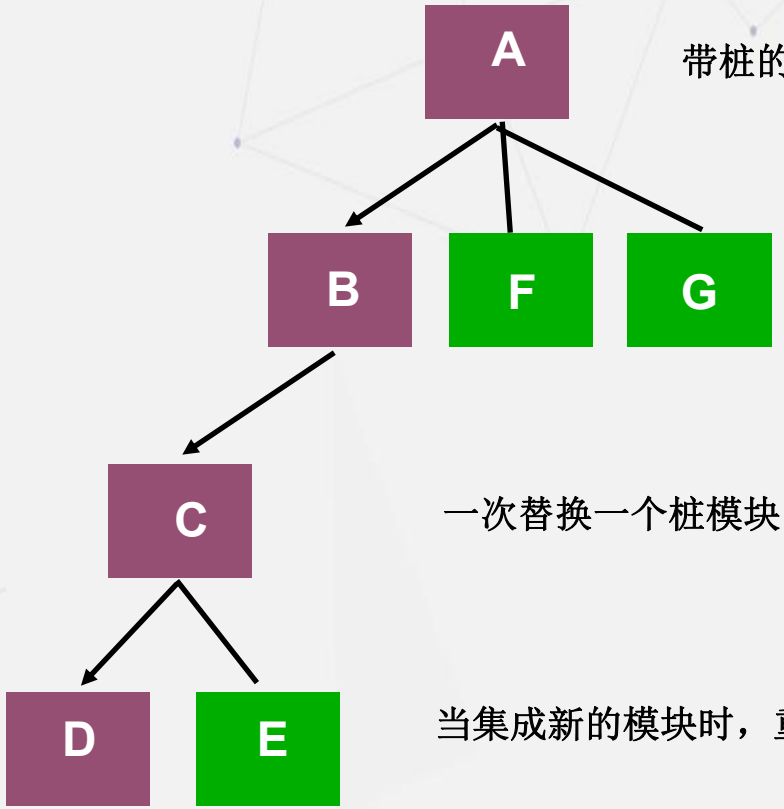
——首先集成在结构中的一个主控路径下的所有模块，主控路径的选择是任意的。

广度优先方式的集成：

——首先沿着水平方向，把每一层中所有直接隶属于上一层的模块集成起来，直到底层。

自顶向下集成测试

深度优先



带桩的顶层模块测试

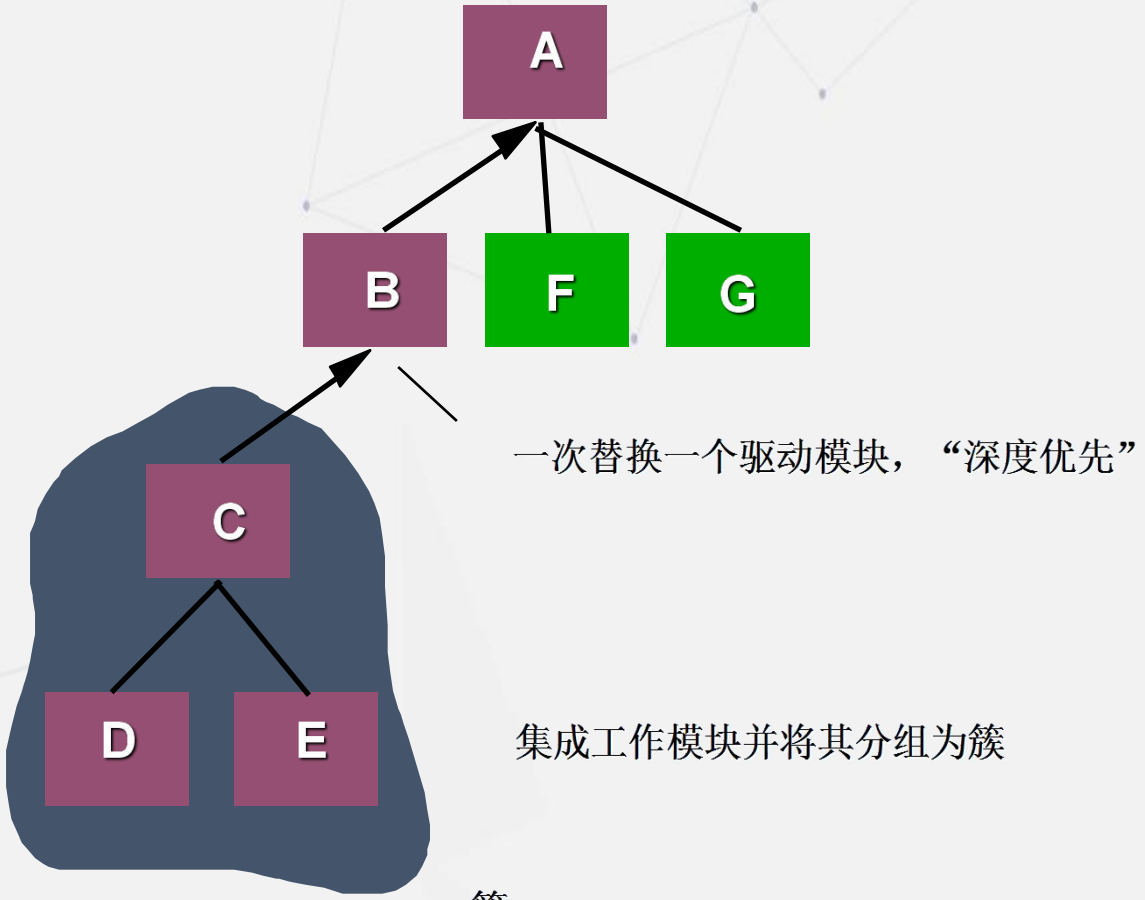
一次替换一个桩模块，“深度优先”

当集成新的模块时，重新运行某些测试子集

自底向上增量式测试

- 自底向上增量式测试表示逐步集成和逐步测试的工作是按结构图自下而上进行的，**即从程序模块结构的最底层模块开始集成和测试。**
- 由于是从最底层开始集成，对于一个给定层次的模块，它的子模块（**包括子模块的所有下属模块**）已经集成并测试完成，所以**不再需要使用桩模块**进行辅助测试。在模块的测试过程中需要从子模块得到的信息可以直接运行子模块得到。

自底向上集成



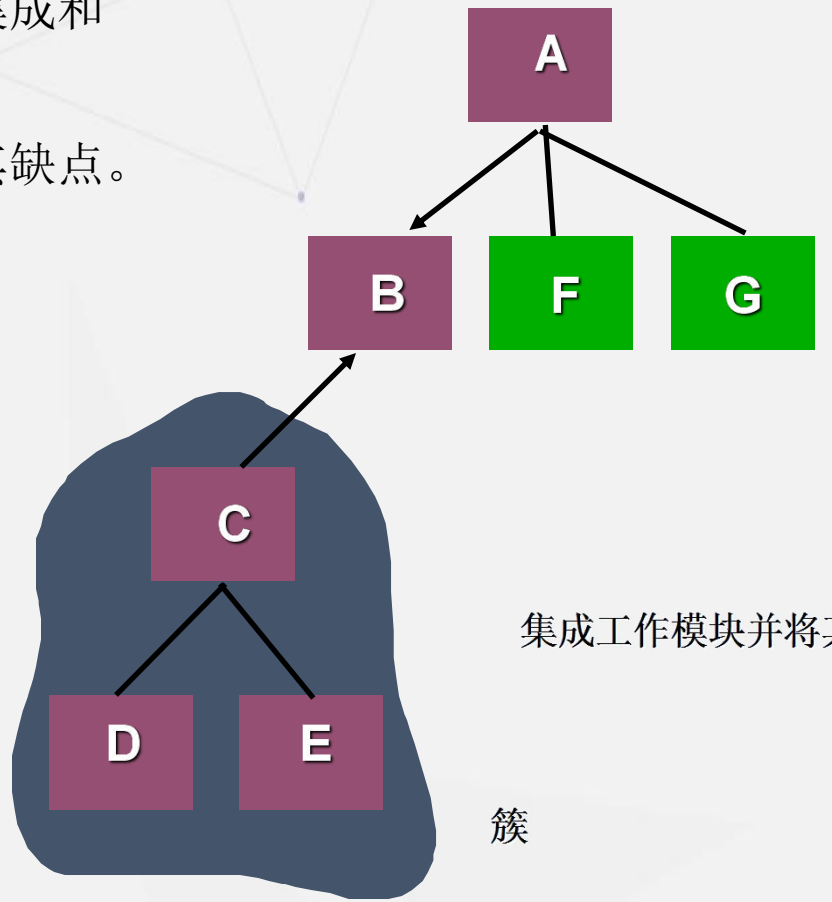
一次替换一个驱动模块，“深度优先”

集成工作模块并将其分组为簇

簇

三明治测试

- **混合增量式测试**是把自顶向下测试和自底向上测试这两种方式结合起来进行集成和测试。
- 这样可以兼具两者的优点，而摒弃其缺点。



带桩测试顶层模块

集成工作模块并将其分组为簇

簇

回归测试

- **回归测试重新执行已测试过的某些子集，以确保变更没有传播不期望的副作用。**
- 无论什么时候修正软件，软件配置的某些方面（程序、文档或支持数据）也发生变更。
- 回归测试有助于保证变更（由于测试或其他原因）不引入无意识行为或额外的错误。
- 回归测试可以手工进行，方法是重新执行所有测试用例的子集，或者利用捕捉/回放工具自动进行。

冒烟测试

- 为生产软件创建“每日构建”的一种常见方法
- 冒烟测试步骤：
  - 将已经转换为代码的软件构件集成到“构造”中去。 一个构造包括所有的数据文件、库、可复用的模块以及实现一个或多个产品功能所需的工程化构件。
  - 设计一系列测试以暴露影响构建正确地完成其功能的错误。
    - 其目的是为了发现极有可能造成项目延迟的“业务阻塞”错误。
  - 每天将该构建与其他构建及整个软件产品（以其当前的形式）集成起来进行冒烟测试。
    - 这种集成方法可以是自顶向下，也可以自底向上。

通用测试标准

- **接口的完整性** –在把每个模块或簇添加到软件中时，测试内部和外部模块的接口。
- **功能的有效性**–通过测试来发现软件中的功能缺陷。
- **信息内容** –通过测试来发现局部的或全局的数据结构中的错误。
- **性能** 验证规定的性能边界是否测试过。

17.4 面向对象测试

- 通过评估分析和设计模型的正确性和一致性开始
- 测试策略改变
  - **由于封装，‘单元’ 的概念扩展。**
    - **封装的类通常是单元测试的重点**，类中包含的操作是最小的可测试单元。面向对象软件的类测试等同于传统软件的单元测试。
  - 集成侧重于类和它们跨线程的执行或使用场景的环境
  - 确认使用传统的黑盒方法
- 测试用例的设计借鉴了传统方法，但是也包含了特殊的特征。

OO测试策略-单元测试和集成测试

- 类测试相当于单元测试
  - 测试类中的操作
  - 检查类中的状态行为
- **集成测试应用三种不同的策略**
  - 基于线程的测试——对响应系统的一个输入或事件所需的一组类进行集成
  - 基于使用的测试——对响应系统的一个使用用例所需的一组类进行集成
  - 簇测试——对演示一个协作所需的一组类进行集成

17.5 确认测试

侧重于需求级的错误

- 集成测试结束后开始确认测试
- 当用户确认后，测试成功

1. 给出测试准则（说明书和一个测试用例文档）
2. 配置评审
3. 完成α测试和β测试，完成验收

17.5.3 α测试和β测试

- α测试
  - 由有代表性的最终用户在开发者的场所进行
- β测试
  - 在一个或多个最终用户场所进行，也称为验收测试

系统测试是软件与其他系统成分（硬件、人、信息）相结合的测试，包括：

恢复测试：通过各种方式强制地使软件发生故障，并验证其能适当恢复。

安全测试：验证建立在系统内的保护机制是否能够实际保护系统不受非法入侵。

压力测试：以非正常的数量、频率或容量的方式执行系统。测试是想要破坏程序。

性能测试：用来测试软件在集成环境中的运行性能，特别是针对实时系统和嵌入式系统，仅提供符合功能需求但不符合性能需求的软件是不能被接受的。

性能测试可以在测试过程的任意阶段进行，但只有当整个系统的所有成分都集成在一起后，才能检查一个系统的真正性能。

性能测试常常和强度（压力）测试结合起来进行，而且常常需要硬件和软件测试设备，这就是说，常常有必要在一种苛刻的环境中衡量资源的使用（比如，处理器周期）。

部署测试：配置测试

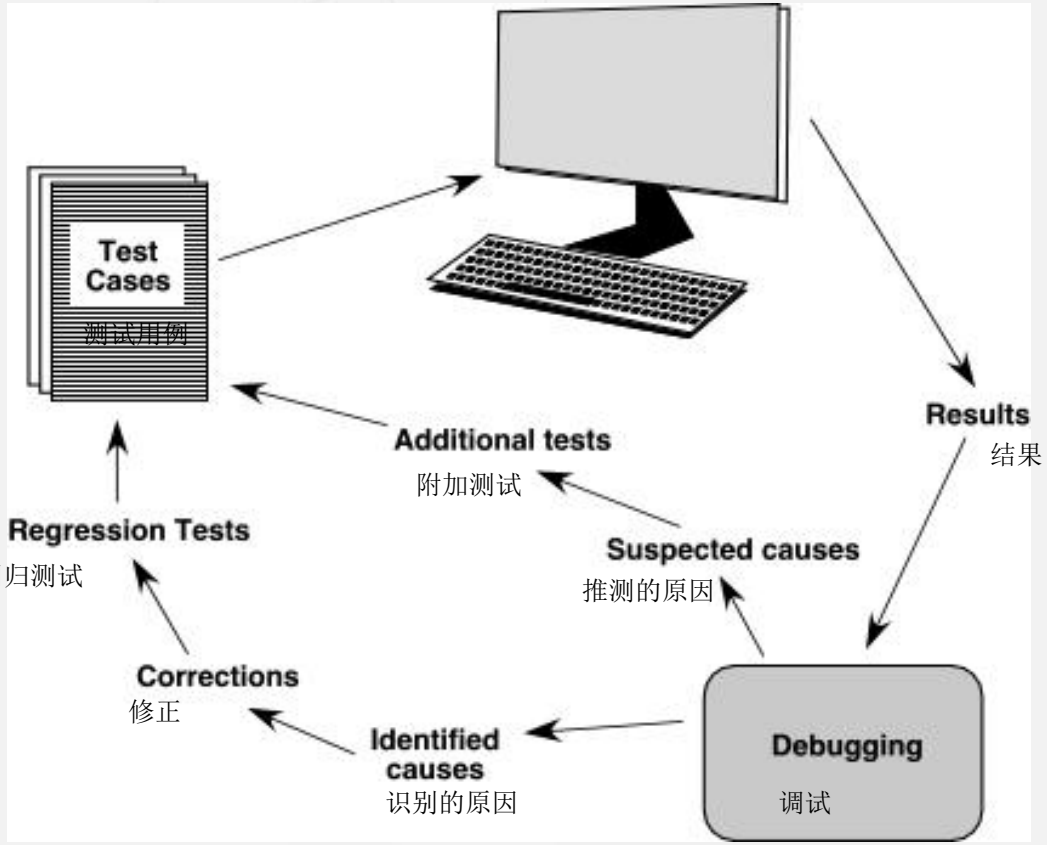
总结-不同测试的分析

- **确认测试**
  - 关注点是软件需求
    - 确认测试也称为合格性测试，是检验所开发的软件是否能按用户提出的要求进行。软件确认要通过一系列证明软件功能和要求一致的黑盒测试来完成，传统软件、面向对象软件及WebApp之间的差别已经消失。
- **系统测试**
  - 关注点是系统集成
    - 由于软件只是计算机系统中的一个组成部分，软件开发完成之后，最终还要和系统中的硬件系统、某些支持软件、数据信息等其他部分配套运行。这些测试已经超出软件过程范围，而且不仅仅由软件工程师执行。
- **α / β 测试**
  - 关注点是客户使用
    - α 测试是由有代表性的最终用户在开发者的现场进行的，开发者在后面观看，并记录错误和使用问题。
    - β 测试是在一个或者多个最终用户场所进行。开发者通常不在场。

17.7 调试：一个诊断过程

- 调试出现在成功的测试之后。也就是说，当测试用例发现错误时，调试是使错误消除的过程。
- 调试就是查找问题症状语气产生原因之间的联系尚未得到很好理解的智力过程

17.7.1 调试过程


Test Cases
测试用例
Results
结果
Additional tests
附加测试
Regression Tests
回归测试
Suspected causes
推测的原因
Corrections
修正
Identified causes
识别的原因
Debugging
调试

调试结果

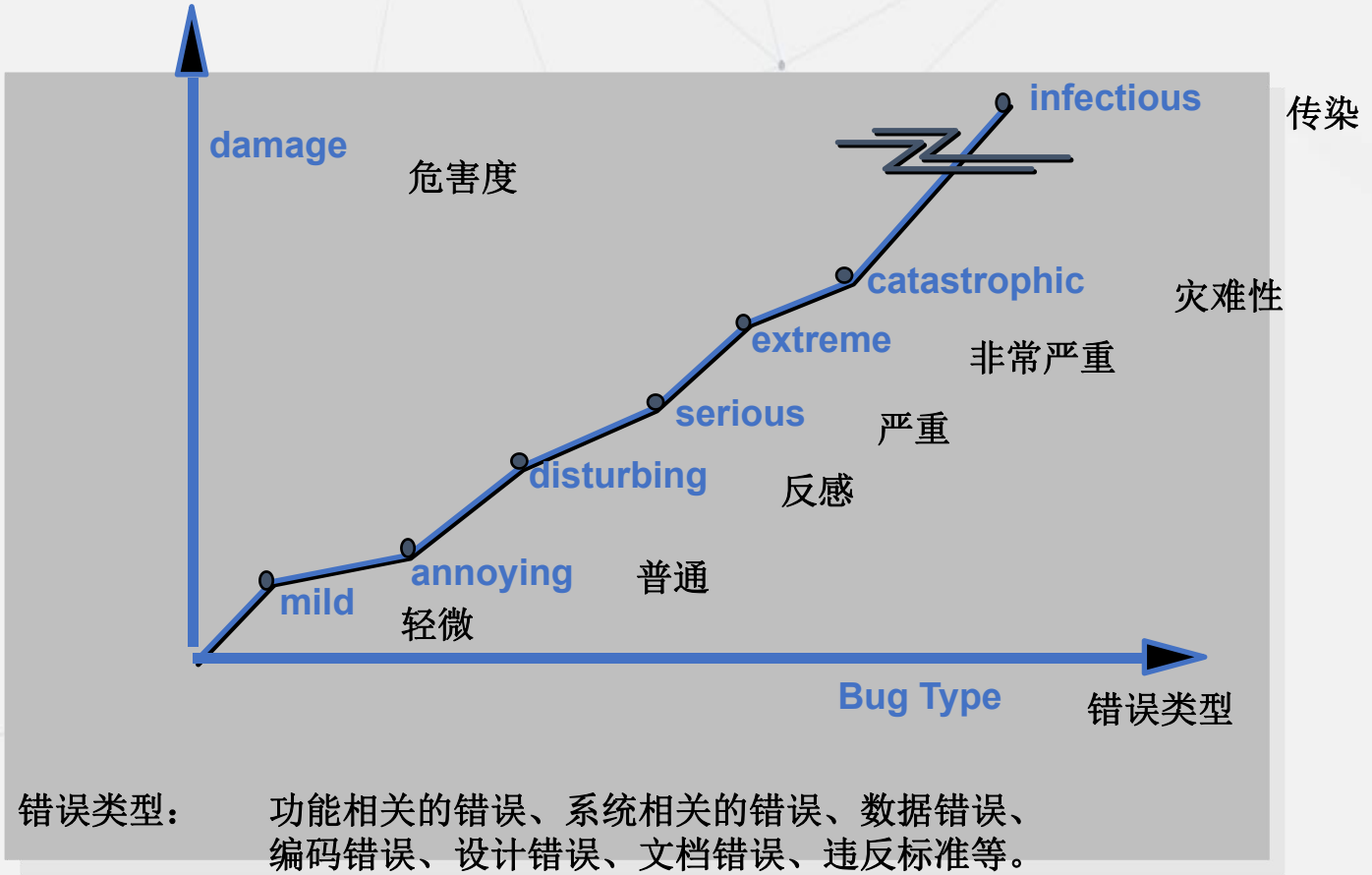
诊断症状并确定原因所需的时间

修正错误并执行回归测试所需的时间

症状和原因

1. 症状与原因出现的地方可能相隔很远
2. 症状可能在另一个错误被改正时消失
3. 实际上可能是由非错误因素引起的
4. 可能是由不易追踪的人为错误引起的
5. 可能是计时问题而不是处理问题引起的
6. 错误的不可复制性
7. 症状可能是间断的
8. 可能是由分布在不同处理器上的不同任务引起的

错误的结果



错误类型：　功能相关的错误、系统相关的错误、数据错误、编码错误、设计错误、文档错误、违反标准等。

17.7.3 调试策略

- 蛮干法/ 测试
- 回溯法
- 归纳法
- 排除法

17.7.4 纠正错误

修改一个错误可能会引入其他错误，不当的修改造成的危害会超过带来的益处，在修改之前，软件工程师应该问以下三个问题：

1. 这个错误的原因在程序的另一部分也产生过吗？在多数情况下，程序的错误是由错误的逻辑模式引起的，这种逻辑模式可能会在别的地方出现。
2. 进行修改可能引发的“下一个错误”是什么？在改正错误之前，应该仔细考虑源代码（最好包括设计）以评估逻辑与数据结构之间的耦合。
3. 为避免这个错误，我们首先应当做什么呢？这个问题是建立统计软件质量保证方法的第一步。若我们不仅修改了过程，还修改了产品，则不仅可以排除现在的程序错误，还可以避免程序今后可能出现的错误。