




哈爾濱工業大學  
HARBIN INSTITUTE OF TECHNOLOGY

软件工程  
第五章 需求与测试  
5-3 黑盒测试

王忠杰  
rainy@hit.edu.cn

2017年10月22日

# 主要内容

- 
- 1 软件测试基础
  - 2 测试过程
  - 3 测试方法分类
  - 4 黑盒测试思想
  - 5 等价类划分方法
  - 6 边界值方法



# 1 软件测试基础



# 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。

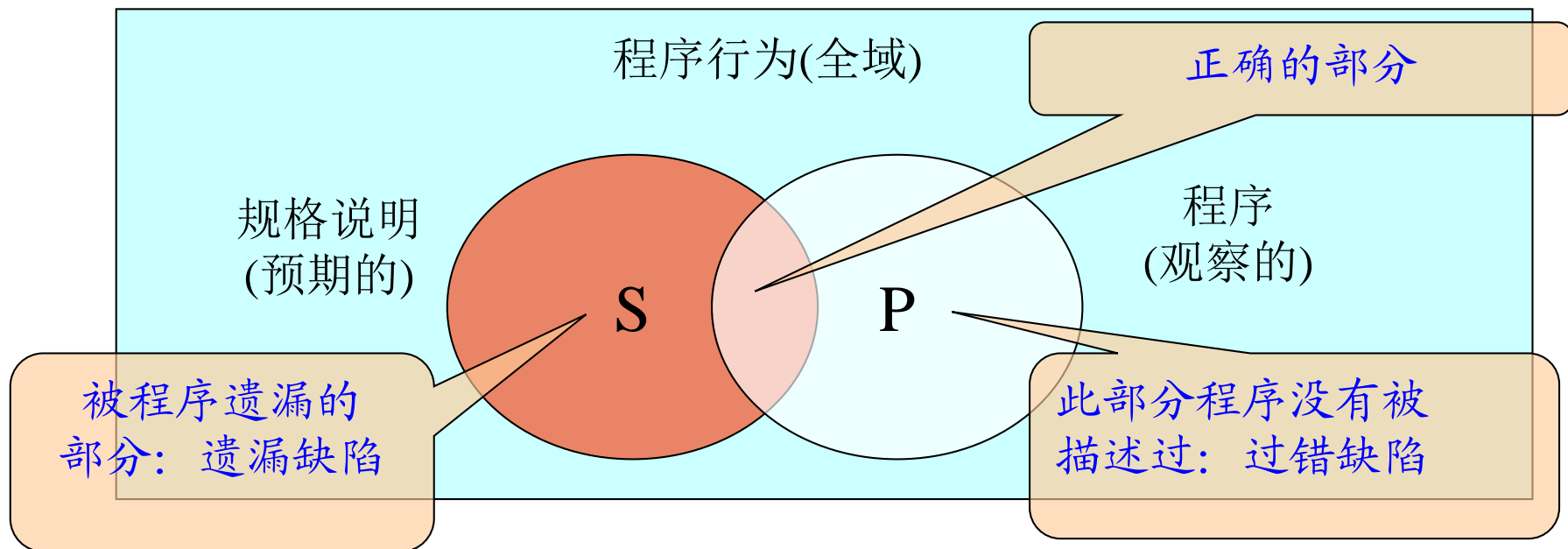
明确提出了“在程序中寻找错误”是测试目的。

- 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。

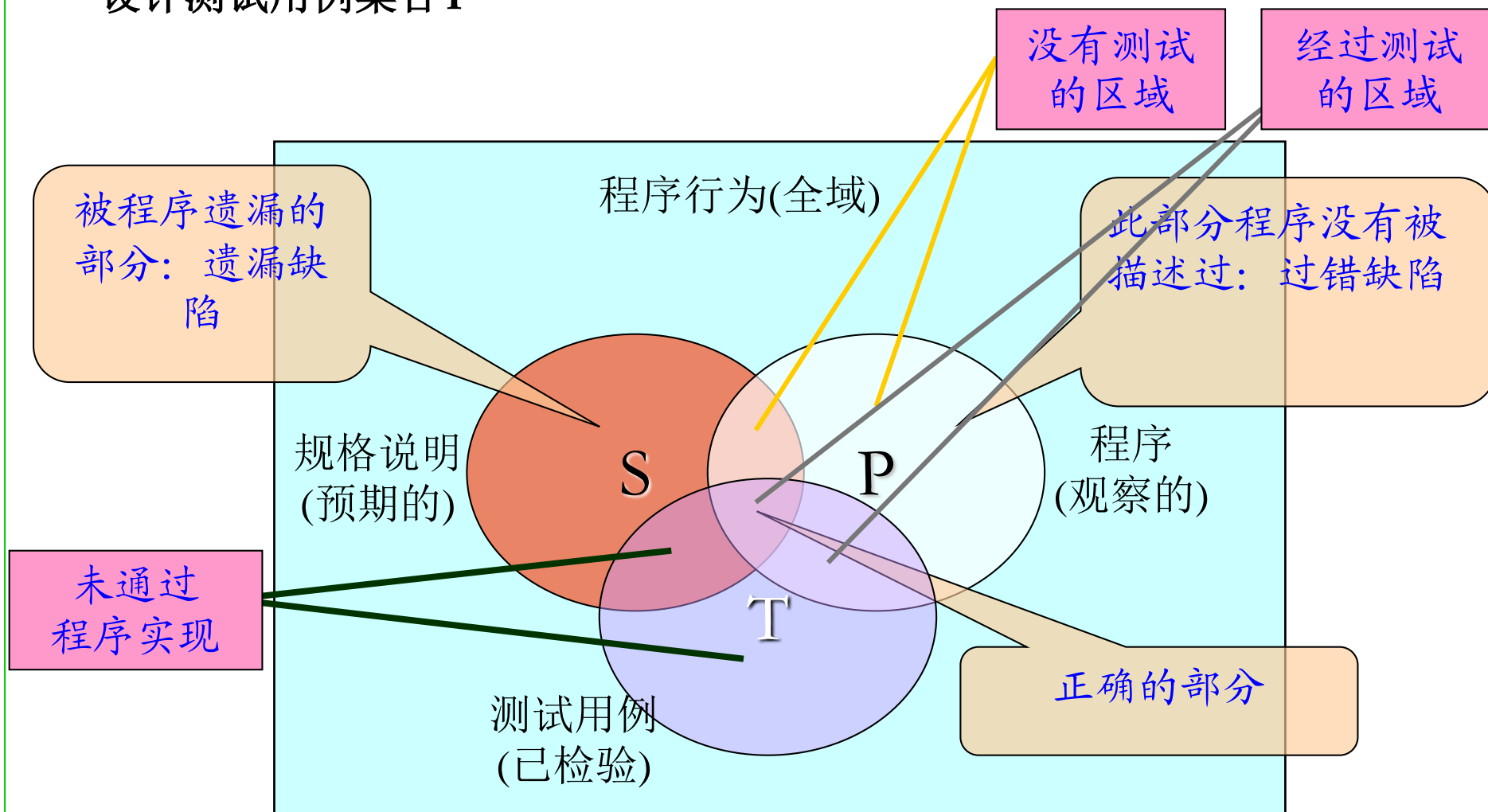
# 用 Venn Diagram 来理解测试

- 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

# 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，逐渐扩大到大规模**
- **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



# 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

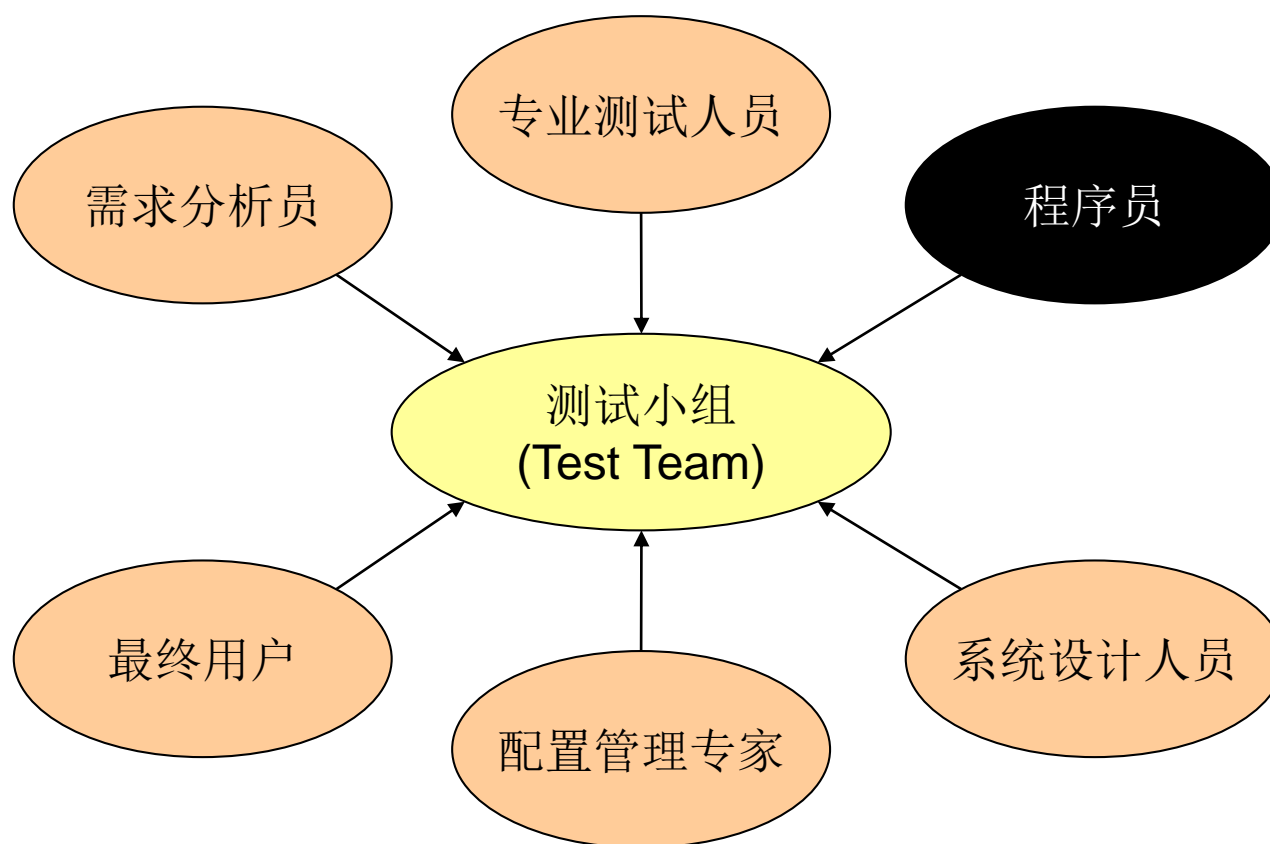
## ■ 测试结果的可判定性:

- 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。

# 软件测试人员



# 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

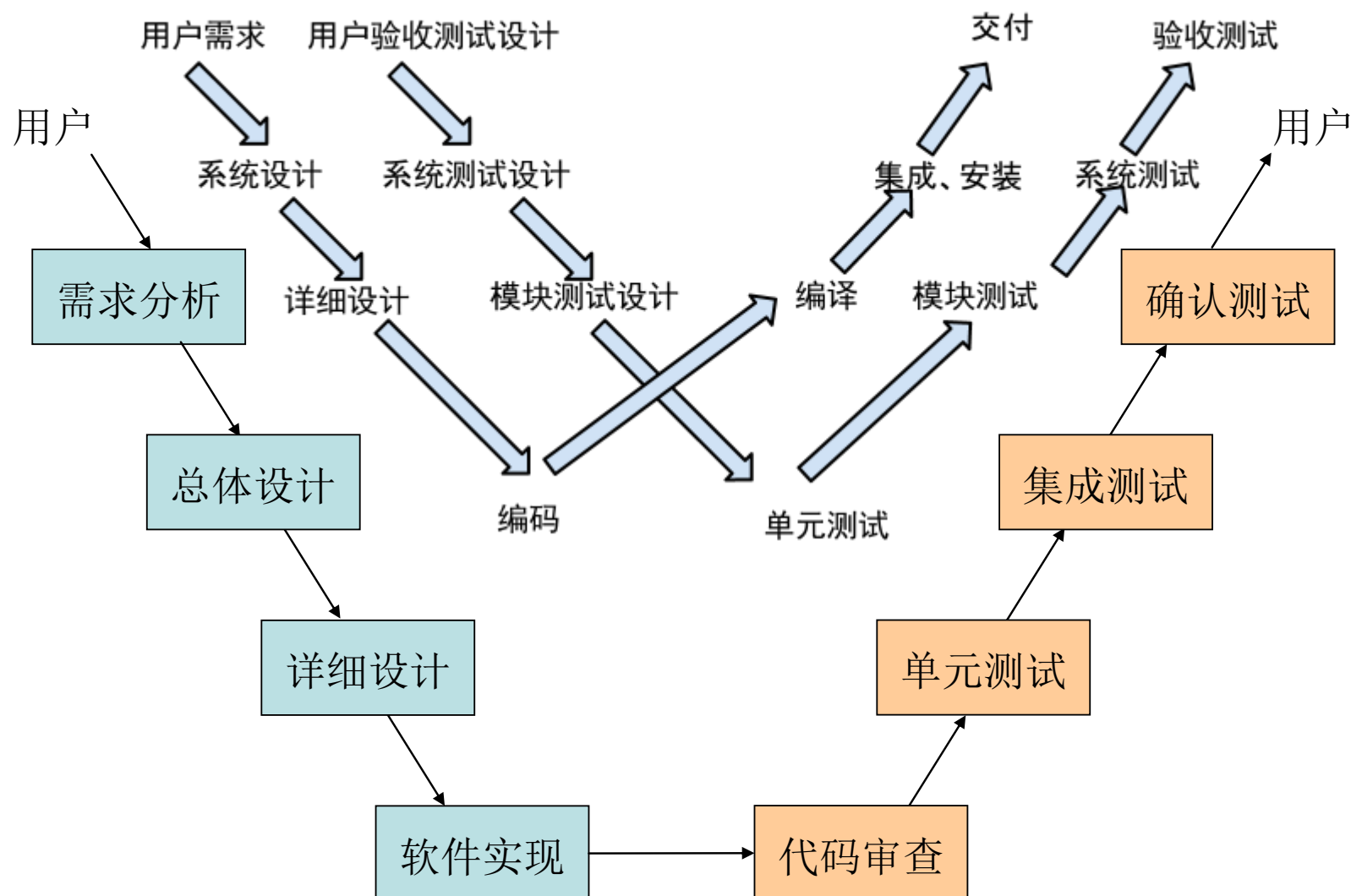
- 一个好的测试人员具有“测试是为了破坏”的观点、捕获用户观点的能力、强烈的质量追求、对细节的关注能力。



## 2 软件测试过程

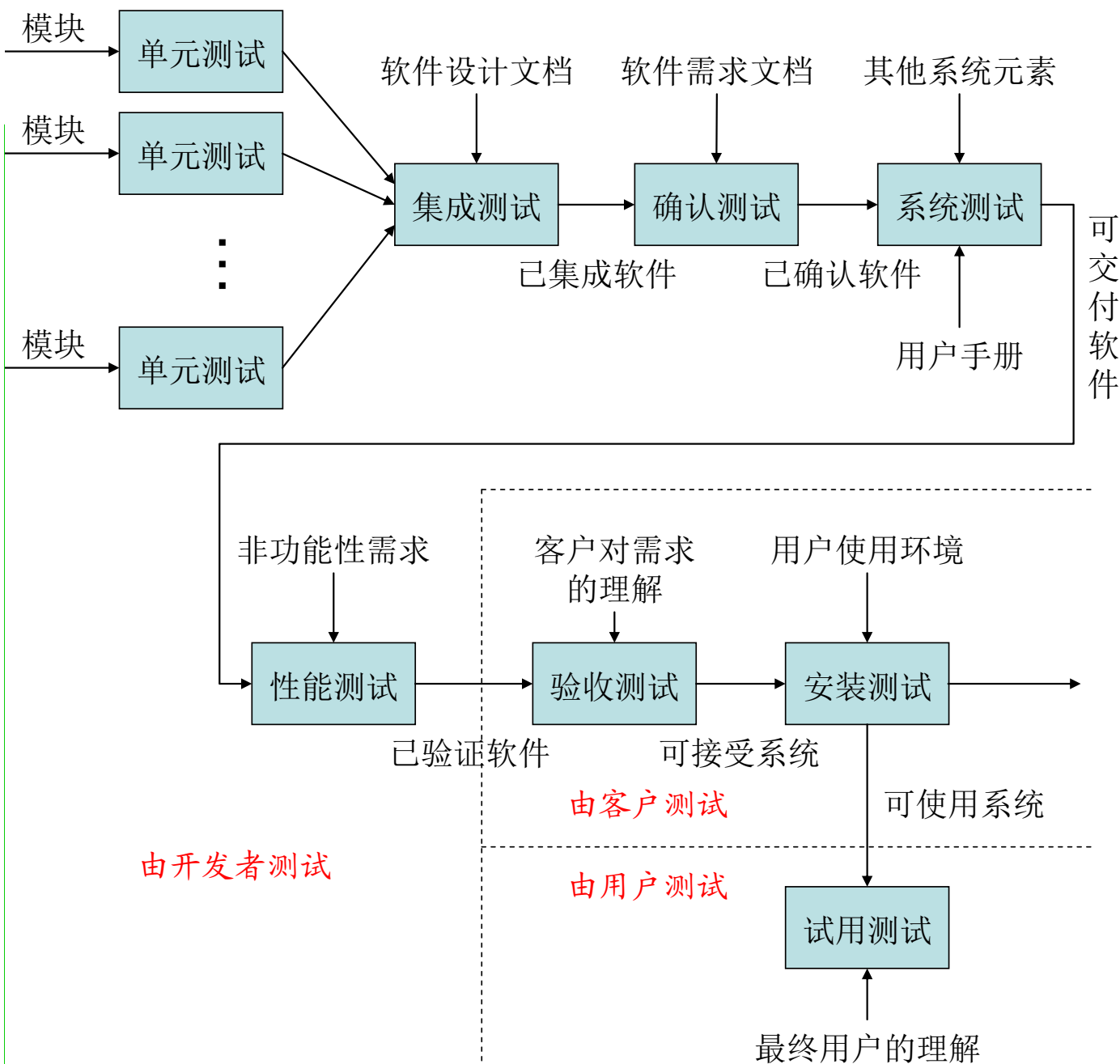


# 软件测试的V模型





## 软件测试活动





### 3 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
- 验收测试      Verification 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

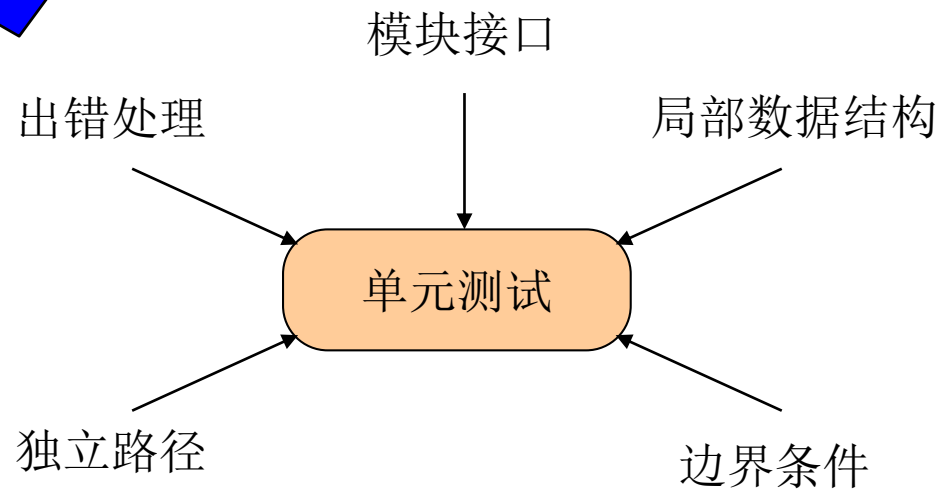
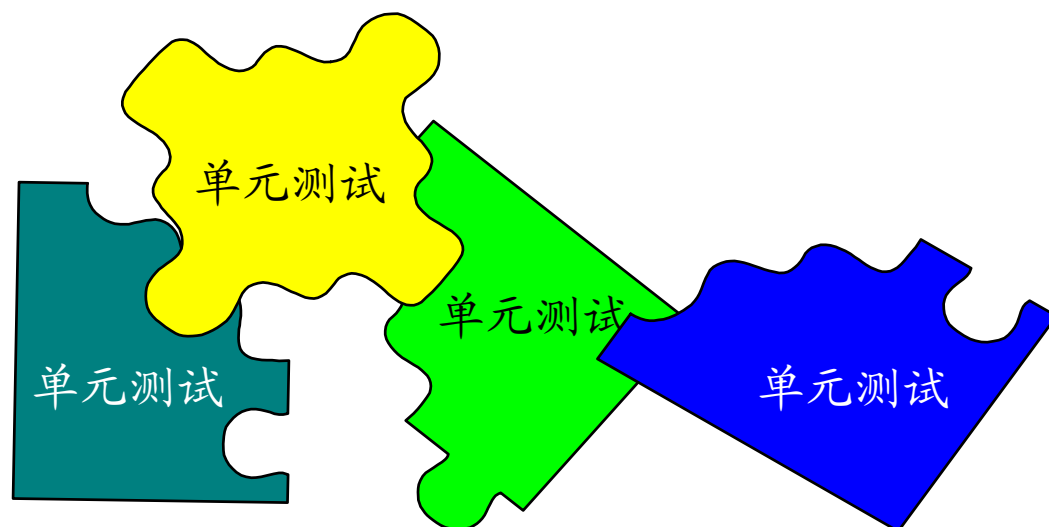
## ■ 单元测试(Unit Testing)

- 单元测试是对软件基本组成单元进行的测试，有时也称“组件测试”。
- 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。

## ■ 单元测试的目的

- 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。

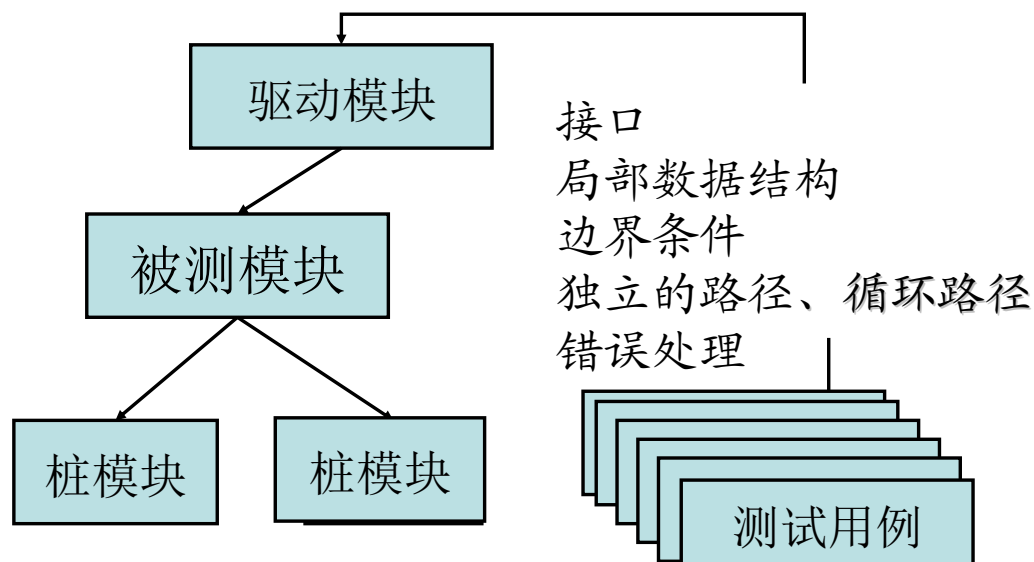
# 单元测试



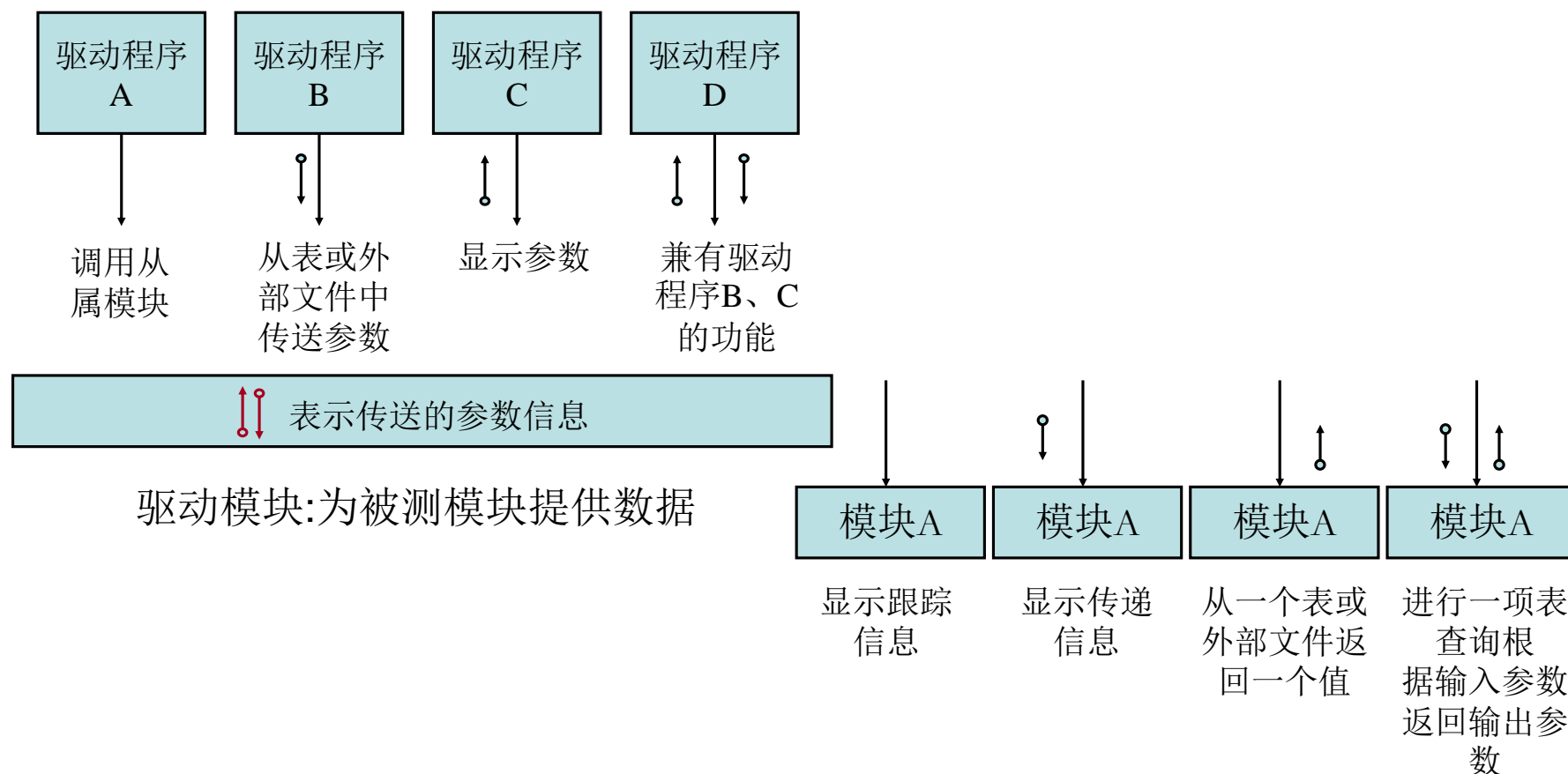
# 单元测试的环境

## ■ 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



桩模块：只做少量的数据操作

## (2) 集成测试

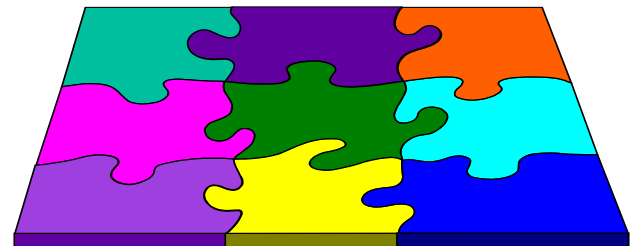
- 每个模块都能单独工作→集成在一起却不能工作

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。
- 集成测试的对象是模块间的接口，其目的是找出在模块接口上和系统体系结构上的问题。





# 集成测试策略

## ■ 集成测试策略

- 基于层次的集成：自顶上下与自底向上
- 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

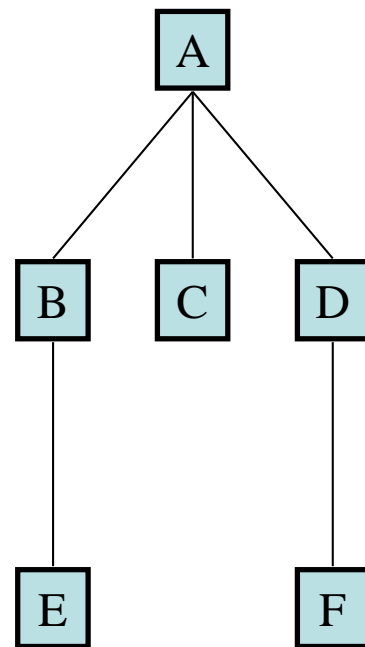
# 集成测试的方法：整体集成

## ■ 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

## ■ 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：

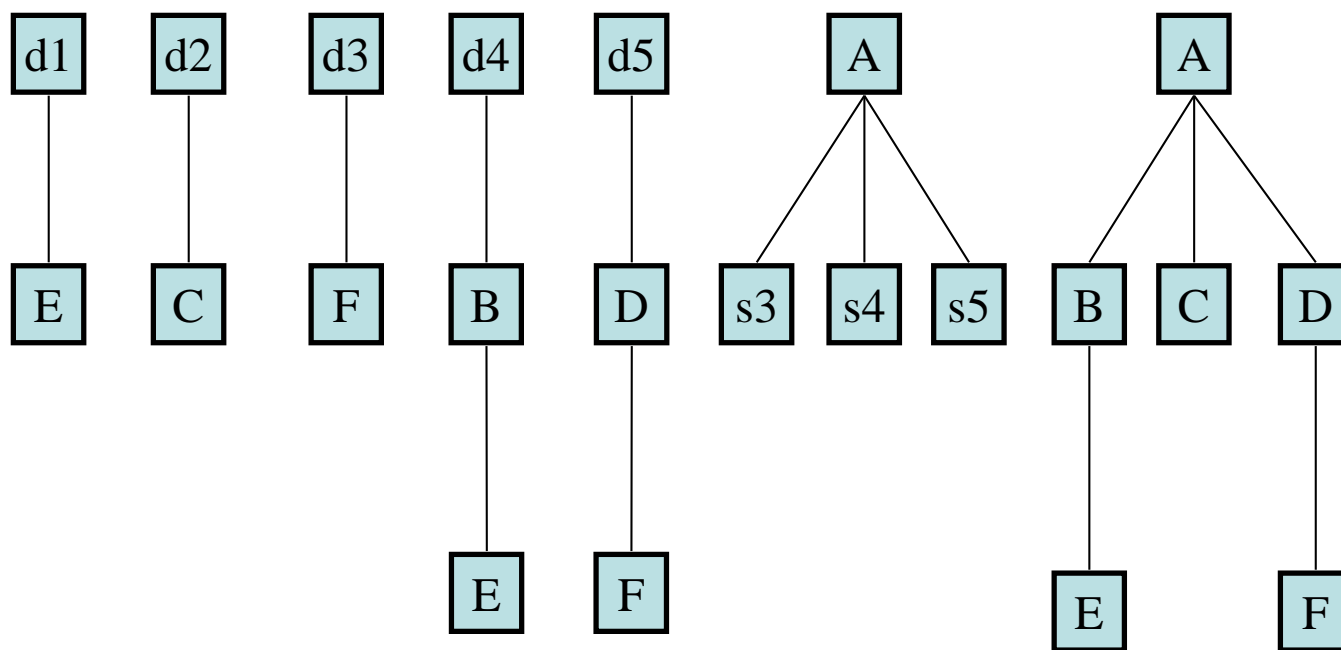
- 效率高，所需人力资源少；
- 测试用例数目少，工作量低；
- 简单，易行；

## ■ 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；

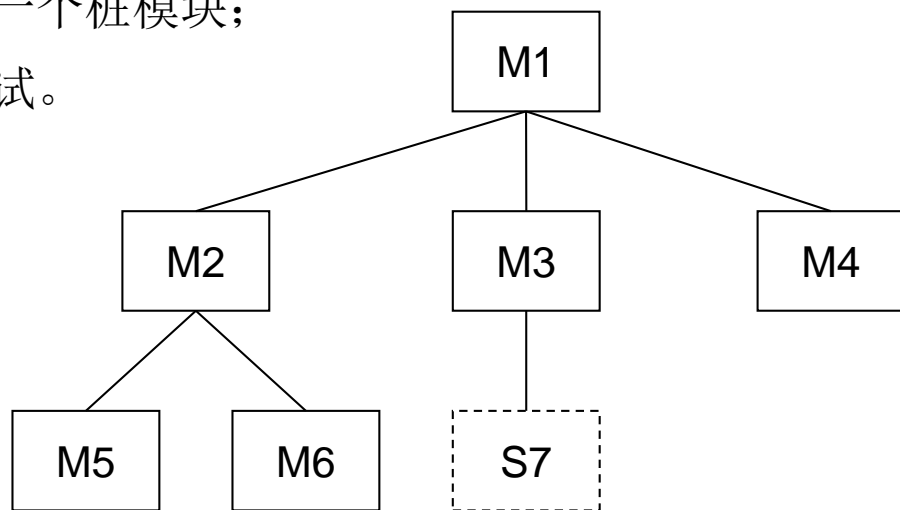
# 集成测试的方法：增量式集成

- 增量式集成测试方法：
  - 逐步将新模块加入并测试



# 自顶向下的增量集成

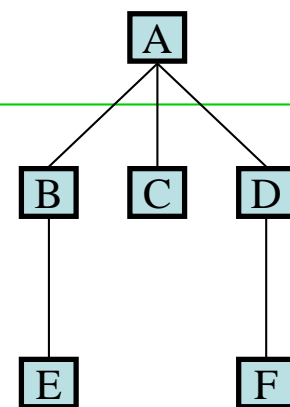
- **自顶向下的集成测试：**从主控模块开始，按软件的控制层次结构，以深度优先或广度优先的策略，逐步把各个模块集成在一起。
- **具体步骤：**
  - 以主控模块作为测试驱动模块，把对主控模块进行单元测试时所引入的所有桩模块用实际模块代替；
  - 依据所选的集成策略(深度优先、广度优先)，每次只替代一个桩模块；
  - 每集成一个模块立即测试一遍；
  - 只有每组测试完成后，才着手替换下一个桩模块；
  - 为避免引入新错误，不断进行回归测试。



# 自顶向下的增量集成

## ■ 自顶向下集成

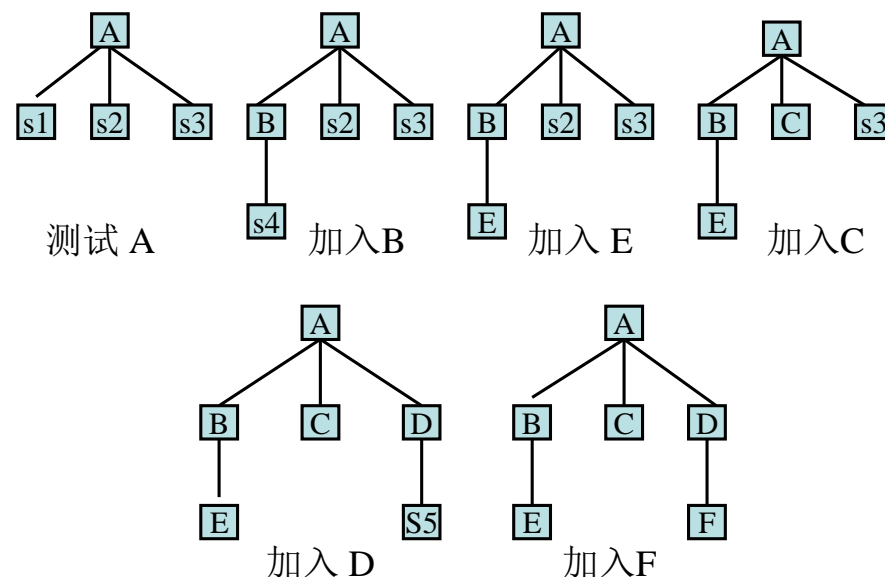
- 深度优先：A、B、E、C、D、F
- 广度优先：A、B、C、D、E、F



## ■ 广度优先的测试过程：

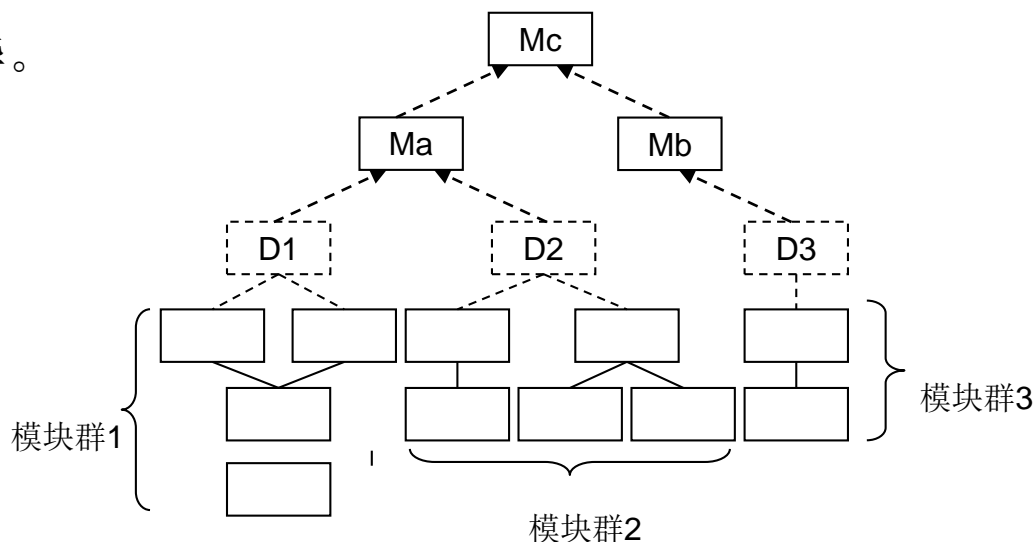
- Test A (with stubs for B,C,D)
- Test A;B (with stubs for E,C,D)
- Test A;B;C (with stubs for E,D)
- Test A;B;C;D (with stubs for E,F)
- Test A;B;C;D;E (with stubs for F)
- Test A;B;C;D;E;F

## 深度优先的测试过程



# 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：**
  - 把底层模块组织成实现某个子功能的模块群(cluster)；
  - 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；
  - 对每个模块群进行测试；
  - 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；
  - 循环，直到整个程序测试完毕。

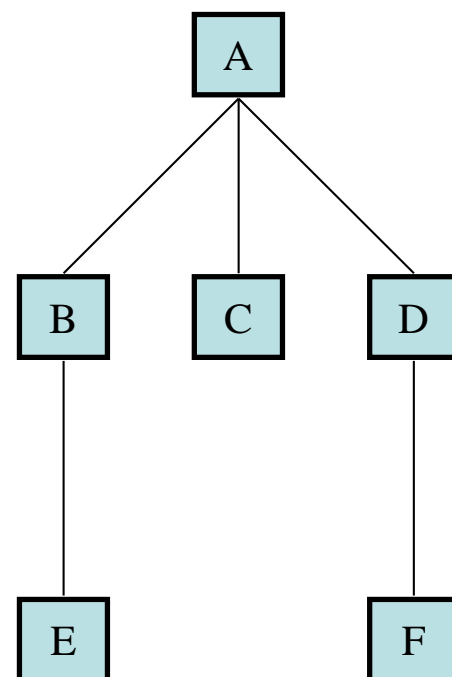




# 自底向上的增量集成

## ■ 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，因此较早地发现错误；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。

# 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。
- 步骤：
  - 确定以哪一层为界来决定使用三明治集成侧路额；
  - 对该层次及其下面的所有各层使用自底向上的集成策略；
  - 对该层次之上的所有各层使用自顶向下的集成策略；
  - 把该层次各模块同相应的下层集成；
  - 对系统进行整体测试。

# 三明治集成

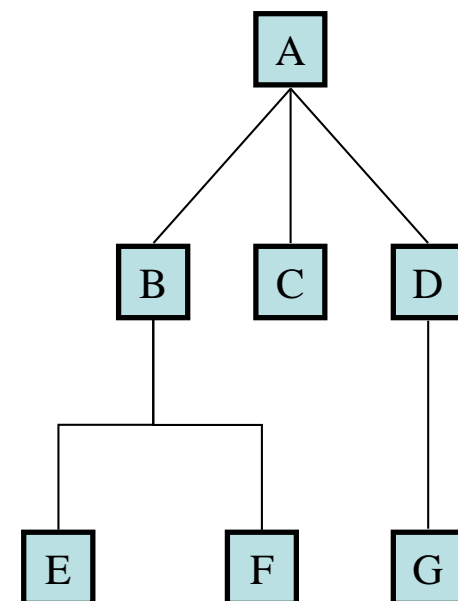
## ■ 选定模块B所在的中间层，过程如下：

- 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

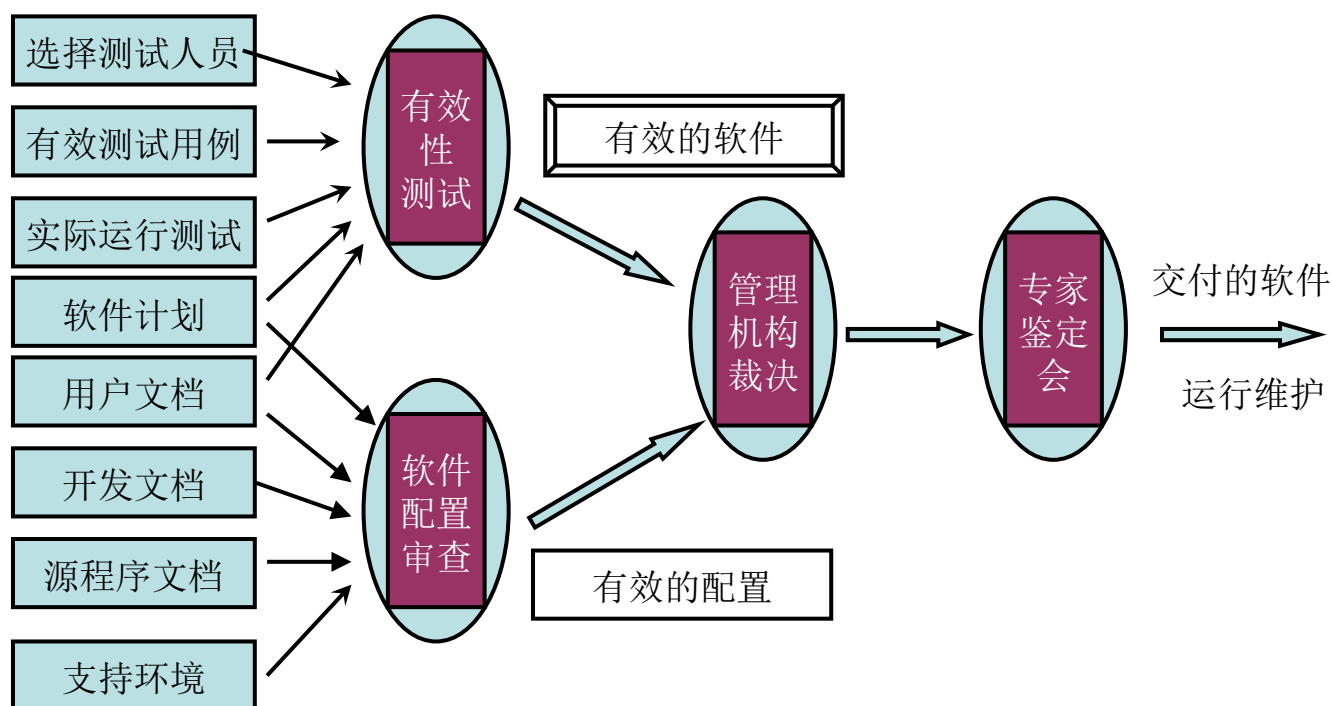
- Test E (with driver for B)
- Test F (with driver for B)
- Test B;E;F (with driver for A)
- Test G (with driver for D)
- Test D;G (with driver for A)

自底向上



### (3) 确认测试

- **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。

### ■ $\alpha$ 测试与 $\beta$ 测试

- $\alpha$ 测试与 $\beta$ 测试是产品在正式发布前经常进行的两种测试；
  - $\alpha$ 测试是由用户在开发环境下进行的测试；
  - $\beta$ 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。

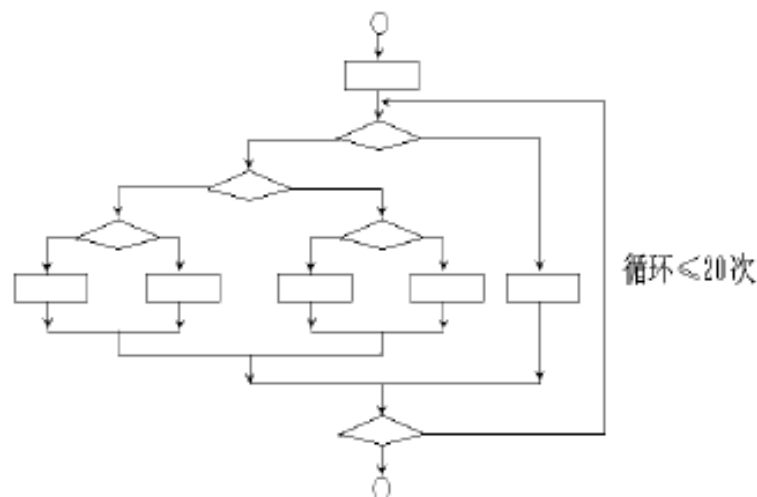
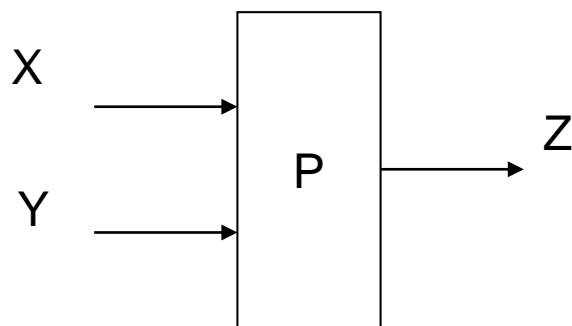
# 典型的软件测试技术

## ■ 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。

## ■ 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。





## 4 黑盒测试概述



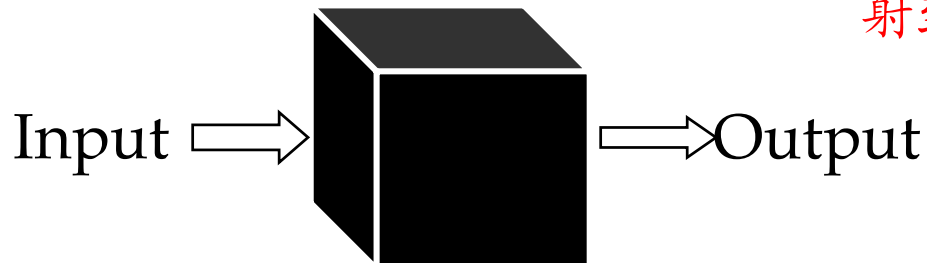


# 黑盒测试的定义

## ■ 黑盒测试(black-box testing):

- 又称“功能测试”、“数据驱动测试”或“基于规格说明书的测试”，是一种从用户观点出发的测试。
- 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。
- 通常在软件接口处进行。

原理：任何程序都可以看作是将输入定义域取值映射到输出值域的函数



# 黑盒测试能发现的错误

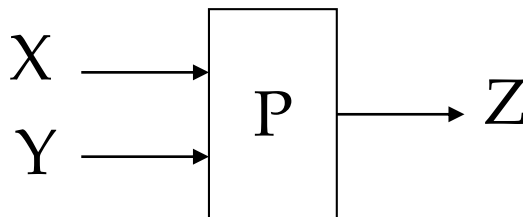
- 是否有不正确或遗漏的功能？
- 在接口上，输入能否被正确的接受？
- 能否输出正确的结果？
- 是否有数据结构错误或外部信息？
- 数据文件访问错误？
- 性能上是否能够满足要求？
- 是否有初始化或终止性错误？

## 黑盒测试中的“穷举”

- 用黑盒测试发现程序中的错误，必须在所有可能的输入条件和输出条件中确定测试数据，来检查程序是否都能产生正确的输出，但这是不可能的。

——因为穷举测试数量太大，无法完成。

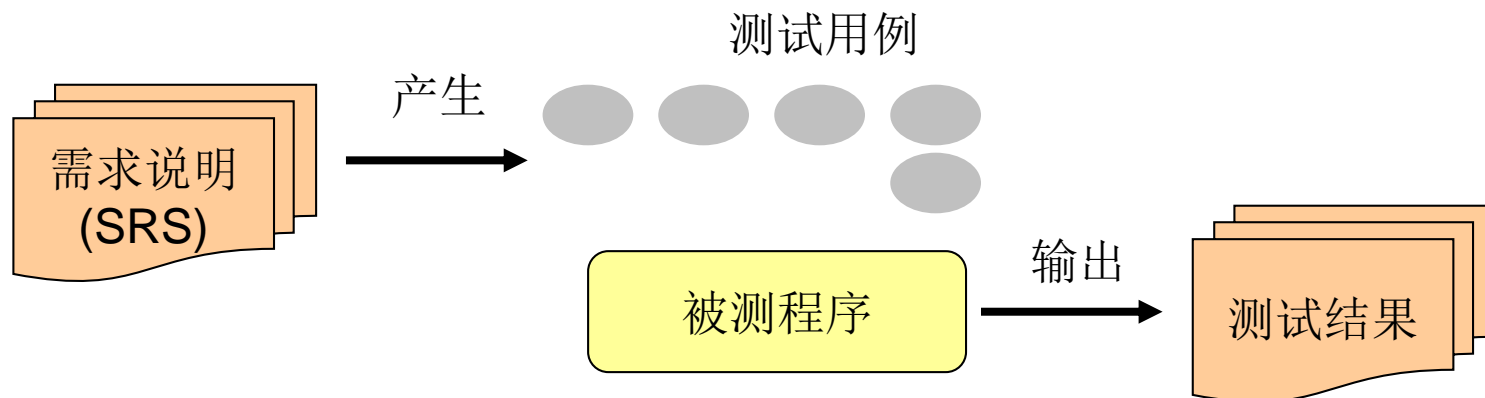
- 举例：程序P输入整数X和Y，输出Z，在字长为32位的计算机上运行。
  - 可能采用的测试数据组： $2^{32} \times 2^{32} = 2^{64}$
  - 如果测试一组数据需要1毫秒，一年工作 $365 \times 24$ 小时，完成所有测试需5亿年。



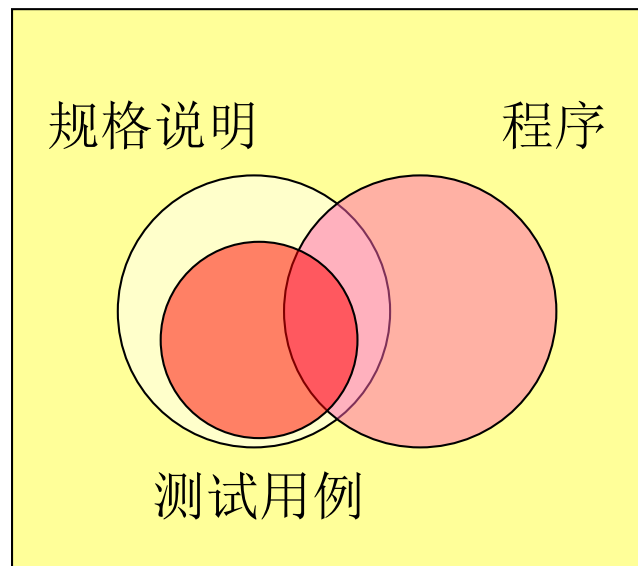
- 因此，测试人员只能在大量可能的数据中，选取其中一部分作为测试用例。

# 黑盒测试的实施过程

- 测试计划阶段
- 测试设计阶段
  - 依据程序需求规格说明书或用户手册，按照一定规范化的方法进行软件功能划分和设计测试用例。**输入数据和期望输出均从需求规格中导出。**
- 测试执行阶段
  - 按照设计的测试用例执行测试;
  - 自由测试(作为测试用例测试的补充)。
- 测试总结阶段

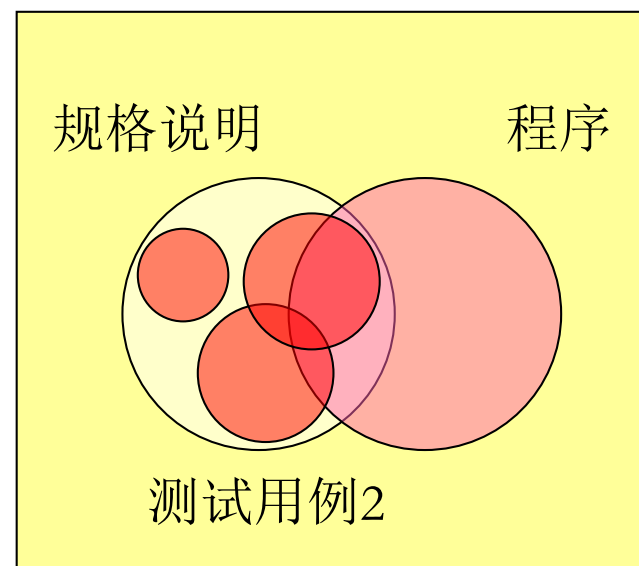
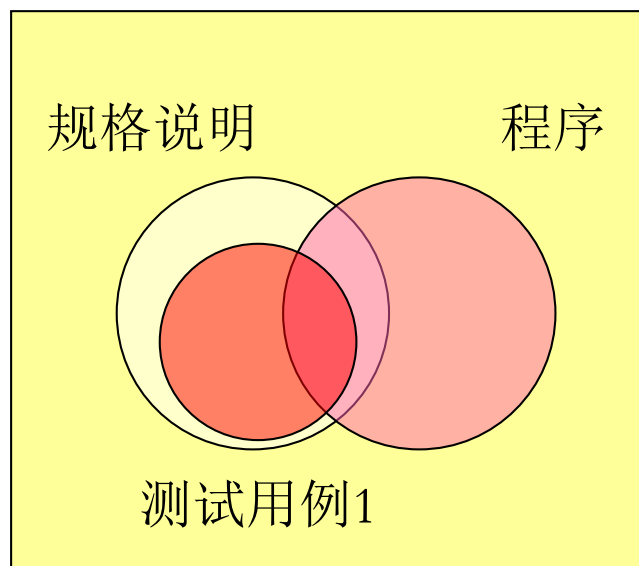


# 黑盒测试的Venn Diagram



注意：覆盖区域只能在规格说明部分

# 黑盒测试的Venn Diagram



测试用例所覆盖的规格说明范围越大，就越优良

设计良好的测试用例，使之尽可能完全覆盖软件的规格说明

# 测试用例的设计技术

- 等价类划分
- 边界值分析
- 错误推测法
- 因果图法
- 随机测试
- 决策树方法
- 判定表驱动分析方法
- 正交实验设计方法
- 功能图分析方法



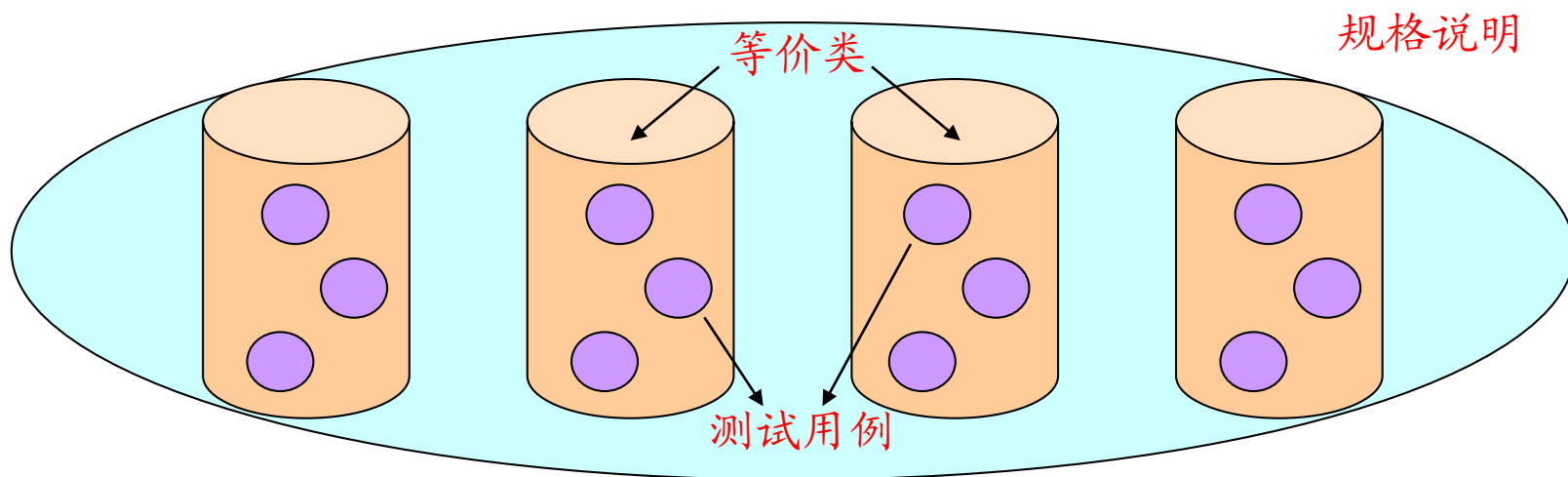
## 5 等价类划分法





# 等价类划分

- **等价类**：输入数据的某个子集，在该子集合中的各个输入数据对于揭露程序中的错误都是等效的，并合理地假定“**测试某等价类的代表值就等于对这一类其它值的测试**”。



- 在每一个等价类中选取少量有代表性的数据作为测试的输入条件，就可以用少量代表性的测试数据，并取得较好的测试结果。

# 等价类划分(Equivalence partitioning)

- 关键步骤：确定等价类和选择测试用例
- 基本原则：
  - 每个可能的输入属于某一个等价类
  - 任何输入都不会属于多个等价类
  - 用等价类的某个成员作为输入时，如果证明执行存在误差，那么用该类的任何其他成员作为输入，也能检查到同样的误差。

# 有效/无效等价类

## ■ 有效等价类

- 对于程序的规格说明来说是合理的、有意义的输入数据构成的集合。
- 利用有效等价类可检验程序是否实现了规格说明中所规定的功能和性能。

## ■ 无效等价类

- 对程序的规格说明是不合理的或无意义的输入数据所构成的集合。
- 无效等价类至少应有一个，也可能有多个。

## ■ 设计测试用例时，要同时考虑这两种等价类。

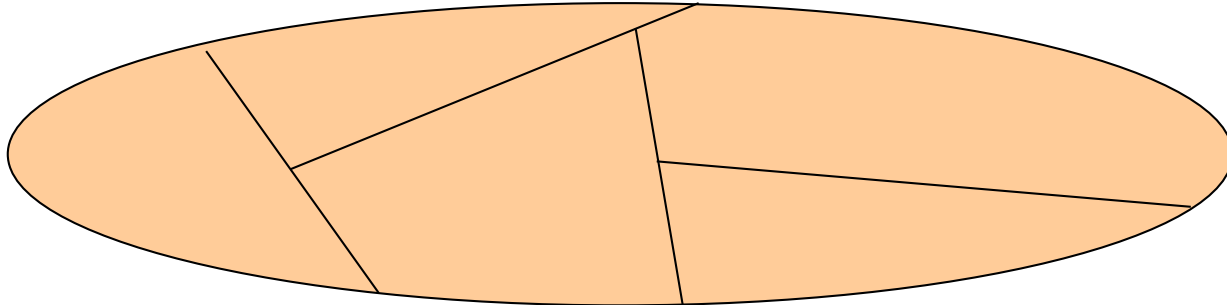
- 软件不仅要能接收合理的数据，也要能经受意外的考验，这样的测试才能确保软件具有更高的可靠性。

# 划分等价类的标准

- 划分等价类的标准：“完备测试、避免冗余”
- 将输入数据的集合(P)划分为一组子集( $E_1, E_2, \dots, E_n$ ), 并尽可能满足:

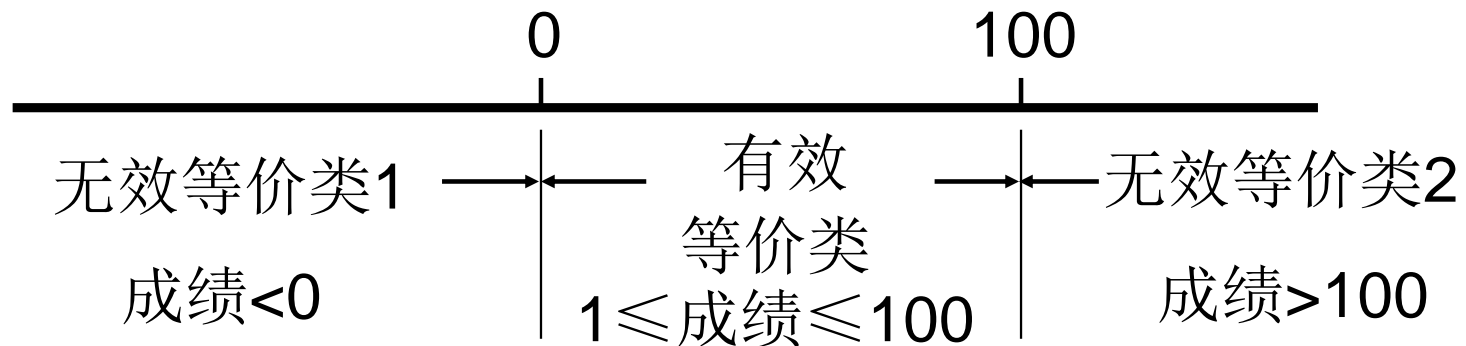
$$E_1 \cup E_2 \cup \dots \cup E_n = P$$

$$E_i \cap E_j = \emptyset$$



## 确定等价类的六大原则

- 原则1: 在输入条件规定了取值范围或值的个数的情况下, 则可以确立一个有效等价类和两个无效等价类。
- 例如: 输入值是学生成绩, 范围是0~100



## 确定等价类的六大原则

- 原则2: 在输入条件规定了输入值的集合或者规定了“必须如何”的条件的前提下,可确立一个有效等价类和一个无效等价类。
- 原则3: 在输入条件是一个布尔量的情况下,可确定一个有效等价类和一个无效等价类。
- 原则4: 在规定了输入数据的一组值(假定 $n$ 个)、并且程序要对每一个输入值分别处理的情况下,可确立 $n$ 个有效等价类和一个无效等价类。
  - 例如,输入条件说明学历可为{专科, 本科, 硕士, 博士}四种之一,则分别取这四种这四个值作为四个有效等价类,另外把四种学历之外的任何学历作为无效等价类。

# 确定等价类的六大原则

- 原则5: 在规定了输入数据必须遵守的规则的情况下, 可确立一个有效等价类(符合规则)和若干个无效等价类(从不同角度违反规则)。
- 原则6: 在确知已划分的等价类中各元素在程序处理中的方式不同的情况下, 则应再将该等价类进一步的划分为更小的等价类。

# 设计测试用例

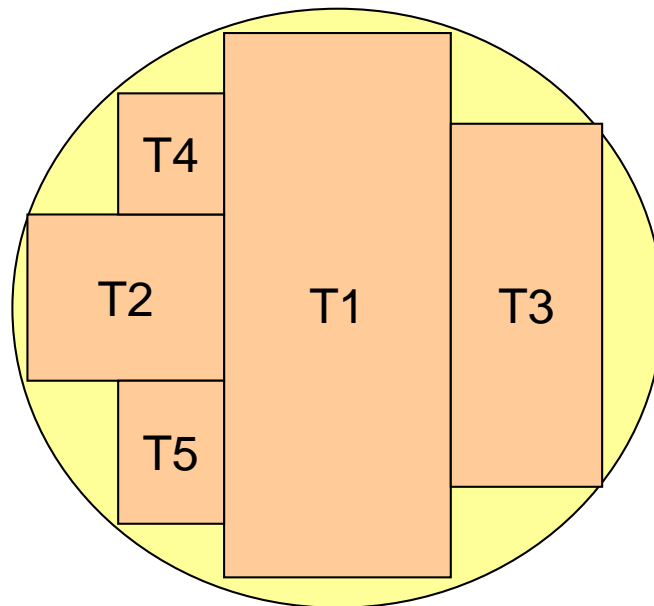
- 测试用例 = {测试数据+期望结果}
- 测试结果 = {测试数据+期望结果+实际结果}
- 在确立了等价类后，可建立等价类表，列出所有划分出的等价类输入数据+期望结果：

输入条件	有效等价类	无效等价类
...	...	...
...	...	...



# 设计测试用例

- 为每一个等价类规定一个唯一的编号
- 设计一个新的测试用例，使其尽可能多的覆盖尚未被覆盖的有效等价类；重复这一步，直到所有的有效等价类都被覆盖为止
- 设计一个新的测试用例，使其仅覆盖一个尚未被覆盖的无效等价类；重复这一步，直到所有的无效等价类都被覆盖为止



## 例1：数据录入问题

- [例1]某一程序要求输入数据满足以下条件：
  - 可输入1个或多个数据，每个数据由1-8个字母或数字构成，且第一个字符必须为字母；
  - 如果满足上述条件，则输出“合法数据”；否则，输出“非法数据”；
- 用等价类划分方法为该程序进行测试用例设计。

## 例1：输入变量问题

### ■ 划分等价类：

输入条件	有效等价类	号码	无效等价类	号码
标识符个数	1个	(1)	0个	(6)
	多个	(2)		
标识符字符数	1~8个	(3)	0个	(7)
			>8个	(8)
标识符组成	数字与字母	(4)	含有非“字母或数字”	(9)
第一个标识符	字母	(5)	非字母	(10)

## 例1：输入变量问题

- 设计测试用例：

测试用例编号	测试用例内容	覆盖的等价类
1	a2ku83t	(1)(3)(4)(5)
2	a2ku83t, fta, b2	(2)(3)(4)(5)
3		(6)
4	a2ku83t, , b2	(7)
5	a2ku83t29, fta	(8)
6	a2ku8\$t	(9)
7	92ku83t	(10)

## 例2：日期检查

- [例2] 档案管理系统，要求用户输入以年月表示的日期。假设日期限定在1990年1月~2049年12月，并规定日期由6位数字字符组成，前4位表示年，后2位表示月。
- 用等价类划分法设计测试用例，来测试程序的“日期检查功能”。

## 例2：日期检查

### ■ 划分等价类：

输入等价类	有效等价类	无效等价类
日期的类型及长度	①6位数字字符	②有非数字字符 ③少于6位数字字符 ④多于6位数字字符
年份范围	⑤在1990~2049之间	⑥小于1990 ⑦大于2049
月份范围	⑧在01~12之间	⑨等于00 ⑩大于12

## 例2：日期检查

- 设计有效等价类的测试用例：

测试用例编号	测试用例内容	覆盖的等价类
1	200711	(1)(5)(8)

- 设计无效等价类的测试用例：

测试用例编号	测试用例内容	覆盖的等价类
1	07June	(2)
2	20076	(3)
3	2007011	(4)
4	198912	(6)
5	205401	(7)
6	200700	(8)
7	200713	(10)

## 课堂讨论：黑盒测试案例

- 南非世界杯足球赛的门票中含有电子信息，验票模块可根据该信息自动检查是否是有效门票。若该电子信息的格式如下：
  - 第1-4位是表示比赛日期的数字，第1-2位表示月(只能为06或07)，第3-4位表示日(范围为01-30)，但月和日组合起来只能在6月10日和7月11日之间。若违反该条规则，则输出“非法日期”；
  - 第5-7位表示比赛地点，每一位都是大写字母；若违反，则输出“非法地点”；
  - 第8-12位是表示球场座位的数字，其范围是[00001, 99999]；若违反，则输出“非法座位”；
  - 若满足上述所有条件，则输出“合法门票”。
- 针对该验票模块，请使用等价类划分方法识别该模块的有效等价类和无效等价类；
- 针对等价类识别的结果，为该模块设计一组黑盒测试用例(包括输入数据和期望结果)。





## 6 边界值分析法



# 边界值分析

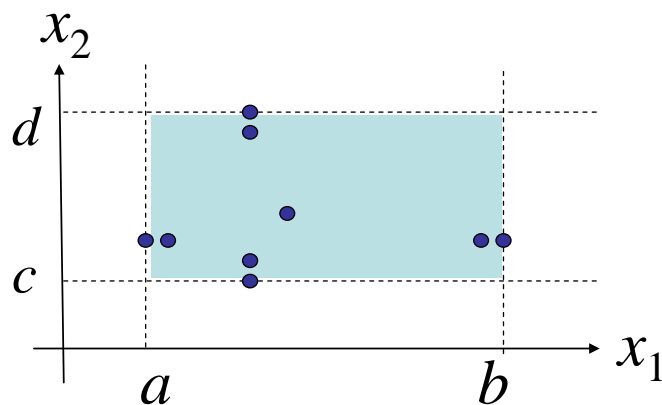
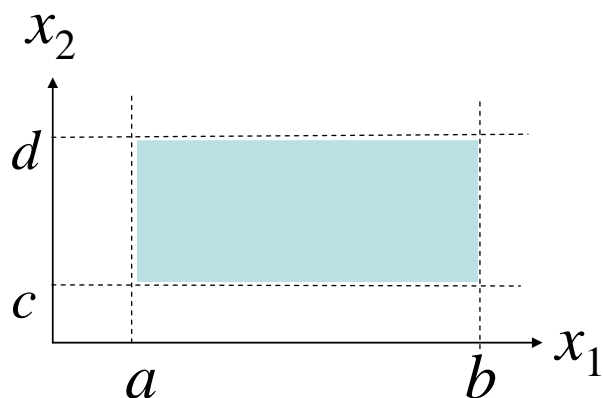
- 边界值分析是等价类测试的特例，主要是考虑等价类的边界条件，在等价类的“边缘”选择元素。
- 长期的测试经验表明：大量的错误是发生在输入或输出范围的边界上，而不是发生在输入输出范围的内部。
- 因此针对各种边界情况设计测试用例，可以查出更多的错误。

# 确定边界

- 使用边界值分析方法设计测试用例，首先应确定边界情况：
  - 通常输入和输出等价类的边界，就是应着重测试的边界情况。
  - 选取正好等于、刚刚大于、刚刚小于边界的值作为测试数据，而不是选取等价类中的典型值或任意值作为测试数据。
- 例：在 $[R_1, R_2]$  的取值区间中，应如何选择？

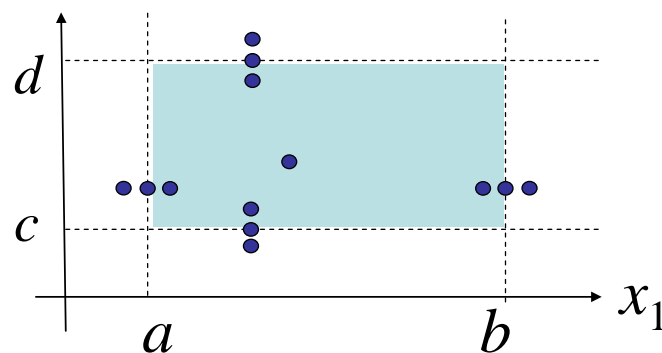


# 根据边界确定测试用例



5个测试用例：  
 最小值；略高于最小值；  
 正常值；  
 略低于最大值；最大值；

$n$ 个变量，有 $4n+1$ 个测试用例



在5个测试用例的基础上增如：  
 略高于最大值；  
 略低于最小值；

# 常见的边界值

- 通常情况下，软件测试所包含的边界检验有几种类型：数字、字符、位置、重量、大小、速度、方位、尺寸、空间等。
- 相应地，以上类型的边界值应该在：最大/最小、首位/末位、上/下、最快/最慢、最高/最低、最短/最长、空/满等情况下。
  - 对16-bit 的整数而言，32767和-32768是边界
  - 屏幕上光标在最左上、最右下位置
  - 报表的第一行和最后一行
  - 数组元素的第一个和最后一个
  - 循环的第0次、第1次和倒数第2次、最后1次

# 边界值分析的原则

- **原则1:** 如果输入条件规定了值的范围，则应取刚达到这个范围的边界的值，以及刚刚超越这个范围边界的值作为测试输入数据。
- 例如：如果程序的规格说明中规定“重量在10公斤至50公斤范围内的邮件，其邮费计算公式为……”。
  - 作为测试用例，应取10、50、10.01、49.99、9.99及50.01等

## 边界值分析的原则

- 原则2: 如果输入条件规定了值的个数, 则用最大个数、最小个数、比最小个数少1, 比最大个数多1的数据作为测试数据。
- 比如, 一个输入文件应包括1~255个记录, 则测试用例可取1和255, 还应取0及256等。

# 边界值分析的原则

- **原则3: 将原则1和原则2应用于输出条件, 即设计测试用例使输出值达到边界值及其左右的值。**
- 例如, 某程序的规格说明要求计算出“每月保险金扣除额为0至1165.25元”, 其测试用例可取0.00及1165.24、还可取-0.01及1165.26等。
- 再如, 某程序要求每次“最少显示1条、最多显示4条查询结果”, 这时应考虑测试用例包括1和4, 还应包括0和5等。



# 边界值分析的原则

- 原则4: 如果程序的规格说明给出的输入域或输出域是有序集合, 则应选取集合的第一个元素和最后一个元素作为测试用例。
- 原则5: 如果程序中使用了一个内部数据结构, 则应当选择这个内部数据结构的边界上的值作为测试用例。
- 原则6: 分析规格说明, 找出其它可能的边界条件。

## [例1] 三角形

- 假定三角形每边边长的取范围值设值为[1, 100]

测试用例	a	b	c	预期输出
Test1	60	60	1	等腰三角形
Test2	60	60	2	等腰三角形
Test3	60	60	60	等边三角形
Test4	50	50	99	等腰三角形
Test5	50	50	100	非三角形
Test6	60	1	60	等腰三角形
Test7	60	2	60	等腰三角形
Test8	50	99	50	等腰三角形
Test9	50	100	50	非三角形
Test10	1	60	60	等腰三角形
Test11	2	60	60	等腰三角形
Test12	99	50	50	等腰三角形
Test13	100	50	50	非三角形

## [例2] NextDate()

### ■ 在NextDate()函数中:

- $1 \leq \text{month} \leq 12$
- $1 \leq \text{day} \leq 31$
- $1912 \leq \text{year} \leq 2050$

测试用例	month	day	year	预期输出
Test1	6	15	1911	1911.6.16
Test2	6	15	1912	1912.6.16
Test3	6	15	1913	1913.6.16
Test4	6	15	1975	1975.6.16
Test5	6	15	2049	2049.6.16
Test6	6	15	2050	2050.6.16
Test7	6	15	2051	2051.6.16
Test8	6	-1	2001	day超出[1...31]
Test9	6	1	2001	2001.6.2
Test10	6	2	2001	2001.6.3
Test11	6	30	2001	2001.7.1
Test12	6	31	2001	输入日期超界
Test13	6	32	2001	day超出[1...31]
Test14	-1	15	2001	Mouth超出[1...12]
Test15	1	15	2001	2001.1.16
Test16	2	15	2001	2001.2.16
Test17	11	15	2001	2001.11.16
Test18	12	15	2001	2001.12.16
Test19	13	15	2001	Mouth超出[1...12]

## [例2] NextDate()

- 另一种更详尽的划分方法

- $D1 = \{ 1 \leq \text{date} < \text{last day of the month} \}$
- $D2 = \{ \text{last day of the month} \}$
- $D3 = \{ \text{Dec. 31} \}$
- $M1 = \{ 30\text{-day months} \}$
- $M2 = \{ 31\text{-day months} \}$
- $M3 = \{ \text{Feb.} \}$
- $Y1 = \{ 2000 \}$
- $Y2 = \{ \text{leap year} \}$
- $Y3 = \{ \text{not leap year} \}$

# [例2] NextDate()

序号	Month	Day	Year	期望输出
1	2	13	6	25
2	2	14	6	26
3	2	15	6	27
4	2	16	6	28
5	2	17	6	29
6	2	18	6	30
7	2	19	6	31
8	2	20	6	1
9	2	21	6	2
10	2	22	6	3
11	2	23	6	4
12	2	24	6	5
13	6	25	8	26
14	6	26	8	27
15	6	27	8	28
16	6	28	8	29
17	6	29	8	30
18	6	30	8	31
19	6	31	8	1
20	6	32	8	2
21	6	33	8	3
22	6	34	8	4
23	6	35	8	5
24	6	36	8	6
25	8	1	2000	2001-1-1
26	8	1	1996	1997-1-1
27	8	1	2002	2003-1-1
28	8	2	2000	无效的输入日期
29	8	2	1996	无效的输入日期
30	8	2	2002	无效的输入日期
31	8	3	2000	无效的输入日期
32	8	3	1996	无效的输入日期
33	8	3	2002	无效的输入日期
34	8	3	0	无效的输入日期
35	8	3	0	无效的输入日期
36	8	3	0	无效的输入日期
37	12	31	2000	2001-1-1
38	12	31	1996	1997-1-1
39	12	31	2002	2003-1-1
40	13	14	2000	无效的输入日期
41	13	30	1996	无效的输入日期
42	13	31	2002	无效的输入日期
43	3	0	2000	无效的输入日期
44	3	32	1996	无效的输入日期
45	9	14	0	无效的输入日期
46	0	0	0	无效的输入日期
47	-1	14	2000	无效的输入日期
48	11	-1	2002	无效的输入日期



## 7 黑盒测试 vs 白盒测试



# 课堂讨论

- 作为两种常用的测试方法，黑盒测试和白盒测试有何异同？
  - 应用场合
  - 发现错误的类型
  - 发现错误的能力
  - 测试效率
  - 测试用例设计的难度
  - 测试成本
  - 测试的时间点
  - 综合起来，各自的优点和缺点？

## 二者的对比

### ■ 有了黑盒测试为何还需要白盒测试？

- 黑盒测试只能观察软件的外部表现，即使软件的输入输出都是正确的，却并不能说明软件就是正确的。因为程序有可能用错误的运算方式得出正确的结果，例如“负负得正，错错得对”，只有白盒测试才能发现真正的原因。
- 白盒测试能发现程序里的隐患，例如内存泄漏、误差累计问题。在这方面，黑盒测试存在严重的不足。

### ■ 有了白盒测试为何还需要黑盒测试？

- 通过了白盒测试只能说明程序代码符合设计需求，并不能说明程序的功能符合用户的需求。如果程序的系统设计偏离了用户需求，即使100%正确编码的程序也不是用户所要的。



## 黑盒测试的优点

- 对于较大的代码单元来说(子系统甚至系统级)，黑盒测试比白盒测试效率要高；
- 测试人员不需要了解实现的细节，包括特定的编程语言；
- 测试人员和编码人员是彼此独立的；
- 从用户的视角进行测试，很容易被理解和接受；
- 有助于暴露任何规格不一致或有歧义的问题；
- 测试用例可以在规格完成之后马上进行。

## 黑盒测试的缺点

- 只有一小部分可能的输入被测试到，要测试每个可能的输入流几乎是不可能的；
- 没有清晰的和简明的规格，测试用例是很难设计的；
- 如果测试人员不被告知开发人员已经执行过的用例，在测试数据上会存在不必要的重复；
- 会有很多程序路径没有被测试到；
- 不能直接针对特定程序段测试，这些程序段可能很复杂(因此可能隐藏更多的问题)；
- 大部分和研究相关的测试都是直接针对白盒测试的。

# 白盒测试的优缺点

## ■ 优点

- 迫使测试人员去仔细思考软件的实现;
- 可以检测代码中的每条分支和路径;
- 揭示隐藏在代码中的错误;
- 对代码的测试比较彻底;
- 最优化。

## ■ 缺点

- 昂贵;
- 无法检测代码中遗漏的路径和数据敏感性错误;
- 不验证规格的正确性。

# 黑盒测试 vs. 白盒测试

黑盒(功能)测试	白盒(结构)测试
只利用规格说明标识测试用例	只利用程序源代码标识测试用例
如果程序实现了未描述的行为，功能测试无法意识到。	如果已描述的行为未能实现，结构性测试无法意识到。
冗余度大，可能会有漏洞	具有覆盖率指标

## 二者的对比

- 白盒测试只考虑测试软件产品，它不保证完整的需求规格是否被满足。
- 黑盒测试只考虑测试需求规格，它不保证实现的所有部分是否被测试到。
- 黑盒测试会发现遗漏的缺陷，指出规格的哪些部分没有被完成。而白盒测试会发现代理方面缺陷，指出哪些实现部分是错误的。
- 白盒测试比黑盒测试成本要高得多。它需要在测试可被计划前产生源代码，并且在确定合适的数据和决定软件是否正确方面需要花费更多的工作量。
- 一个白盒测试的失败会导致一次修改，这需要所有的黑盒测试被重复执行并且重新决定白盒测试路径。

# 课堂讨论

- 通过查阅资料，了解测试驱动的开发（TDD）；
- 简要阐述TDD与传统的“编码+测试”相比有何优势和不足？
- 为什么目前工业界和学术界的一部分牛人仍对TDD持有较大的怀疑态度？



結束

2017年10月22日