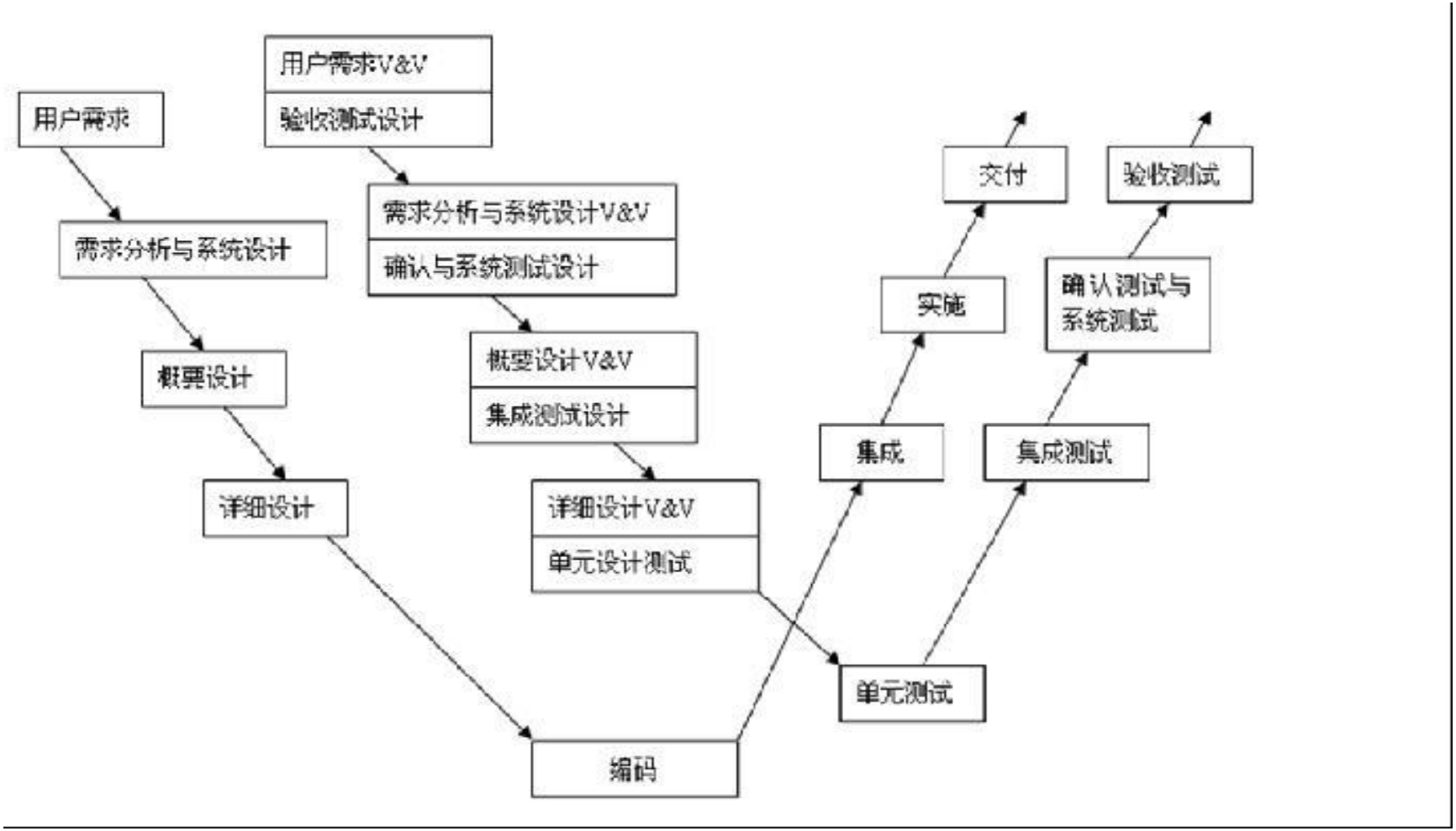
单元测试

及其他阶段性测试

不同阶段的测试

- Unit testing
- Integration testing
- Smoke testing
- System testing
- Regression testing
- Acceptance testing
- Alpha testing
- Beta testing
- Gamma testing

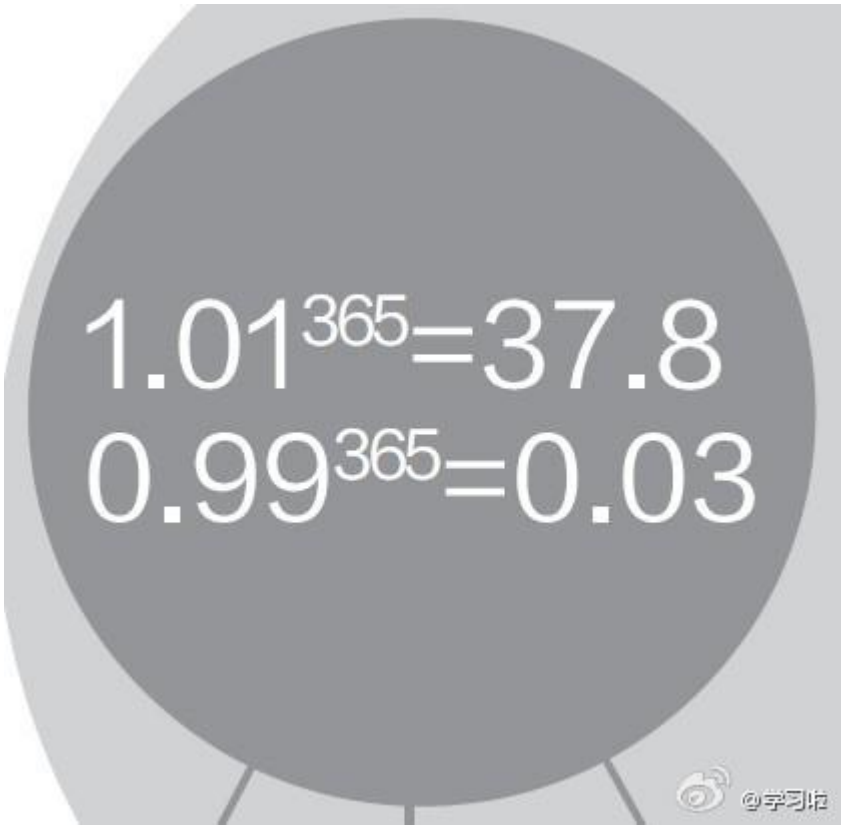
不同阶段的测试


用户需求
用户需求V&V
验收测试设计
需求分析与系统设计
需求分析与系统设计V&V
确认与系统测试设计
概要设计
概要设计V&V
集成测试设计
详细设计
详细设计V&V
单元设计测试
编码
单元测试
集成
集成测试
实施
确认测试与
系统测试
交付
验收测试

1 单元测试理念

- 一辆汽车有1—3万个零件，一架波音747飞机有600万个零件
- 每个零件可靠性90%，总体只有34.87%。只有提高到99.99%，才能达到总体99.9%
- 单元质量决定系统质量
- 只有每行代码、每个变量、每个输入数据、每个函数调用的参数和返回值都被测试过，我们才能对软件质量有足够的信心

励志数学式



- 英语听力累计效应
- 程序故障定位方法的原理

单元测试

单元测试集中检验软件设计的最小单元——模块，可以是函数、过程或类对象等。

正式测试之前必须先通过编译程序检查并且改正所有语法错误，然后用详细设计描述作指南，对重要的执行通路进行测试，以便发现模块内部的错误。

单元测试可以使用白盒测试法，而且对多个模块的测试可以并行地进行。

单元测试

- 单元测试（模块测试）是开发者编写的一小段代码，用于检验被测代码的一个很小的、很明确的功能是否正确。通常而言，一个单元测试是用于判断某个特定条件（或者场景）下某个特定函数的行为。
- 例如，你可能把一个很大的值放入一个有序list 中去，然后确认该值出现在list 的尾部。或者，你可能会从字符串中删除匹配某种模式的字符，然后确认字符串确实不再包含这些字符了

单元测试

- 单元测试是在软件开发过程中要进行的最低级别的测试活动，在单元测试活动中，软件的独立单元将在与程序的其他部分相隔离的情况下进行测试。
- 单元测试不仅仅是作为无错编码一种辅助手段在一次性的开发过程中使用，单元测试必须是可重复的，无论是在软件修改，或是移植到新的运行环境的过程中。因此，所有的测试都必须在整个软件系统的生命周期中进行维护。

单元测试-评价内容

在单元测试期间主要评价模块的下述五个特性：

① 模块接口；

② 局部数据结构；

③ 重要的执行通路；

④ 出错处理通路；

⑤ 影响上述各方面特性的边界条件。

单元测试-接口

在对**接口**进行测试时主要检查下述各点：

(1) 参数数目和由调用模块送来的变元的数目是否相等？
(2) 参数的属性和变元的属性是否匹配？
(3) 参数和变元的单位系统是否匹配？
(4) 传送给被调用模块的变元的数目是否等于那个模块的参数的数目？
(5) 传送给被调用模块的变元属性和参数的属性是否一致？
(6) 传送给被调用模块的变元的单位系统和该模块参数的单位系统是否一致？
(7) 传送给内部函数的变元属性、数目和次序是否正确？
(8) 是否修改了只做输入用的变元？
(9) 全程变量的定义和用法在各个模块中是否一致？

单元测试-接口

如果一个模块完成外部的输入或输出时，还应该再检查下述各点：

(1) 文件属性是否正确？
(2) 打开文件语句是否正确？
(3) 格式说明书与输入／输出语句是否一致？
(4) 缓冲区大小与记录长度是否匹配？
(5) 使用文件之前先打开文件了吗？
(6) 文件结束条件处理了吗？
(7) 输入／输出错误检查并处理了吗？
(8) 输出信息中有文字书写错误吗？

单元测试-局部数据结构

对于一个模块而言，**局部数据结构**是常见的错误来源。应该仔细设计测试方案，以便发现下述类型的错误：

(1) 错误的或不相容的说明；

(2) 使用尚未赋值或尚未初始化的变量；

(3) 错误的初始值或不正确的缺省值；

(4) 错误的变量名字（拼写错或截短了）；

(5) 数据类型不相容；

(6) 上溢、下溢或地址异常。

单元测试-独立路径

- 在控制结构中的所有路径覆盖法或基本路径覆盖法都是需要测试的，以保证在一个模块中的所有语句或分支元素等都能被执行一次。
- 测试用例应当能够发现由于错误计算、不正确的比较或者不正常的控制流而产生的错误如：误解的或不正确的算术优先级；混合模式的操作；不正确的初始化；精度不够准确；表达式的不正确符号表示。
- 当比较和控制流紧密地耦合在一起，即控制流的转移是在比较之后发生的，此时测试用例应当能够发现的错误如：不同数据类型的比较；不正确的逻辑操作或优先级；应该相等的地方由于精度的错误而不能相等；不正确的比较或者变量；不正常的或者不存在的循环终止；当遇到分支循环的时候不能退出；不适当地修改循环变量等。

单元测试-错误处理路径

- 要对所有处理错误的路径进行测试，好的设计要求错误条件是可以预料的，而当错误真的发生的时候，错误处理路径被建立，以重定向或者终止处理。
- 在错误处理部分应当考虑潜在的错误包括：对错误描述不够准确；所报的错误与真正遇到的错误不一致；错误条件在错误处理之前就引起了系统异常；异常条件处理不正确；错误描述没有提供足够的信息来帮助确定错误发生的位置。

单元测试-边界条件

- 保证模块单元在极限或某些严格条件下仍然正确执行，例如错误往往出现在一个n元数组的第n个元素被处理的时候，或者一个i次循环的第i次执行，或允许的最大值或最小值出现的时候。用边际值方法可以有效地发现这类错误。

单元测试过程

1.代码审查及走查

人工测试源程序可以由编写者本人非正式地进行，也可以由审查小组正式进行。后者称为代码审查，它是一种非常有效的程序验证技术，对于典型的程序来说，可以查出30％一70％的逻辑设计错误和编码错误。审查小组最好由下述四人组成：

⑴ 组长，他应该是一个很有能力的程序员，而且没有直接参与这项工程；

⑵ 程序的设计者；

⑶ 程序的编写者；

⑷ 程序的测试者。

如果一个人既是程序的设计者又是编写者，或既是编写者又是测试者，则审查小组中应该再增加一个程序员。

单元测试过程

- 审查之前，小组成员应该先研究设计说明书，力求理解这个设计。为了帮助理解，可以先由设计者扼要地介绍他的设计。在审查会上由程序的编写者解释他是怎样用程序代码实现这个设计的，通常是逐个语句地讲述程序的逻辑，小组其他成员仔细倾听他的讲解，并力图发现其中的错误。
- 审查会上进行的另外一项工作，是对照类似于上一小节中介绍的程序设计常见错误清单，分析审查这个程序。当发现错误时由组长记录下来，审查会继续进行（审查小组的任务是发现错误而不是改正错误）。

单元测试过程

- 审查会还有另外一种常见的进行方法（称为预排）：由一个人扮演“测试者”，其他人扮演“计算机”。会前测试者准备好测试方案，会上由扮演计算机的成员模拟计算机执行被测试的程序
- 当然，由于人执行程序速度极慢，因此测试数据必须简单，测试方案的数目也不能过多
- 但是，测试方案本身并不十分关键，它只起一种促进思考引起讨论的作用
- 在大多数情况下，通过向程序员提出关于他的程序的逻辑和他编写程序时所做的假设的疑问，可以发现的错误比由测试方案直接发现的错误还多

单元测试过程

- 代码审查比计算机测试优越的是：一次审查会上可以发现许多错误；用计算机测试的方法发现错误之后，通常需要先改正这个错误才能继续测试，因此错误是一个一个地发现并改正的。也就是说，采用代码审查的方法可以减少系统验证的总工作量。
- 实践表明，对于查找某些类型的错误来说，人工测试比计算机测试更有效；对于其他类型的错误来说则刚好相反。因此，人工测试和计算机测试是互相补充，相辅相成的，缺少其中任何一种方法都会使查找错误的效率降低。

单元测试过程

2. 测试软件

模块并不是一个独立的程序，因此必须为每个单元测试开发驱动模块和(或)桩(存根)模块。

通常驱动模块也就是一个“主程序”，它接收测试数据，把这些数据传送给被测试的模块，并且印出有关的结果。

桩(存根)模块代替被测试的模块所调用的模块。因此桩(存根)模块也可以称为“虚拟子程序”。它使用被它代替的模块的接口，可能做最少量的数据操作，印出对入口的检验或操作结果，并且把控制归还给调用它的模块。

单元测试工具

- CppUnit
- C++ Unit
- Vtest
- Gtest

Example 1

- void CMyClass::Grow(int years)
  {
  mAge += years;
  - if(mAge < 10)
    mPhase = "儿童";
    else if(mAge <20)
    mPhase = "少年";
    else if(mAge <45)
    mPhase = "青年";
    else if(mAge <60)
    mPhase = "中年";
    else
    mPhase = "老年";
  }
- 测试函数中的一个测试用例：
  CaseBegin();{
  int years = 1;
  pObj->mAge = 8;
  pObj->Grow(years);
  ASSERT(pObj->mAge == 9);
  ASSERT(pObj->mPhase == "儿童");
  }CaseEnd();

单元测试的优点 1

- 它是一种验证行为。
- 程序中的每一项功能都是测试来验证它的正确性。它为以后的开发提供支缓。就算是开发后期，我们也可以轻松的增加功能或更改程序结构，而不用担心这个过程中会破坏重要的东西。而且它为代码的重构提供了保障。这样，我们就可以更自由的对程序进行改进。

单元测试的优点 2

- 它是一种设计行为。
- 编写单元测试将使我们从调用者观察、思考。特别是先写测试（test-first），迫使我们把程序设计成易于调用和可测试的，即迫使我们解除软件中的耦合。

单元测试的优点 3

- 它是一种编写文档的行为。
- 单元测试是一种无价的文档，它是展示函数或类如何使用的最佳文档。这份文档是可编译、可运行的，并且它保持最新，永远与代码同步。

单元测试的优点 4

- 它具有回归性。
- 自动化的单元测试避免了代码出现回归，编写完成之后，可以随时随地的快速运行测试。

单元测试的一些参考准则

- 单元测试通常由最熟悉代码的人来编写和执行 (程序的作者)
- 单元测试的执行速度要快
- 单元测试应产生可重复、一致的结果
- 单元测试过后，机器状态保持不变
- 单元测试应具有一定的独立性，其执行和结果不依赖于别的测试
- 单元测试需要保证一定的代码覆盖率
- 单元测试需要和产品代码一起维护

```
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 2) {
        dayOfMonth += 31;
    } else if (month == 3) {
        dayOfMonth += 59;
    } else if (month == 4) {
        dayOfMonth += 90;
    } else if (month == 5) {
        dayOfMonth += 31 + 28 + 31 + 30;
    } else if (month == 6) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31;
    } else if (month == 7) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
    } else if (month == 8) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;
    } else if (month == 9) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
    } else if (month == 10) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
    } else if (month == 11) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
    } else if (month == 12) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
    }
    return dayOfMonth;
}
```

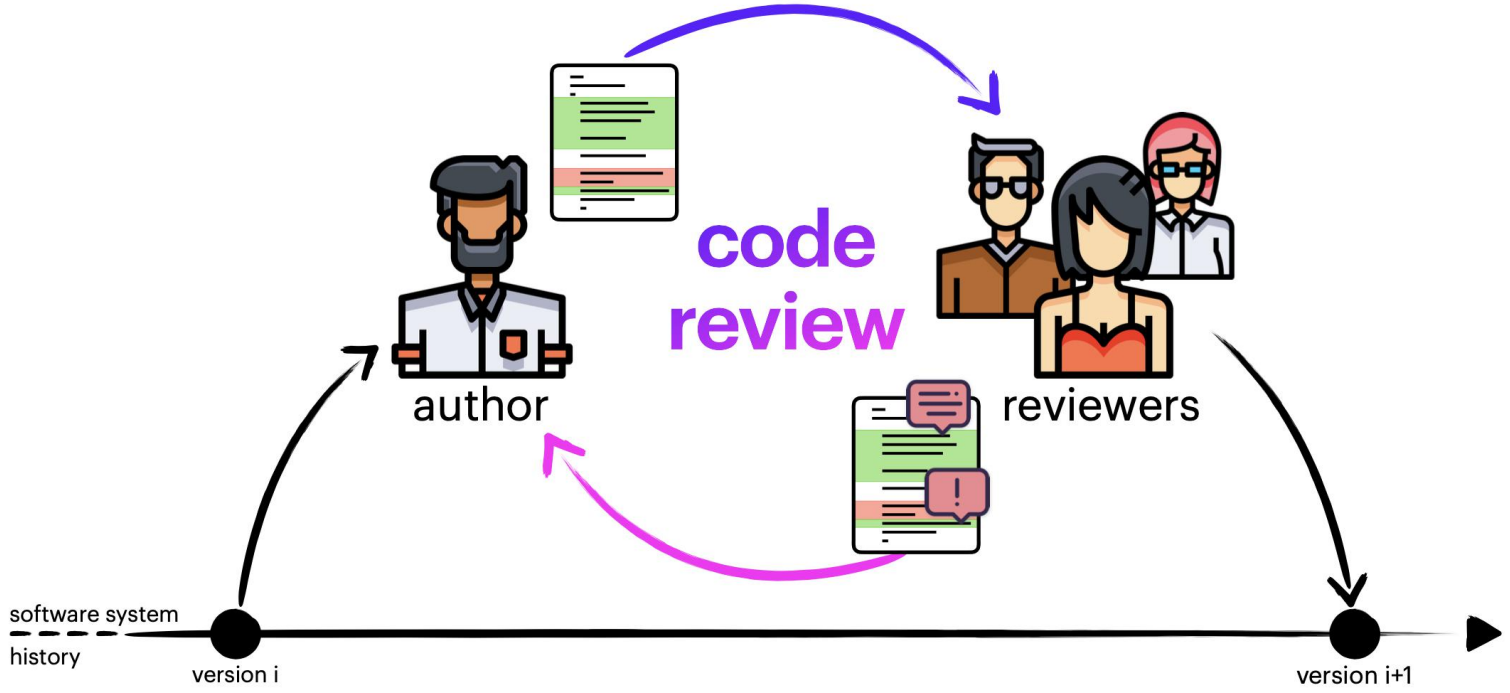
What would you comment on?

代码审查

Code Review

通过同行评审 (*peer-review*) 的方式对代码进行系统性的检查，以尽可能提高代码库 (*code repository*) 的质量

- 一种“静态“的单元测试



代码审查

Code Review

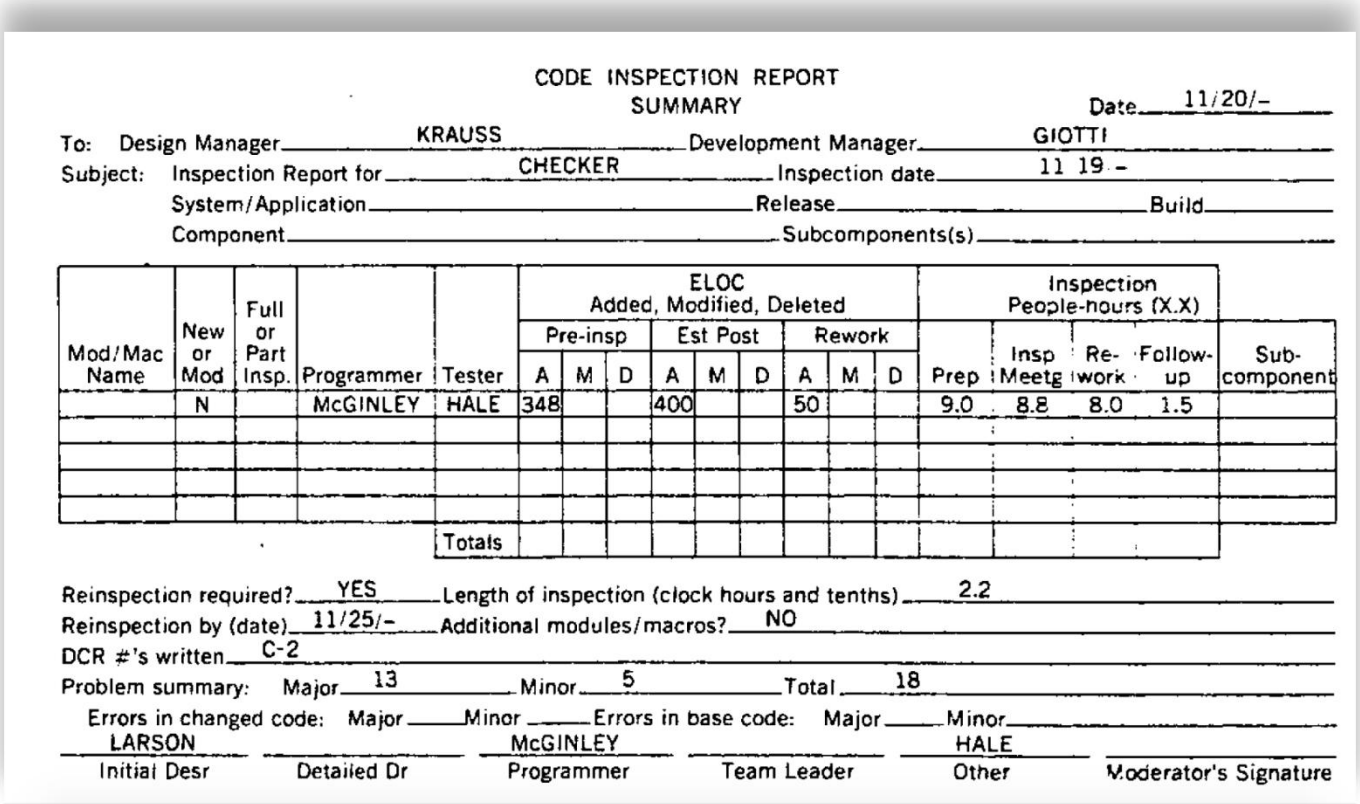
通过同行评审 (*peer-review*) 的方式对代码进行系统性的检查，以尽可能提高代码库 (*code repository*) 的质量

- 找出并及时修正在软件开发初期产生的错误
- 确保开发人员写出的代码是其它人员容易理解的 (代码是否符合特定的编码风格、是否有充分的测试用例 ...)
- 代码作者和评审员之间的知识共享、协作式的问题求解
- 提高安全性、防止意外 (*gatekeeping*)
  - 没有人能在缺乏监管的情况下提交任意代码

代码审查

Code Review

- 在软件开发的不同重要阶段间对设计和代码进行正式的检查 (*Code Inspection, Michael Fagan, 1976*)



CODE INSPECTION REPORT
SUMMARY

Date 11/20/-

To: Design Manager KRAUSS Development Manager GIOTTI

Subject: Inspection Report for CHECKER Inspection date 11 19 -

System/Application Release Build

Component Subcomponents(s)

Mod/Mac Name	New or Mod	Full or Part Insp.	Programmer	Tester	ELOC Added, Modified, Deleted									Inspection People-hours (X.X)				Sub-component
					Pre-insp			Est Post			Rework			Prep	Insp Meetg	Re-work	Follow-up	
					A	M	D	A	M	D	A	M	D					
	N		McGINLEY	HALE	348			400			50			9.0	8.8	8.0	1.5	
				Totals														

Reinspection required? YES Length of inspection (clock hours and tenths) 2.2

Reinspection by (date) 11/25/- Additional modules/macros? NO

DCR #'s written C-2

Problem summary: Major 13 Minor 5 Total 18

Errors in changed code: Major Minor Errors in base code: Major Minor

LARSON (Initial Desr) | (Detailed Dr) | McGINLEY (Programmer) | (Team Leader) | HALE (Other) | (Moderator's Signature)

代码审查

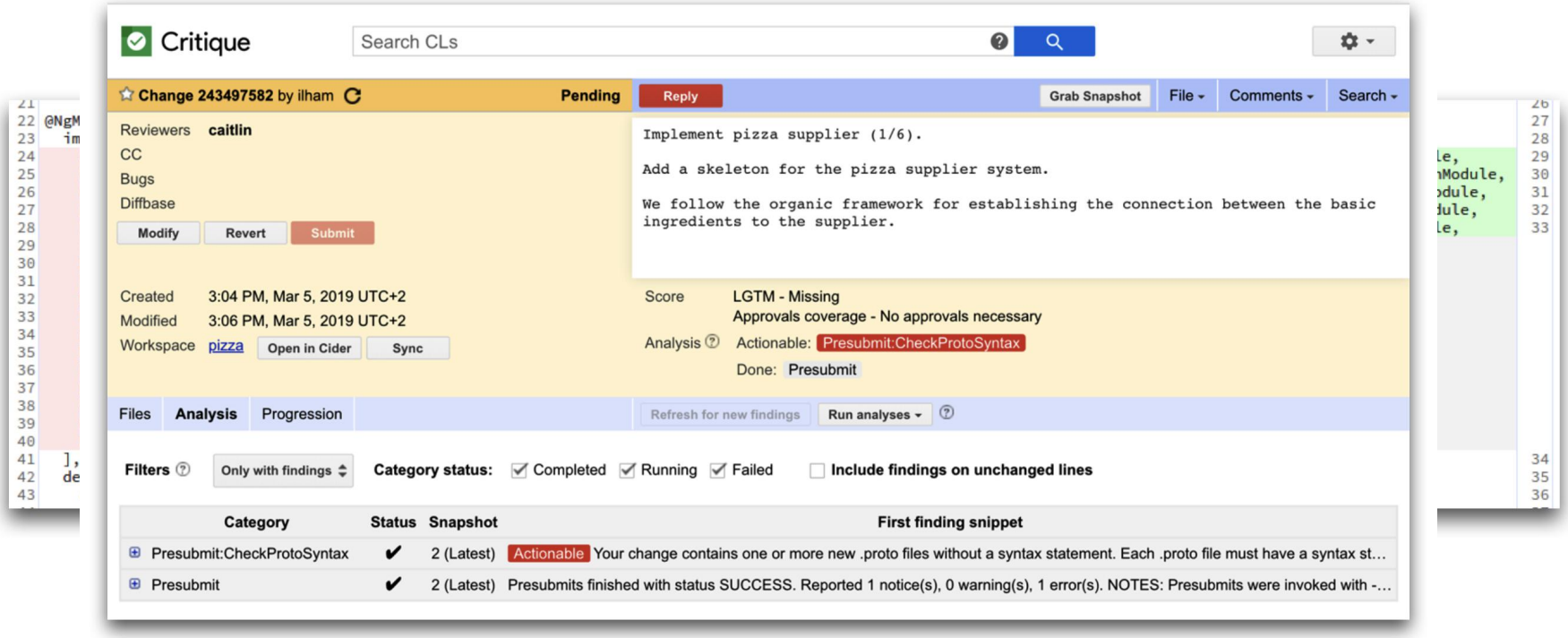
Code Review

- 在软件开发的不同重要阶段间对设计和代码进行正式的检查 (Code Inspection, Michael Fagan, 1976)
- 与早期的方式相比，现代的代码审查实践更强调
  - 更少的角色和步骤要求 (informal)
  - 使用自动化工具支撑整个流程 (tool based)
  - 通常以在线的方式进行 (asynchronous)
  - 更关注当前变化的代码 (code changes)

代码审查

Practice at Google

- *Creating*: 代码作者对代码进行修改
- *Previewing*: 代码作者使用工具 (*Critique*) 来审查代码变化 (包括自动化代码分析工具的结果)，并发送邮件通知评审者

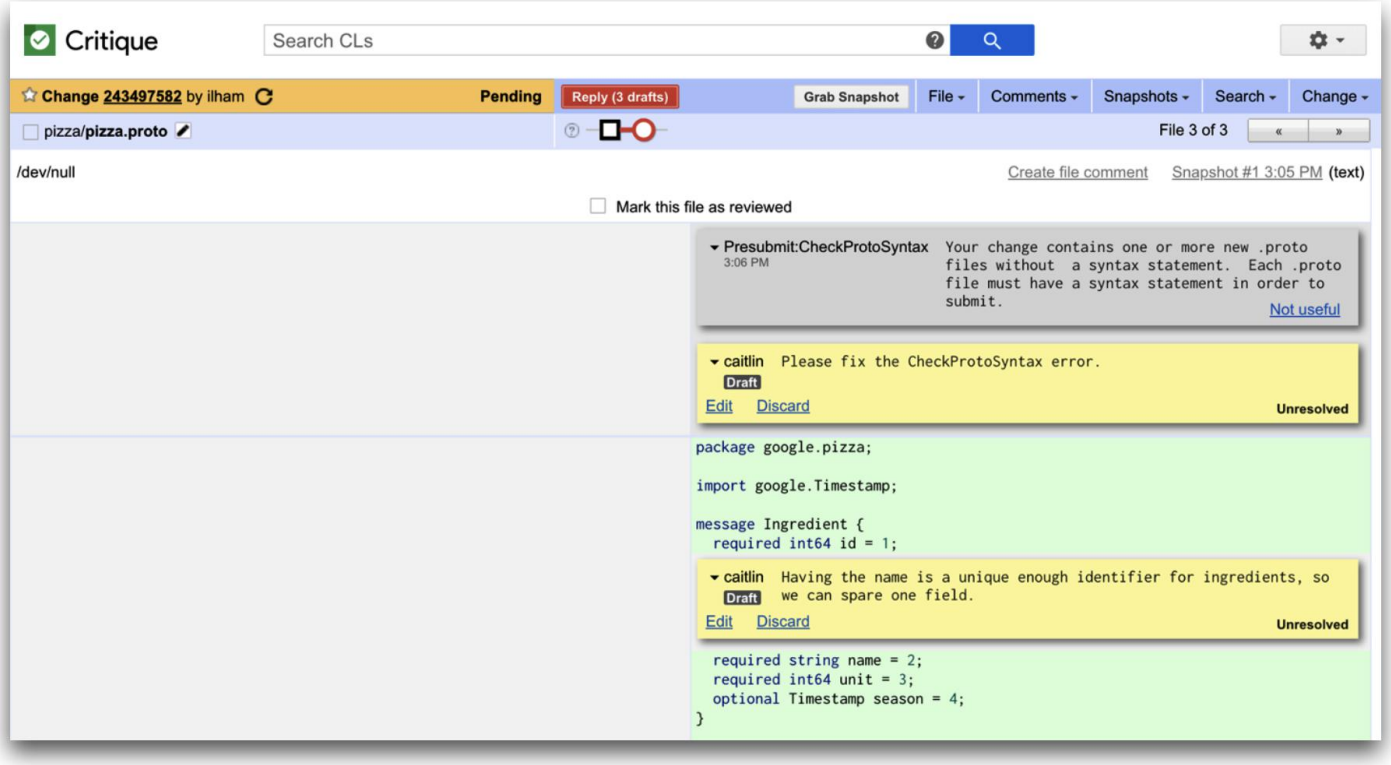


Modern Code Review: A Practice at Google, ICSE SEIP 2018

代码审查

Practice at Google

- Commenting: 评审者在基于 Web 的 GUI 界面上给出评审意见 (unresolved comments 作为代码作者的行动项)

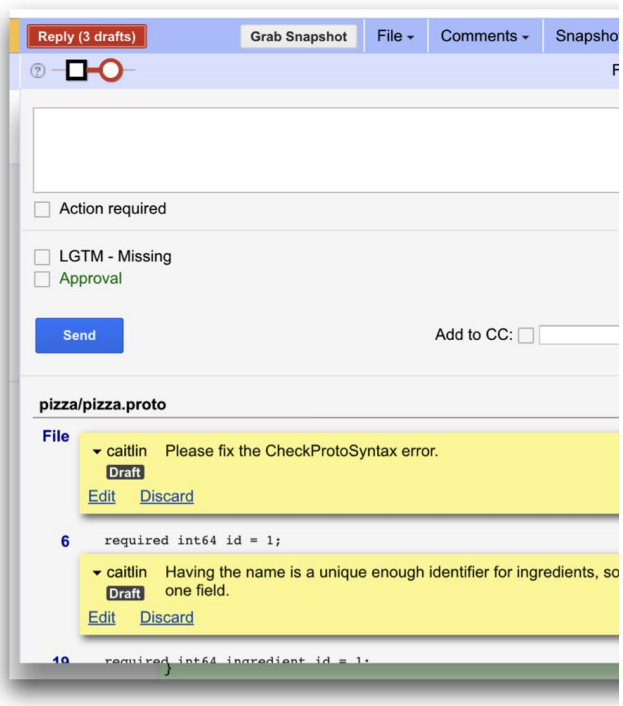


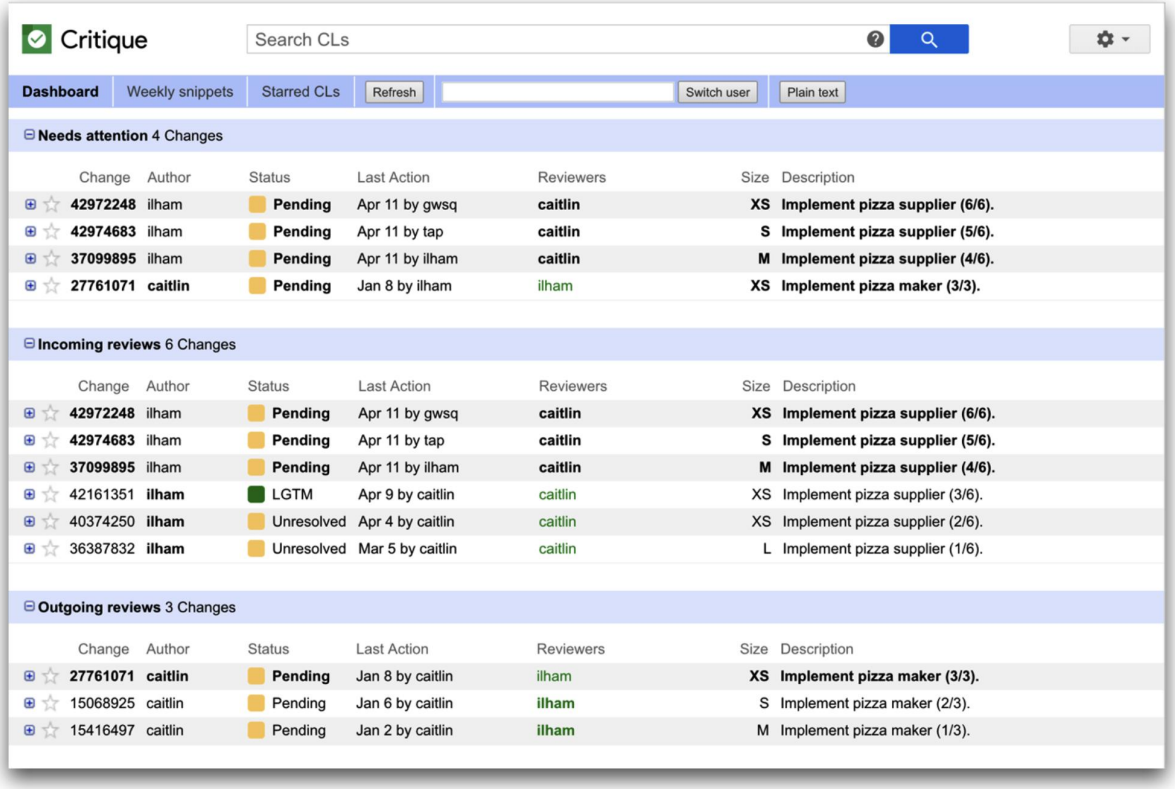
Modern Code Review: A Practice at Google, ICSE SEIP 2018

代码审查

Practice at Google

- *Addressing Feedback*: 代码作者根据要求修改代码、或者对评审意见作出回应，直到所有评审意见都得到解决





Modern Code Review: A Practice at Google, ICSE SEIP 2018

代码审查

Practice at Google

- Creating: 代码作者对代码进行修改
- Previewing: 代码作者使用工具 (Critique) 来审查代码变化 (包括自动化代码分析工具的结果)，并发送邮件通知评审者
- Commenting: 评审者在基于 Web 的 GUI 界面上给出评审意见 (unresolved comments 作为代码作者的行动项)
- Addressing Feedback: 代码作者根据要求修改代码、或者对评审意见作出回应，直到所有评审意见都得到解决
- Approving: 至少一位评审员批准变更 (标记为 LGTM, Looks Good To Me) 后，可以提交代码

代码审查的关注点

- *Design*: 代码是否与系统的其他部分很好地集成？现在是添加此功能的好时机吗？
- *Functionality*: 代码是否符合开发者的意图？开发者的意图对代码的用户是否有益？
- *Complexity*: 代码是否已经超过它原本所必须的复杂度 (避免过度设计和优化)？其他开发者是否能容易地理解并复用该代码？
- *Tests*: 代码是否有配套的正确、合理且有用的测试用例 (单元测试、集成测试、端到端测试等)？
- *Naming*: 开发人员是否为所有变量、函数、类等都选择了合适的名字？

https://google.github.io/eng-practices/review/reviewer/

代码审查的关注点

- *Comments*: 开发者是否用清晰、易理解的语言撰写了注释？这些注释是否有必要？
- *Style and Consistency*: 代码是否符合预定的编码风格？
- *Documentation*: 开发者是否对应更新了和代码改动相关的所有文档 (*e.g., README*) ？
- *Every Line*: 仔细评审每一行代码，并确定你理解所有代码在做什么 (如果难于理解，应将此反馈给开发者)
- *Context*: 有时需要查看整个文件以确保改动是否合理
- *Good Things*: 对良好实践进行鼓励和赞赏

https://google.github.io/eng-practices/review/reviewer/

代码审查的一些参考原则

Be polite, constructive, and positive

- 代码审查的目的是接受并部署代码，而不是拒绝代码 (没有“完美”的代码，只有更好的代码)
- 技术事实和数据优于个人意见和偏好 (避免讽刺和羞辱)
- 审查意见应该具体并且有建设性 (建议可能的修改)
- 遵循合适的代码审查指南
- 及时进行评审并提交回

2 集成测试定义

- 在单元测试的基础上，我们通常需要对由经过单元测试的模块组装起来形成的一个子系统进行的测试，这样的测试被称为子系统测试。子系统测试时重点测试模块的接口。
- 而对由经过测试的子系统测试组装成的系统进行的测试则称为系统测试。在系统测试中发现的往往是软件设计中的错误，也可能发现需求说明中的错误。
- 我们不难看出，不论是子系统测试还是系统测试都兼有检测和组装的含义，这样的测试通常就称为集成测试。
- 集成测试又叫组装测试。

非渐增式测试和渐增式测试

- 根据模块组成程序时的两种不同方法，集成测试方法可以分为两类一是非渐增式测试，一是渐增式测试。
- 渐增式测试是指把下一个要测试的模块同已经测试好的模块结合起来进行测试，测试完以后再把下一个应该测试的模块结合进来测试。这种每次增加一个模块的方法称为渐增式测试。这种方法同时完成单元测试和集成测试。
- 非渐增式测试是先分别测试每个模块，再把所有模块按设计要求放在一起结合成所要的程序。

非渐增式测试和渐增式测试

两种方法的优缺点对比如如下：

第一、由于渐增式的测试方法是利用已测试过的模块作为部分测试软件，因此编写测试软件的工作量比较小。而非渐增式测试方法分别测试每个模块，需要编写的测试软件通常比较多，所需工作量较大。

第二、渐增式测试可以较早发现模块间的接口错误。非渐增式测试最后才把模块组装在一起，因此接口错误发现较晚。

第三、如果发现错误，渐增式测试方法较易查找错误原因。因为如果发生错误往往和最近加进来的那个模块有关。而非渐增式测试一下子把所有模块组合在一起，如果发现错误很难发现确诊。

第四、渐增式测试方法把已经测试好的模块和新加进来的那个模块一起测试，已测试好的模块可以在新的条件下受到新的检验，使程序的测试更彻底。

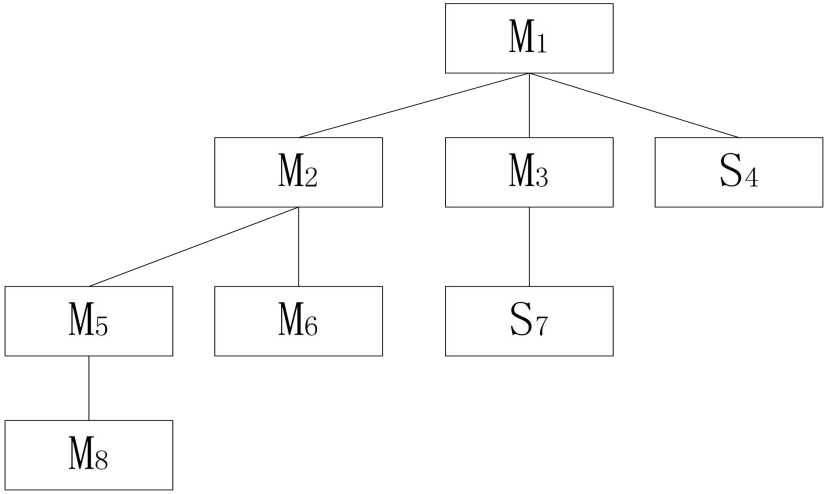
第五、由于测试每个模块时所有已经测试完的模块也要跟着一起运行，因此，渐增式测试需要较多的机器时间。

第六、使用非渐增式测试方法可以并行测试所有模块，因此能充分利用人力，工程进度可以加快。

- 总的来看，渐增式测试方法比较好。

自顶向下集成测试

- 自顶向下的集成测试方法是一个日益为人们广泛采用的组装软件的途径。从主控制模块（“主程序”）开始，沿着软件的控制层次向下移动，从而逐渐把各个模块结合起来。在把附属于（以及最终附属于）主控制模块的那些模块组装到软件结构中去时，或者使用深度优先的策略，或者使用宽度优先的策略。



自顶向下结合

自顶向下集成测试

把模块结合进软件结构的具体过程由下述四个步骤完成：

第一步，对主控制模块进行测试，测试时用存根程序代替所有直接附属于主控制模块的模块；

第二步，根据选定的结合策略（深度优先或宽度优先），每次用一个实际模块代换存根程序（新结合进来的模块往往又需要新的存根程序）；

第三步，在结合进一个模块的同时进行测试；

第四步，为了保证加入模块没有引进新的错误，可能需要进行回归测试（即，全部或部分地重复以前做过的测试）。

自顶向下集成测试

- 自顶向下的结合策略能够在测试的早期对主要的控制或关键的抉择进行检验。在一个分解得好的软件结构中，关键的抉择位于层次系统的较上层，因此首先碰到。如果主要控制确实有问题，早期认识到这类问题是很有好处的，可以及早想办法解决。
- 如果选择深度优先的结合方法，可以在早期实现软件的一个完整的功能并且验证这个功能。早期证实软件的一个完整的功能，可以增强开发人员和用户双方的信心。

自顶向下集成测试

自顶向下的方法讲起来比较简单，但是实际使用时可能退到逻辑上的问题。这类问题中最常见的是，为了充分地测试软件系统的较高层次，需要在较低层次上的处理。然而在自顶向下测试的初期，存根程序代替了低层次的模块，因此，在软件结构中没有重要的数据自下往上流。为了解决这个问题，测试人员有两种选择：

第一，把许多测试推迟到用真实模块代替了存根程序以后再进行；

第二，从层次系统的底部向上组装软件；

自底向上集成测试

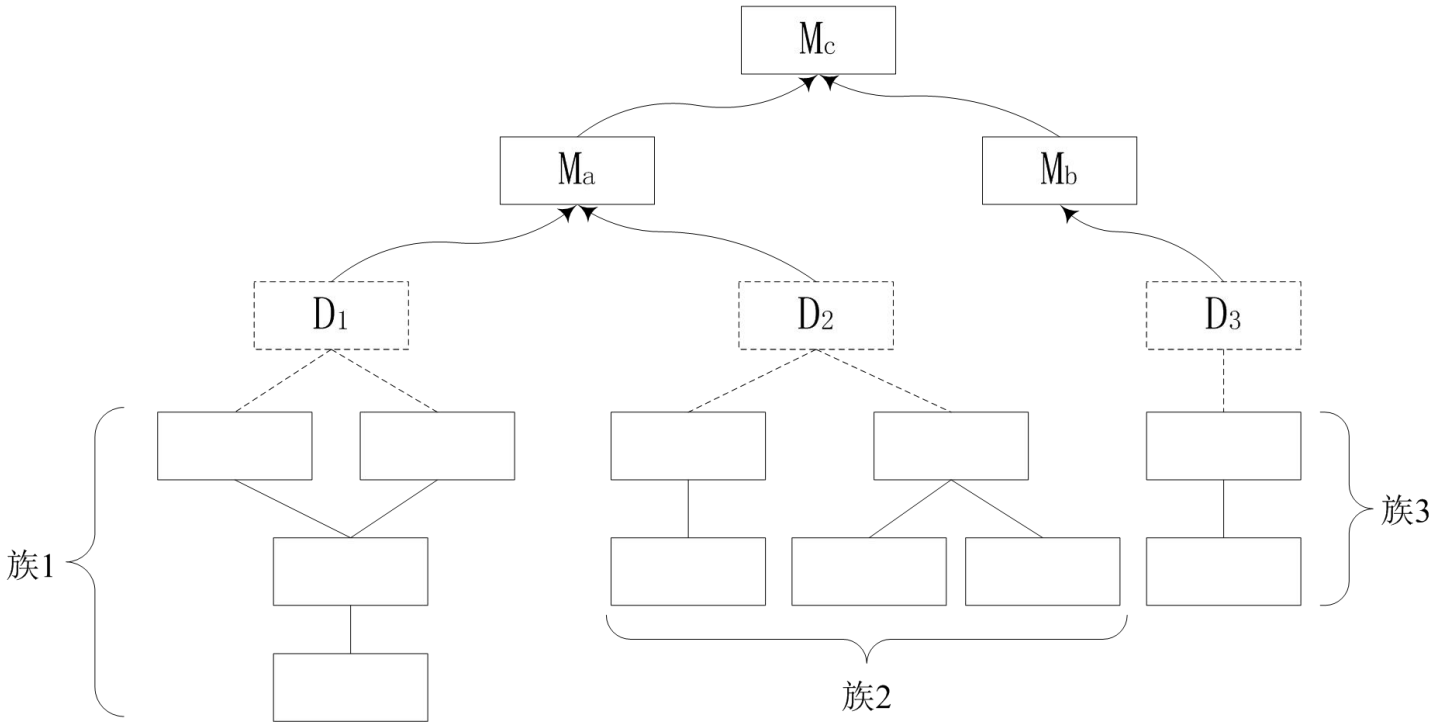
自底向上测试从“原子”模块（即在软件结构最低层的模块）开始组装和测试。因为是从底部向上结合模块，总能得到需要的下层模块处理功能，所以不需要存根程序。用下述步骤可以实现自底向上的结合策略：

(1) 把低层模块组合成实现某个特定的软件子功能的族；

(2) 写一个驱动程序（用于测试的控制程序），协调测试数据的输入和输出；

(3) 对由模块组成的子功能族进行测试；

(4) 去掉驱动程序，沿软件结构自下向上移动，把子功能族组合起来形成更大的于功能族。



自底向上结合

不同集成测试策略的比较

- 上面介绍了集成测试的两种策略，到底哪种方法更好一些呢？一般说来，一种方法的优点企好对应于另一种方法的缺点。
- 自顶向下测试方法的主要优点是不需要测试驱动程序，能够在测试阶段的早期实现并验证系统的主要功能，而且能在早期发现上层模块的接口错误。
- 自顶向下测试方法的主要缺点是需要存根程序，可能遇到与此相联系的测试困难，低层关键模块中的错误发现较晚，而且用这种方法在早期不能充分展开人力。可以看出，自底向上测试方法的优缺点与上述自顶向下测试方法的优缺点刚好相反。

不同集成测试策略的比较

- 在测试实际的软件系统时，应该根据软件的特点以及工程进度安排，选用适当的测试策略。一般说来，纯粹自顶向下或纯粹自底向上的策略可能都不实用，人们在实践中创造出许多混合策略：
- 改进的自顶向下测试方法 基本上使用自顶向下的测试方法，但是在早期，就使用自底向上的方法测试软件中的少数关键模块。一般的自顶向下方法所具有的优点在这种方法中也都有，而且能在测试的早期发现关键模块中的错误；但是，它的缺点也比自顶向下方法多一条，即，测试关键模块时需要驱动程序。
- 混合法 对软件结构中较上层，使用的是自顶向下方法；对软件结构中较下层，使用的是自底向上方法，两者相结合。这种方法兼有两种方法的优点和缺点，当被测试的软件中关键模块比较多时，这种混合法可能是最好的折衷方法。

集成测试

Integration Testing

采用适当的策略将已通过单元测试的模块组装成子系统或系统，并确保各模块组合到一起后能按既定的设计要求运行

•重点测试模块的接口、兼容性和全局数据结构

•基于分解的集成 (Decomposition-Based)：通常采用渐增式方法，根据模块间的结构关系来设计集成策略

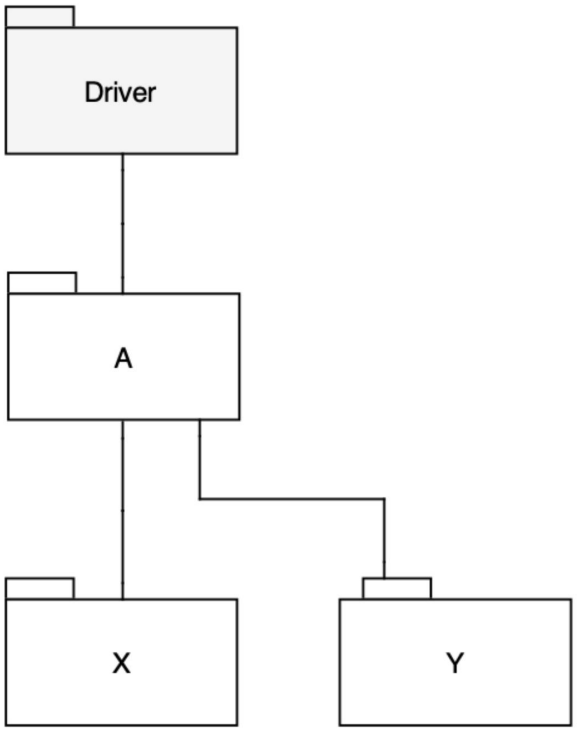
- Big bang
- Bottom up
- Top down
- Sandwich

集成测试

Integration Testing

自底向上集成 (Bottom up)

- 从原子模块开始，逐步向上组装和测试各模块
- 有助于优先验证某个特定子功能的正确性
- 需要一个测试驱动程序 (Driver) 来协调测试数据的输入和输出

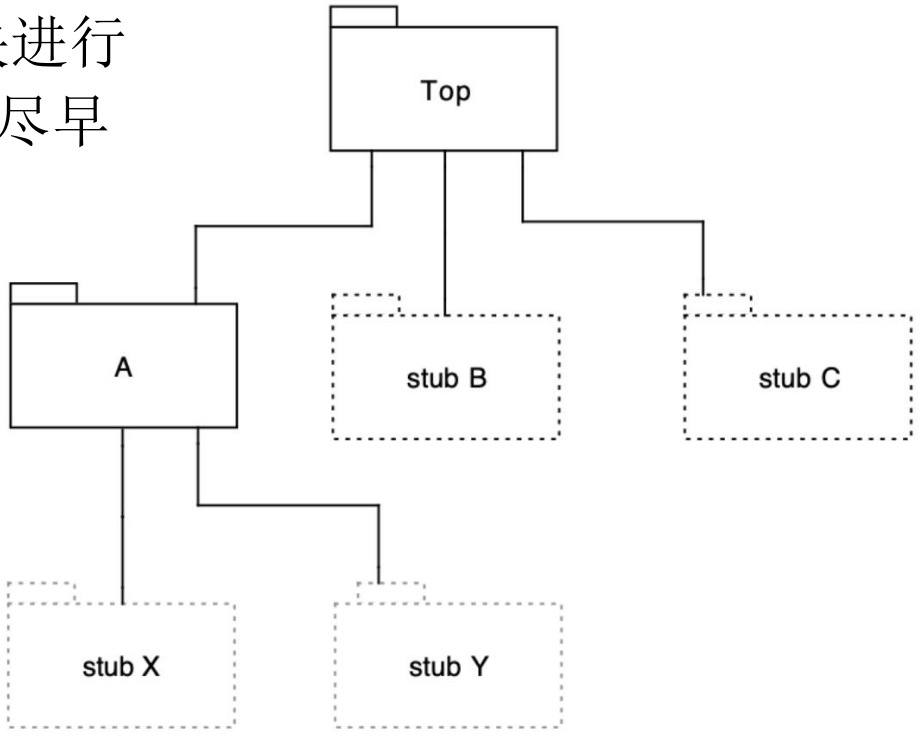


集成测试

Integration Testing

自顶向下集成 (Top down)

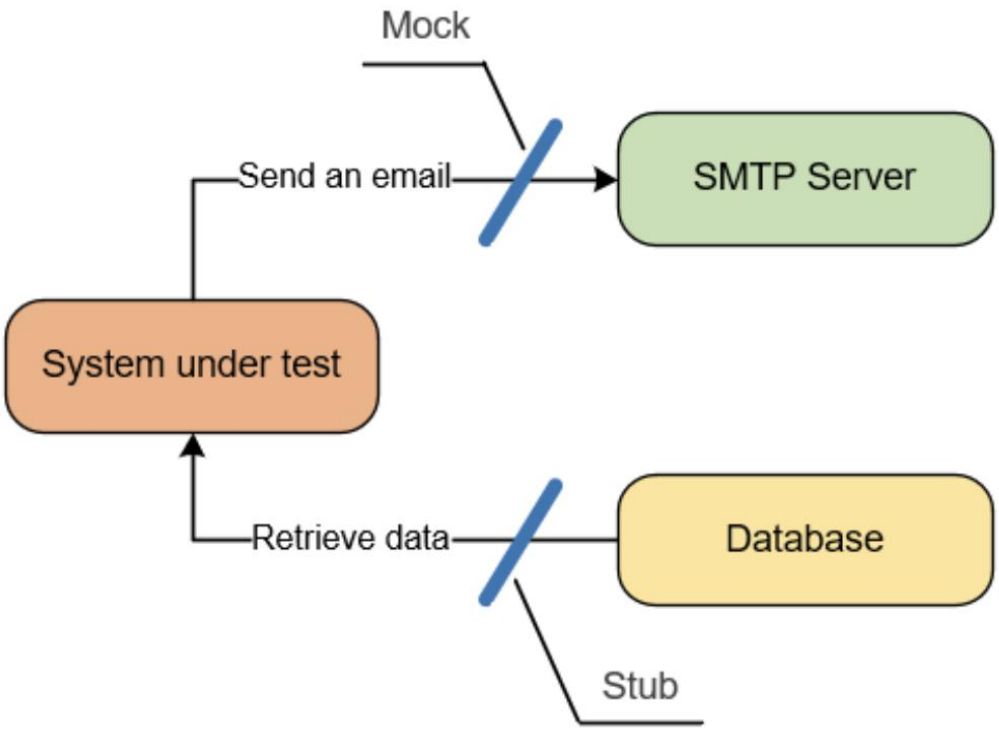
- 从主控制模块开始，沿着软件的控制层次向下移动，逐渐把各个模块结合起来 (深度优先 / 广度优先)
- 有助于尽早对主要的控制模块进行检验 (e.g., 使用深度优先可以尽早验证一个完整功能)
- 需要使用桩程序 (Stub) 和模拟程序 (Mock) 来替代尚未开发的模块



集成测试

Integration Testing

桩程序 (*Stub*) *vs.* 模拟程序 (*Mock*)



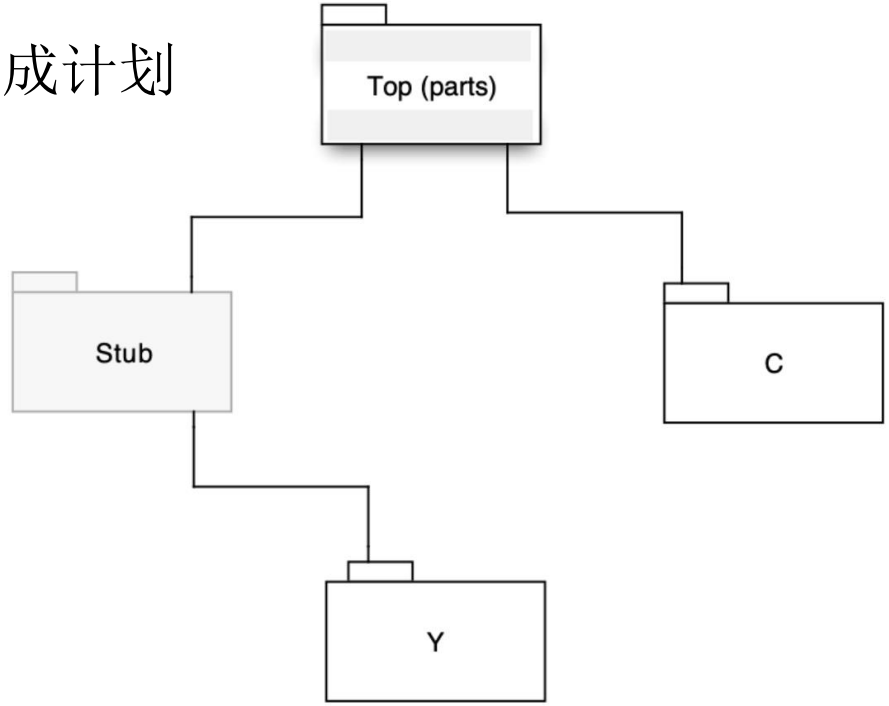
Mocks are for outcoming interaction; stubs—for incoming

集成测试

Integration Testing

混合策略 (Sandwich)

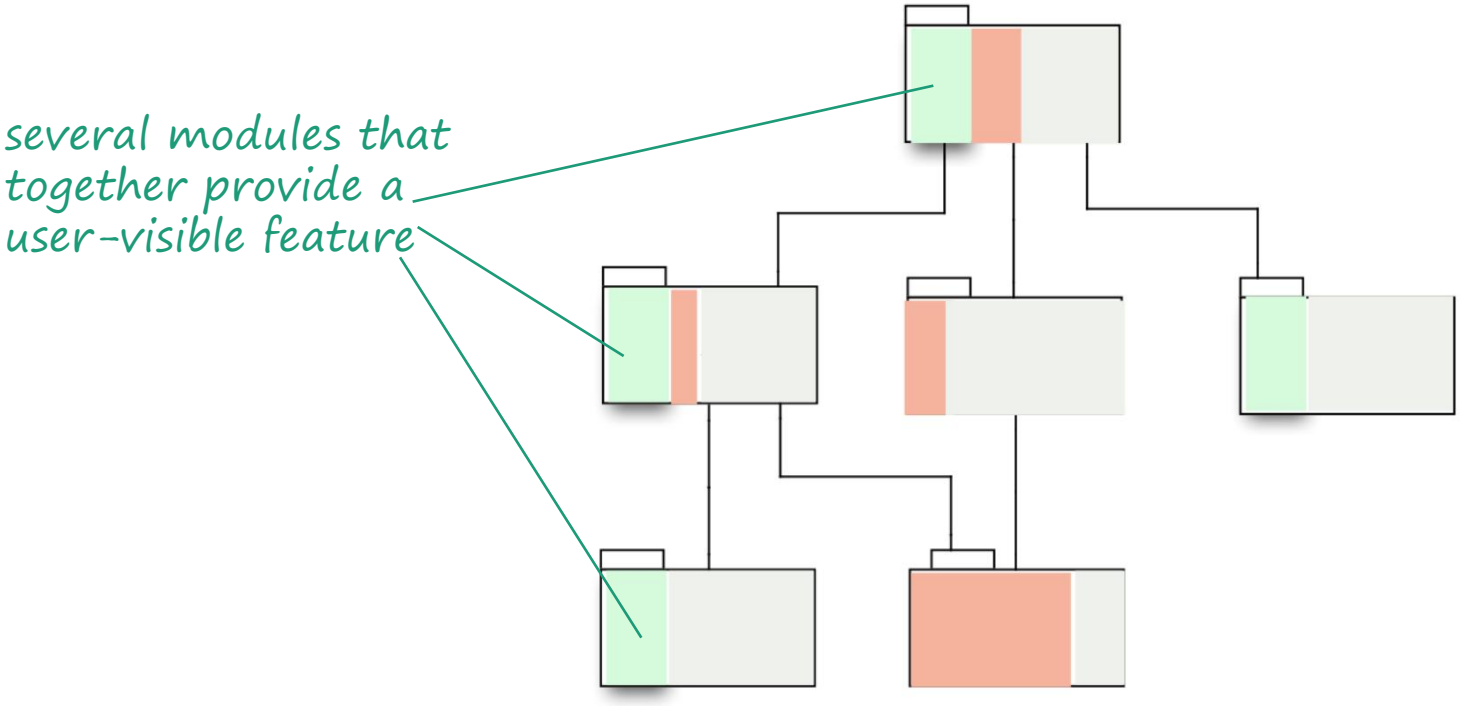
- 对软件结构中上层使用的是自顶向下方法，而对软件结构中下层使用的是自底向上方法
- 更加灵活，但需要更复杂的集成计划



集成测试

Integration Testing

- 基于功能的集成 (*Function-Based*): 每次集成提供一个用户所能感知的具体功能



集成测试

Integration Testing

- 制定集成策略的关键是实现一种面向风险的集成过程 (*risk-oriented process*)
- 通常从风险最高的模块开始集成 (*technical / business risk, ...*)
- 将集成测试看做一个逐步降低风险的活动 (*risk-reduction*)，在集成过程中尽早发现故障

3 系统测试

System Testing

将整个软件系统视为一个整体来进行测试 (whole system, whole specification)

- 重点测试软件产品的各项功能是否满足用户的要求
- 还包括性能、安全性、兼容性、可用性等软件特性及方面的测试 (we will discuss these aspects later)

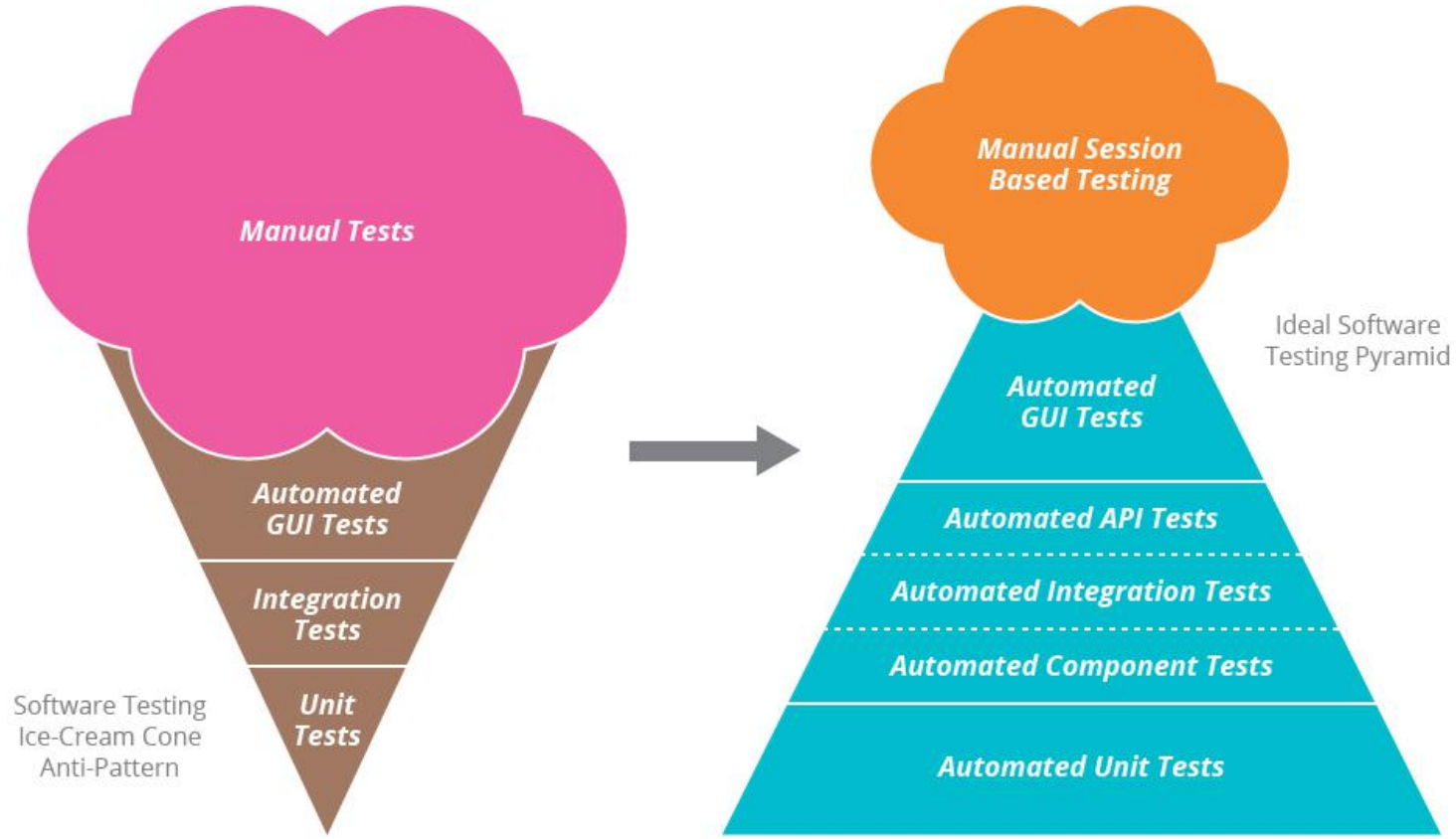
软件不同开发阶段的测试

关注点	单个模块的行为	模块集成和交互	整个系统的功能
测试用例设计	模块规格说明	体系结构和设计规格说明	需求规格说明
代码能见度	所有代码细节	部分代码细节 + 接口	通常无需代码细节
测试依赖环境	在依赖 *drivers* 或 *stubs* 时可能会十分复杂，同时还有 *test oracle*	取决于体系结构和集成策略	主要是 *test oracle*，有时也会依赖特定的执行环境 (*e.g., embedded system*)

Testing Pyramid

软件开发不同阶段的测试方法各自应投入多少时间/资源？

no scientific evidence, but widely accepted by the community



4 冒烟测试

Smoke Testing

Daily build and smoke test (Microsoft)

- 一种将代码更改提交到代码库之前的验证过程
  - 把一个软件项目的所有的最新的代码从配置库中取出，然后从头进行编译、链接和运行
  - 在每日 build 建立后，对系统的基本功能进行测试
- 最小化集成风险、简化故障诊断
- 在现代 CI/CD 流程中，通常在每次代码提交后都会执行测试

150 million test executions / day @ Google (averaging 35 runs / test / day)

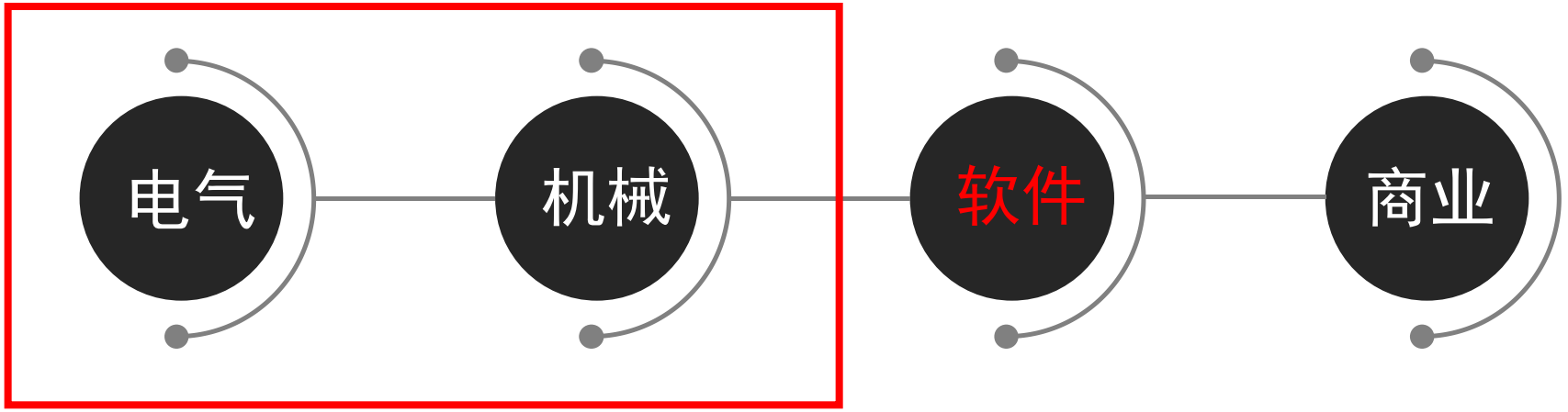
Smoke Testing - 概念

- 管道是否泄漏、乐器键盘是否正确安装、电路是否短路、软件是否会崩溃、舞台是否可以承受更大压力……
- Wikipedia: confidence testing / sanity testing / build verification test(BVT) / build acceptance test
- 微软: “Daily build and smoke test”

  https://msdn.microsoft.com/en-us/library/ms182613(v=vs.90).aspx

PART ONE 释义

“冒烟测试”源于此做法：对一个硬件或硬件组件进行更改或修复后，直接给设备加电。如果没有冒烟，则该组件就通过了测试。



looking for smoke when powering electrical items for the first time

the practice of using smoke to test for leaks

在软件中，“冒烟测试”这一术语描述的是在将代码更改签入到产品的源树中之前对这些更改进行验证的过程。

testing for market demand of a value proposition prior to building a functioning product or service

PART TWO 执行

“Daily Build and Smoke Test”

daily build 就是把一个软件项目的所有的最新的代码从配置库中取出,然后从头进行编译,链接和运行。更甚者可以再运行测试包对软件的主要功能进行测试，发现并报告错误的整个过程。通常由工具自动完成。冒烟测试就是在每日build建立后，对系统的基本功能进行简单的测试。

A common practice at Microsoft and some other shrink-wrap software companies is the "daily build and smoke test" process. Every file is compiled, linked, and combined into an executable program every day, and the program is then put through a "smoke test," a relatively simple check to see whether the product "smokes" when it runs.

BENEFITS. This simple process produces several significant benefits.

PART TWO 执行

每日构建有时也叫作(Nightly build)，构建服务器首先从CVS服务器上，下载最新的源代码，然后编译单元测试，运行单元测试通过后，编译可执行文件，可执行文件若可运行，并能执行最基本的功能，则认为通过了冒烟测试，这时，构建服务器会把程序打包成安装文件，然后上传到内部网站，第二天一早，测试人员来了以后，会收到构建服务器发来的邮件提示昨晚是否构建成功。若构建成功，则测试人员进行相关的功能测试。所有这些功能的完成，一般是靠编写脚本完成的，目前比较常用的脚本有TCL,PERL,PYTHON及功能较弱的批处理。用这些可以完成系统的每日构建。

冒烟测试就是先保证系统能跑的起来，不至于让测试工作做到一半突然出现错误导致业务中断。目的就是先通过最基本的测试，如果最基本的测试都有问题，就直接打回开发部了，减少测试部门时间的浪费。

Smoke Testing - 方法

- 只关注系统最小功能集合
- 浅且广
- 正面测试

Smoke Testing - 方法

Smoke Testing - 方法

与开发人员一起工作

在冒烟测试前进行代码评审

在"干净"的调试环境中对个人更改版本运行冒烟测试项目

每日构建

Smoke Testing - 实例

- 程序能不能运行？
- 用户接口是否打开？
- 按下主要按键是否有反应？（GUI是否响应？）
- Hello World程序是否运行成功？
- 文件的打开、写入、关闭是否成功？
- 银行账户的存取款是否正常？
- ……

一种适用于网站测试的自动化测试系统

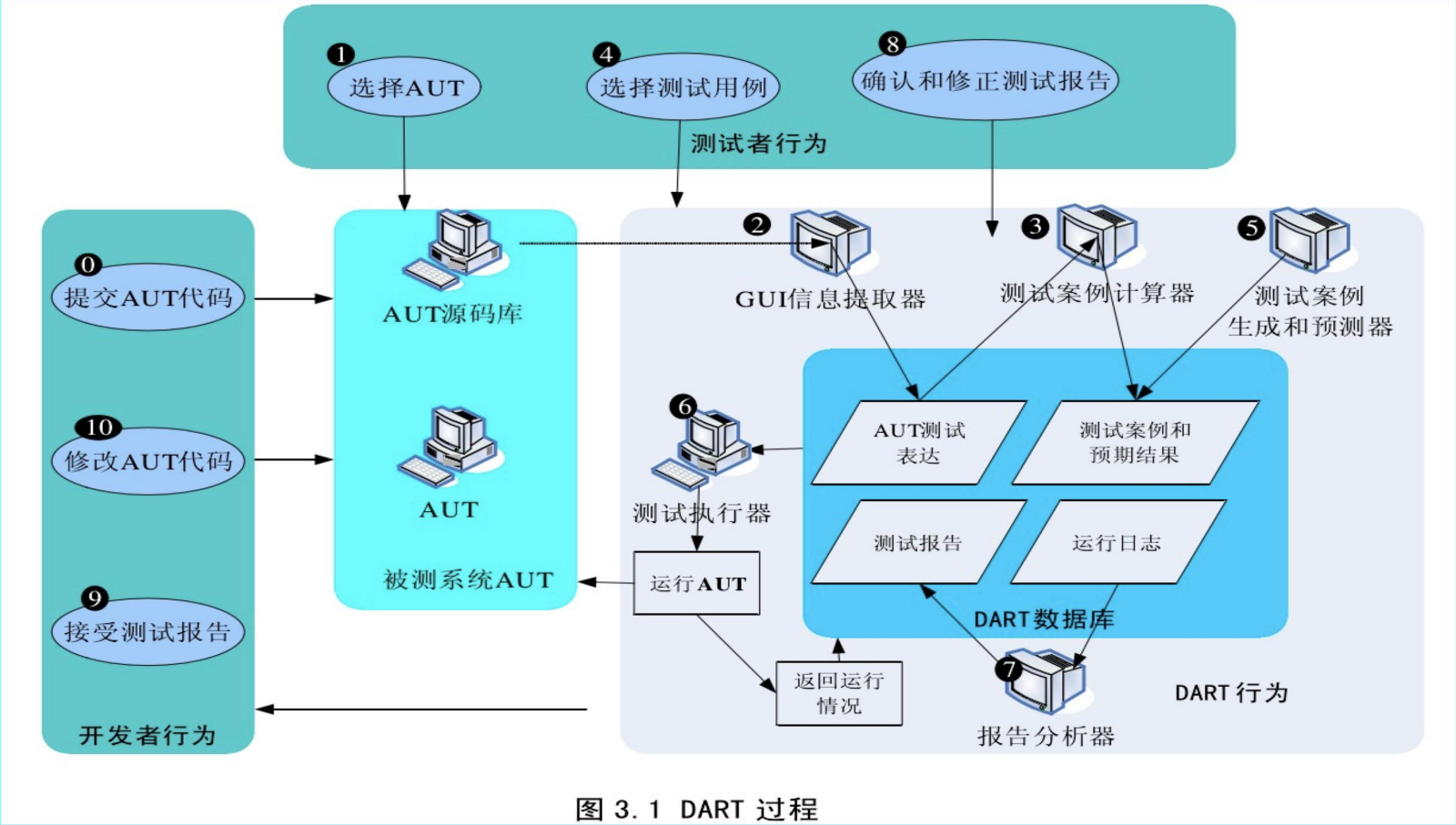


图 3.1 DART 过程

PART THREE 不同阶段的Smoke Testing

在实际的软件测试工作中，Smoke Testing 在软件研发的不同阶段有所不同。大体可以分为三类：

1. **形成集成测试版本以前**Smoke Testing 是随着代码的不断开发必做的一项工作，目的是验证各个单元能够成功执行，并保证测试版本能够顺利集成。
2. **形成集成测试版本以后**在代码 check in 到 daily build accept之前执行 Smoke Testing，以保证新的或者更改过的代码不破坏集成版本的完成性和稳定性。
3. 后期预测试 Bug 的修正后期 daily build **相对稳定**时，针对每个 Bug 所做的 Bug Fix 都要先在干净的 build 中进行 Smoke Testing，测试通过的 Bug Fix 才能 check in 到新的 daily build 中。

Smoke Testing - 优缺点

测试周期短，响应速度快

能最小化集成风险

能减小产品低质量的风险

能简单化错误诊断

冒烟测试的其他解释

理解为“用抽烟的功夫就能完成的测试”？？

冒烟测试和回归测试

把所有粗浅的测试都作为 Smoke Testing?

Smoke Testing 就是 BVT？？ build verification test

5 验收测试

经过集成测试，已经按照设计把所有模块组装成一个完整的软件系统，接口错误也已经基本排除了，接着就应该进一步验证软件的有效性，这就是验收测试的任务。但是，什么样的软件才是有效的呢？软件有效性的一个简单定义是：

- 如果软件的功能和性能如同用户所合理地期待的那样，则软件是有效的。
- 在需求分析阶段产生的文档准确地描述了用户对软件的合理期望，因此是软件有效的标准，也是验收测试的基础。

验收测试的范围

验收测试的目的是向未来的用户表明系统能够像预定要求那样工作。验收测试的范围与系统测试类似，但是也有一些差别，例如：

(1) 某些已经测试过的纯粹技术性的特点可能不需要再次测试；

(2) 对用户特别感兴趣的功能或性能，可能需要增加一些测试；

(3) 通常主要使用生产中的实际数据进行测试；

(4) 可能需要设计并执行一些与用户使用步骤有关的测试。

验收测试必须有用户积极参与，或者以用户为主进行。用户应该参加设计测试方案，使用用户接口输入测试数据并且分析评价测试的输出结果。为了使用户能够积极主动地参与验收测试，特别是为了使用户能有效地使用这个系统，通常在验收之前由开发部门对用户进行培训。

验收测试的范围

验收测试一般使用黑盒测试法。应该仔细设计测试计划和测试过程，测试计划包括要进行的测试的种类和进度安排，测试过程规定用来检验软件是否与需求一致的测试方案。通过测试要保证软件能满足所有功能要求，能达到每个性能要求，文档资料是准确而完整的，此外，还应该保证软件能满足其他预定的要求（例如，可移植性、兼容性和可维护性等等）。

验收测试有两种可能的结果：

(1) 功能和性能与用户要求一致，软件是可以接受的；

(2) 功能或性能与用户的要求有差距。

在这个阶段发现的问题往往和需求分析阶段的差错有关，涉及的面通常比较广，因此解决起来也比较困难。为了确定解决验收测试过程中发现的软件缺陷或错误的策略，通常需要和用户充分协商。

软件配置复查

- 验收测试的一个重要内容是复查软件配置。复查的目的是保证软件配置的所有成分都齐全，各方面的质量都符合要求，文档与程序一致，具有维护阶段所必须的细节，而且已经编排好目录。
- 除了按合同规定的内容和要求；由人工审查软件配置之外，在验收测试的过程中应该严格遵循用户指南以及其他操作程序，以便检验这些使用手册的完整性和正确性。必须仔细记录发现的遗漏或错误，并且适当地补充和改正。

6 回归测试

- 回归测试是指修改了旧代码后，重新进行测试以确认修改没有引入新的错误或导致其他代码产生错误。自动回归测试将大幅降低系统测试、维护升级等阶段的成本
- 回归测试作为软件生命周期的一个组成部分，在整个软件测试过程中占有很大的工作量比重，软件开发的各个阶段都会进行多次回归测试。在渐进和快速迭代开发中，新版本的连续发布使回归测试进行的更加频繁，而在极端编程方法中，更是要求每天都进行若干次回归测试。因此，通过选择正确的回归测试策略来改进回归测试的效率和有效性是非常有意义的。

回归测试

- 回归测试需要时间、经费和人力来计划、实施和管理。为了在给定的预算和进度下，尽可能有效率和有效力地进行回归测试，需要对测试用例库进行维护并依据一定的策略选择相应的回归测试包。

回归测试测试用例库的维护

- 1、删除过时的测试用例
- 2、改进不受控制的测试用例
- 3、删除冗余的测试用例
- 4、增添新的测试用例

回归测试测试策略

- 再测试全部用例
- 基于风险选择测试
- 基于操作剖面选择测试
- 再测试修改的部分

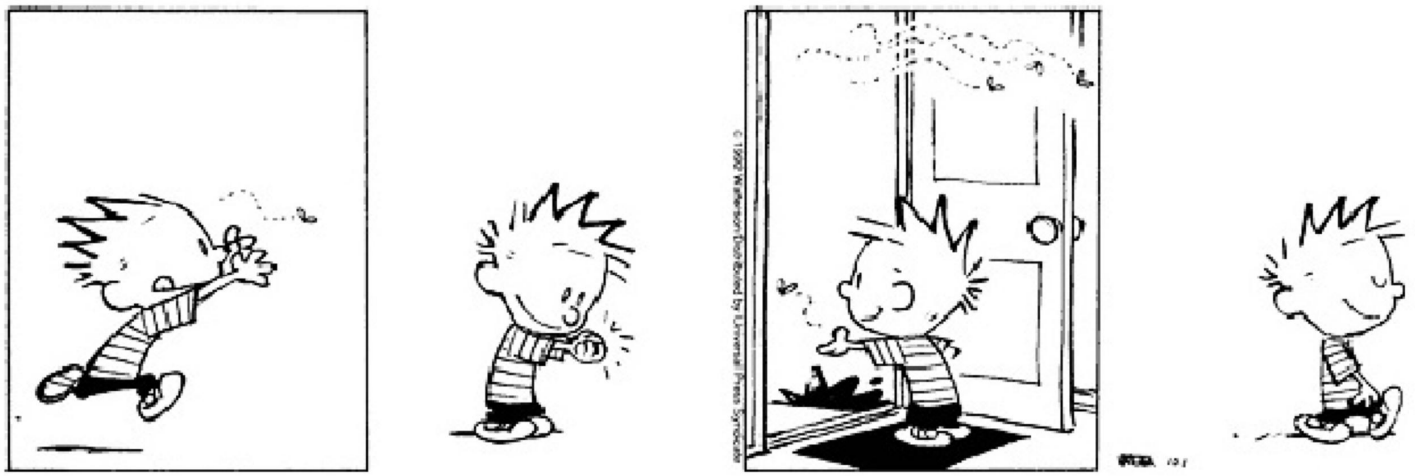
测试过程

- 有了测试用例库的维护方法和回归测试包的选择策略，回归测试可遵循下述基本过程进行:
- (1). 识别出软件中被修改的部分；
- (2). 从原基线测试用例库T中，排除所有不再适用的测试用例，确定那些对新的软件版本依然有效的测试用例，其结果是建立一个新的基线测试用例库T0。
- (3). 依据一定的策略从T0中选择测试用例测试被修改的软件。
- (4). 如果必要，生成新的测试用例集T1，用于测试T0无法充分测试的软件部分。
- (5). 用T1执行修改后的软件。
- 第(2)和第(3)步测试验证修改是否破坏了现有的功能，第(4)和第(5)步测试验证 修改工作本身。

回归测试

Regression Testing

在软件版本更新后重新进行测试，以确认新的代码修改没有引入新的错误 (*regression fault*)



回归测试

Regression Testing

在软件版本更新后重新进行测试，以确认新的代码修改没有引入新的错误 (regression fault)

- 回归测试需要自动化：软件开发的各个阶段都会进行多次回归测试，新版本的连续发布会使回归测试进行的更加频繁 (通常每天都需要进行若干次回归测试)
- 如何进行回归测试？
  - 将已有的所有测试用例都执行一遍 (re-start all)
  - 但是，测试用例集的规模通常会随着软件开发的进行而持续增大，对应的测试执行时间和开销？

回归测试

Regression Testing

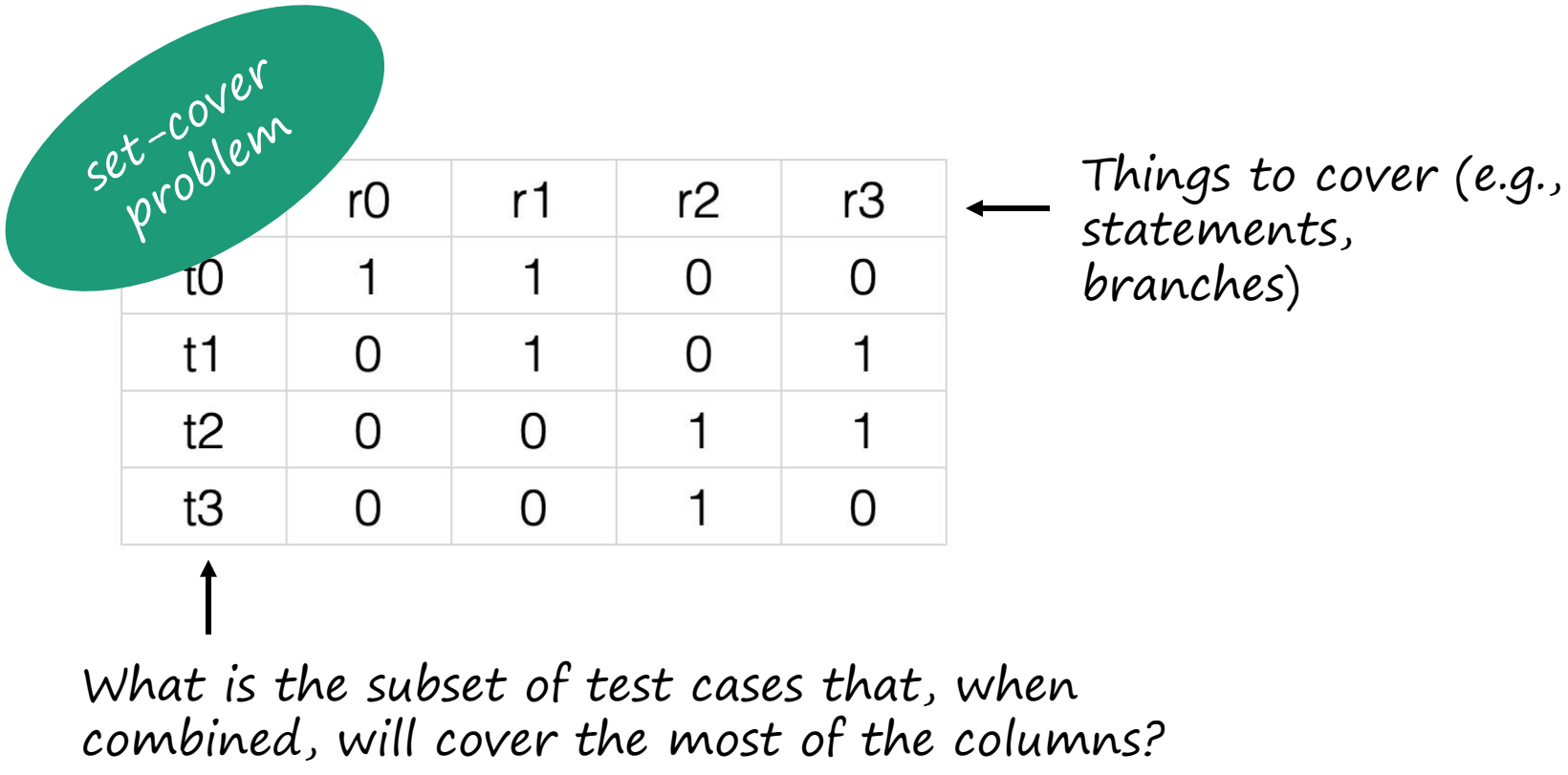
如何在有限的测试资源下 (无法执行所有测试用例) 仍然确保/最大程度提高测试的有效性？

- 测试用例约简 (*Minimisation*)
- 测试用例选择 (*Selection*)
- 测试用例排序 (*Prioritisation*)

测试用例约简

Test Suite Minimisation

测试用例存在冗余：在保证某种测试覆盖率的前提下，从所有测试用例中删除一些冗余的测试用例



测试用例约简

Test Suite Minimisation

测试用例存在冗余：在保证某种测试覆盖率的前提下，从所有测试用例中删除一些冗余的测试用例

```
greedy_minimisation(TestSuite T)
  S = {}
  while(True)
    find t in T s.t. S ∪ {t} covers max. cols.;
    if t exists:
      S = S ∪ {t
      continue
    else:
      break
```

How about the cost?

测试用例约简

Test Suite Minimisation

测试用例存在冗余：在保证某种测试覆盖率的前提下，从所有测试用例中删除一些冗余的测试用例

	r0	r1	r2	r3	Time
t0	1	1	0	0	2
t1	0	1	0	1	3
t2	0	0	1	1	7
t3	0	0	1	0	3

Will {t0, t2} still be the best choice?

测试用例约简

Test Suite Minimisation

测试用例存在冗余：在保证某种测试覆盖率的前提下，从所有测试用例中删除一些冗余的测试用例

```
greedy_minimisation(TestSuite T)
  S = {}
  while(True)
    find t in T w/ max. Δcov./cost of t;
    if t exists:
      S = S ∪ {t}
      continue
    else:
      break
```

测试用例选择

Test Suite Selection

并非所有测试用例都与变化相关：从所有测试用例中挑选出一部分，以使得所有与代码改动相关的部分都能被检测到

P

```
void test_me(int x)
{
  if(x == 0){
    print “Bwahaha”;
  }
  return;
}
```

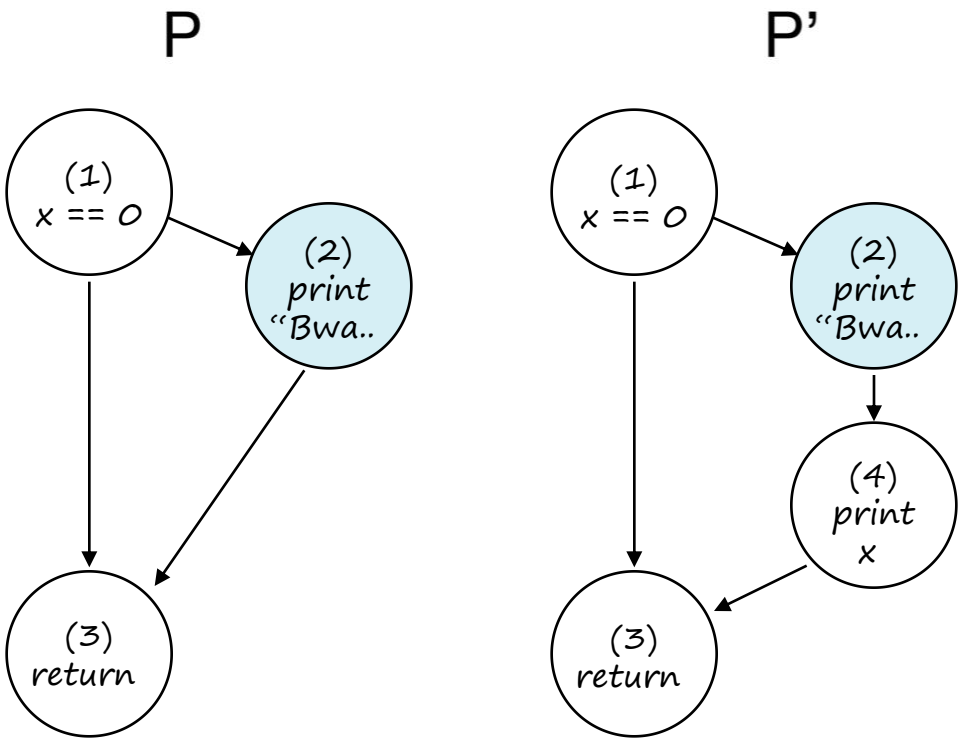
P’

```
void test_me_again(int x)
{
  if(x == 0){
    print “Bwahaha”;
    print x;
  }
  return;
}
```

测试用例选择

Test Suite Selection

并非所有测试用例都与变化相关：从所有测试用例中挑选出一部分，以使得所有与代码改动相关的部分都能被检测到



P and P' start to differ from node 2

Any test that execute node 2 should be selected to test the changes

Test suite of P

- x = 0 {1, 2, 3} ✓
- x = 1 {1, 3}

测试用例选择

Test Suite Selection

并非所有测试用例都与变化相关：从所有测试用例中挑选出一部分，以使得所有与代码改动相关的部分都能被检测到

- 测试用例选择技术应该尽可能 "safe" (i.e., 将所有最有可能触发缺陷的测试用例都执行一遍)
- 收集测试用例的执行路径很多时候并不容易
  - 与系统配置相关的修改通常难于分析 (non-executable modifications)
- 确保 "safety" 也会面临巨大的测试开销
  - 测试用例数目仍然会十分庞大

测试用例排序

Test Suite Prioritisation

测试用例的重要程度不同：优先执行那些更有可能发现故障的测试用例，从而在测试停止时获得最大的收益

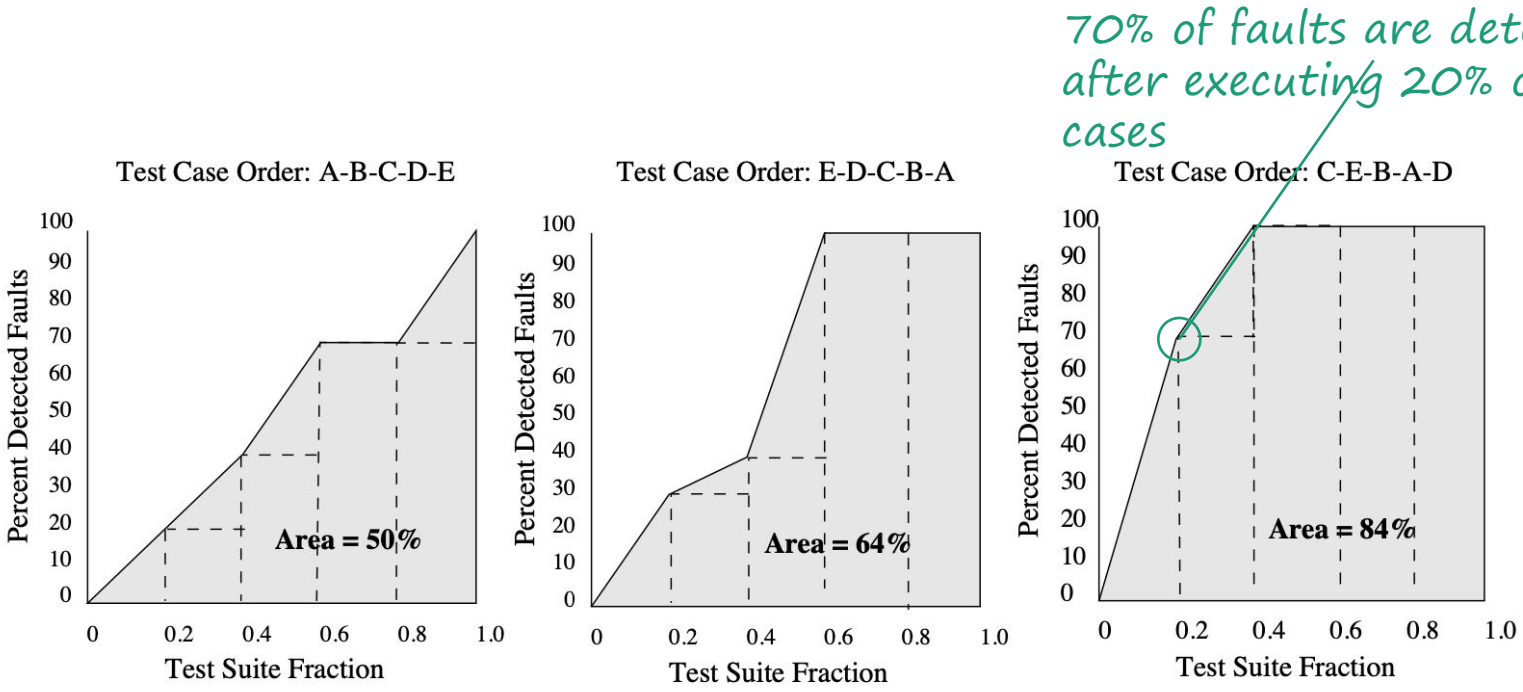
	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9
t0	x				x					
t1	x				x	x	x			
t2	x	x	x	x	x	x	x			
t3					x					
t4								x	x	x

由于无法在测试执行前度量测试的故障检测能力，需要依赖某种 *surrogate metric* (*e.g., structural coverage*) 来进行排序，即最大化某种 *surrogate metric* 的实现

测试用例排序

Test Suite Prioritisation

测试用例的重要程度不同：优先执行那些更有可能发现故障的测试用例，从而在测试停止时获得最大的收益



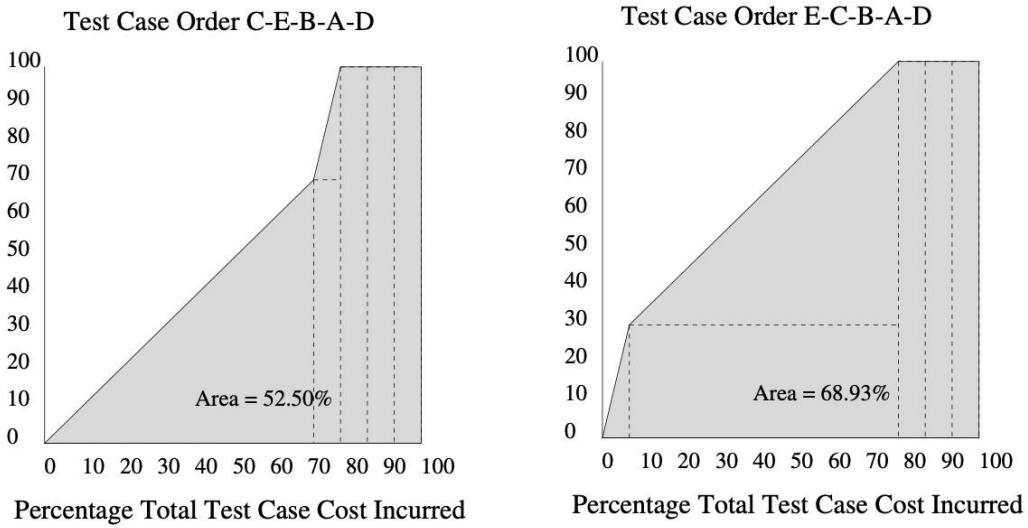
度量一个测试执行序列的故障检测速度：APFD

(Average Percentage of Fault Detection)

测试用例排序

Test Suite Prioritisation

测试用例的重要程度不同：优先执行那些更有可能发现故障的测试用例，从而在测试停止时获得最大的收益



Test Cost 也是测试用例排序中需要考虑的一个关键因素 ($APFD_c$)

回归测试

Regression Testing

如何在有限的测试资源下 (无法执行所有测试用例) 仍然确保/最大程度提高测试的有效性？

- 测试用例约简 (Minimisation) — *Redundancy*

  确实能降低测试成本，但一切都取决于你选择的覆盖标准

- 测试用例选择 (Selection) — *Safety*

  希望以一种保守的方式来执行所有与变化相关的测试，但仍然可能面临庞大的测试用例集

- 测试用例排序 (Prioritisation) — *ASAP / Surrogate*

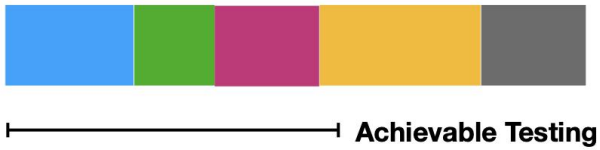
  最大化测试的故障检出速度，但必须依赖某种足够有效的指标

回归测试

Regression Testing

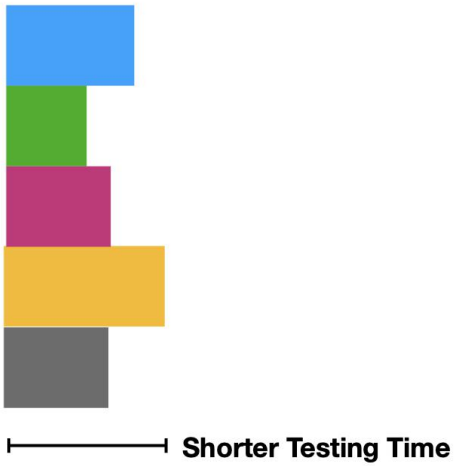
测试用例约简、选择和排序技术的一个假设是测试资源受限，如果在测试时没有测试资源的限制 *(e.g., use cloud computing)* ？

Single Testing Machine



If all test cases have to be executed sequentially, we have to optimise the regression testing process as much as possible, to get the most out of the limited testing resources.

Elastic Cloud Resource / Parallel Instances



If all test cases can be executed in parallel, the time needed for testing is only as long as the longest test execution.

回归测试

Regression Testing

- 测试用例约简和选择：仍然有助于降低测试成本
- 测试用例排序：取决于 *CI* 测试的不同阶段
  - 开发者在代码提交前所做的测试 (*Pre-Commit Testing*)
    - 通常在某个单一系统上执行，排序技术仍然有益

回归测试

Regression Testing

- 测试用例约简和选择：仍然有助于降低测试成本
- 测试用例排序：取决于 CI 测试的不同阶段
  - 开发者在代码提交前所做的测试 (Pre-Commit Testing)
  - 代码提交后 CI 系统自动执行测试 (Post-Commit Testing)
    - CI pipeline 上通常会有很多 commits，很容易 flooded
      (Google engineers have to wait up to 9 hours to receive test results from the CI pipeline)
    - 对待测试 commits 进行排序
      (e.g., commits relevant to test cases that have recently failed are given higher priority)

7α测试

- α测试是由一个用户在开发环境下，进行的测试，也可以是开发机构内部的用户在模拟实际操作环境下进行的测试。α测试的目的是评价软件产品的FLURPS（即功能、局域化、可使用性、性能和支持），尤其注重产品的界面和特色。α测试人员是除产品开发人员之外首先见到产品的人，他们提出的功能和修改意见是特别有价值的。α测试可以从产品编码结束之时开始，或在模块测试完成之后开始，也可在验收测试过程中产品达到一定的稳定和可靠程度以后再开始。

7 α测试

- 一方面要提高国内企业对软件测试的重视程度，另一方面要壮大软件测试队伍，提高测试人员的素质。
- 其次是要善于学习与吸收。国外有完善的测试机，有丰富的软件测试经验，有强大的测试工具，有优秀的测试管理水平,这些我们都应好好地学习
- 大力发展第三方的专业测试公司，重视利用第三方的测试力量进行测试

8 β测试

- 经过α测试调整的软件产品称为β版本。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，这些用户返回有关错误信息给开发者。测试时，开发者通常不在测试现场。因而，β测试是在开发者无法控制的环境下进行的软件现场应用。在β测试中，由用户记下遇到的所有问题，包括真实的以及主观认定的，定期向开发者报告。β测试主要衡量产品的FLURPS，着重于产品的支持性，包括文档，客户培训和支持产品生产能力。
- 只有当α测试达到一定的可靠程度时，才能开始β测试。它处在整个测试的最后阶段。同时，产品的所有手册文本也应该在此阶段完全定稿。

Perpetual Beta

Release Early & Release Often: keeping software or a system at the beta development stage for an extended or indefinite period of time

9 λ测试

- λ是第三个阶段，此时产品已经相当成熟，只需在个别地方再做进一步的优化处理即可上市发行
- a third stage of software testing sometimes peformed after beta testing but before commercial release. In gamma testing, the software is believed to be complete and free of errors, but the manuals and packaging may not yet be in final form.

思考题

- 1 简述单元测试的优缺点
- 2 集成测试的策略有几种？简述各自特点
- 3 系统测试的目标是什么？
- 4 验收测试主要测试什么？
- 5 为什么要进行回归测试？
- 6 α测试主要检测软件哪些方面的问题？
- 7 α测试与β测试的区别在哪里？
- 8 冒烟测试有什么作用？