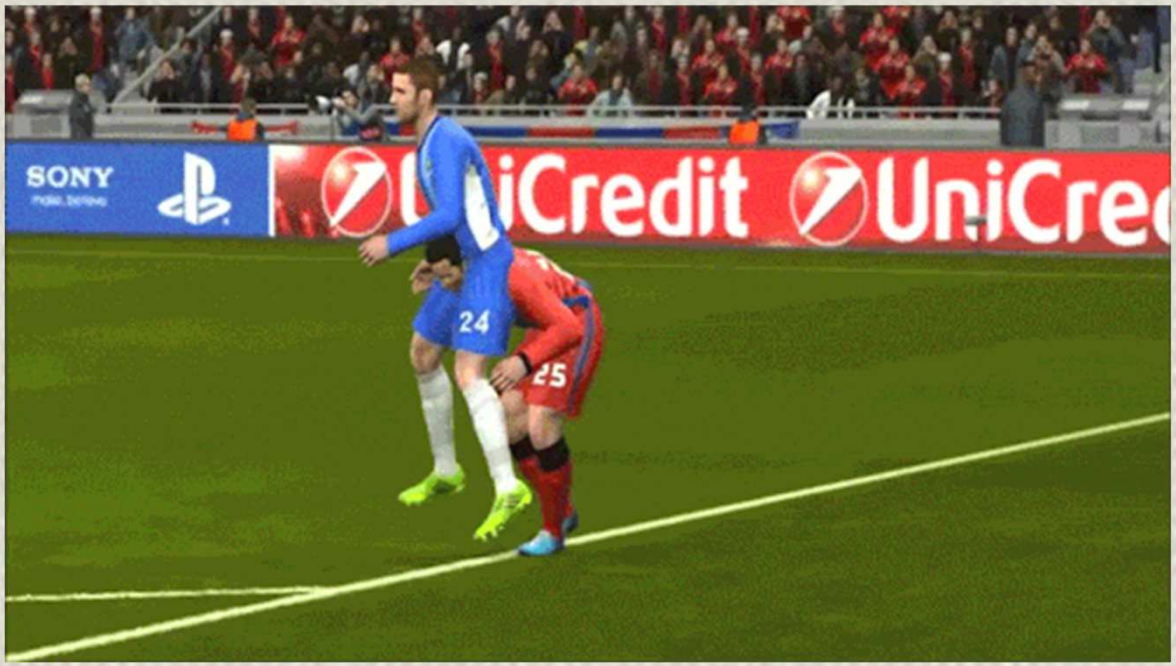
Ch.6 Testing & QA

2018.6

§ 4.0 BUGs

Funny Bugs

Funny Bugs



Costly Bugs

Ariane5
3.7亿美金的大烟花
copy了上一代的代码

Costly Bugs

聪明人留下的错

> 5000 亿!，$

Fatal Bug

宰赫兰反导系统拦截侯赛因飞毛腿导弹失败
0.33s
空速达4.2马赫（每秒1.5公里）
炸死28个美国士兵，炸伤100多人

目录

主要概念

- Software Testing
- Quality Assurance

目标

- 找到bug → 检查是否合格的找茬游戏
- 找到bug并且将严重的bug fix掉 → 确保软件“每个角度”都满足要求

Synonym

Defect--缺陷　　**Variance--偏差**

Fault--故障　　**Failure--失败**

Problem--问题　**Inconsistency--不一致**

Error--错误　　**Feature--特性**

Anomaly--异常（反常）

Exception--异常（不正常）

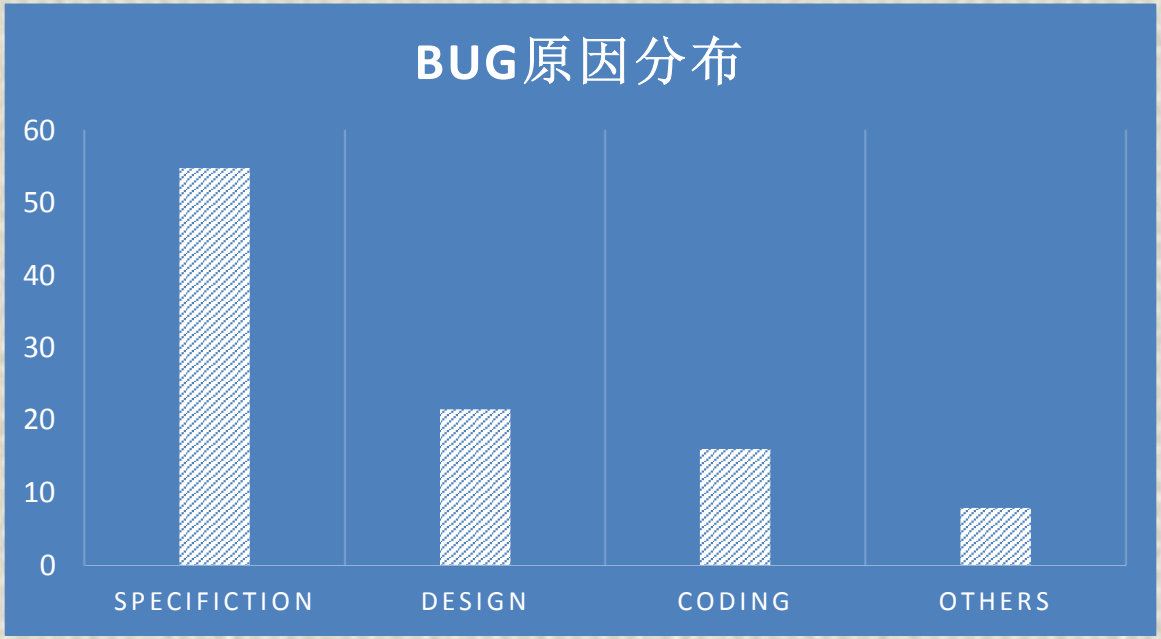
测试一个水杯

也是水杯

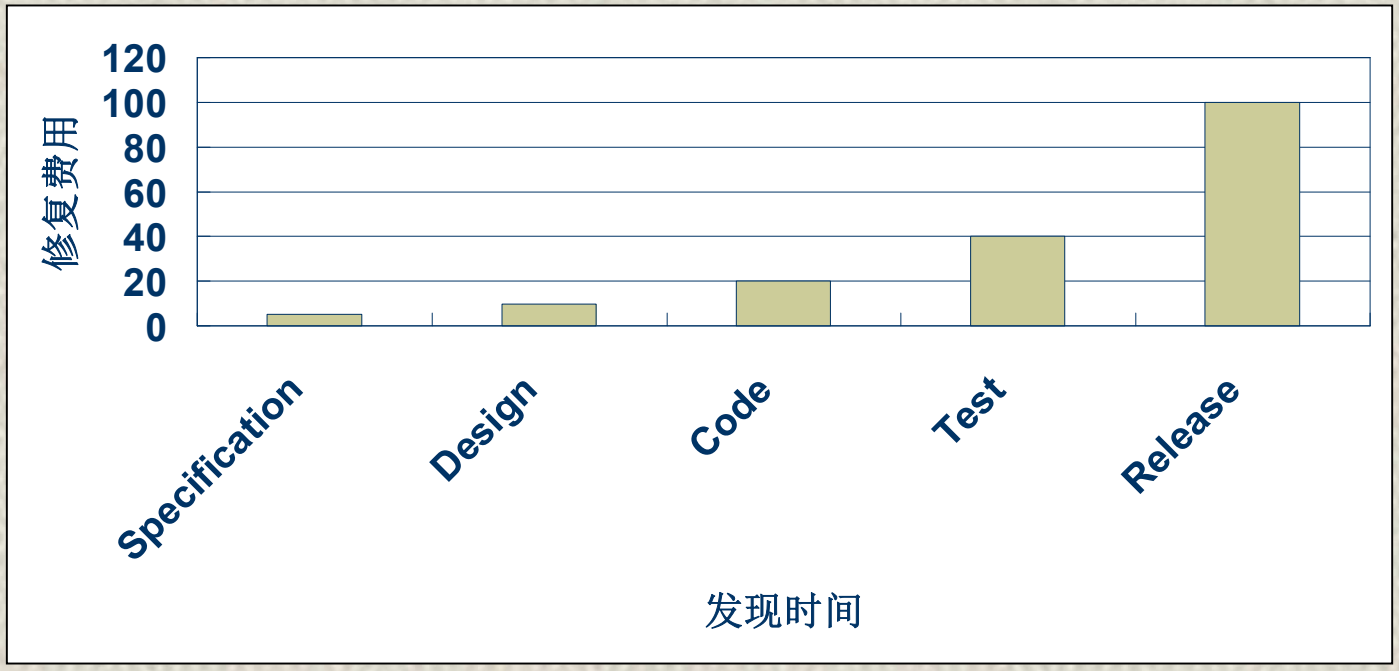
怎么定义软件缺陷？

- 软件未达到产品说明书标明的功能
- 软件出现了产品说明书指明不会出现的错误
- 软件功能超出产品说明书指明范围
- 软件未达到产品说明书虽未指出但应达到的目标
- 软件测试员认为软件难以理解、不易使用、运行速度缓慢、或者最终用户认为不好

BUG怎么产生的？


BUG原因分布
60
50
40
30
20
10
0
SPECIFICTION
DESIGN
CODING
OTHERS

修复BUG的成本


120
100
80
60
40
20
0
修复费用
Specification
Design
Code
Test
Release
发现时间

说出来你可能不信

- **QA和测试的工作从项目一开始就展开了**
  - 文档审核
  - 资源审查
  - 流程检测

- **100%的充分测试是不可能的**
  - 工程量太大（太多的输入/输出）
  - 测试时间有限
  - 软件说明书没有客观标准
  - 总有些信息是无法获取的

- 测试本身具有风险
  - 由于错误地估计BUG的严重性，导致成本增加
  - 过分依赖于测试团队，导致工程质量下降
  - 团队更大，更容易泄漏核心技术

- 有些BUG永远都不能修复
  - 没时间修复，工期太紧张！
  - 没法修复，就是找不出什么原因！
  - 功能太复杂，修复过程和重做没差别！
  - 牵一发动全身，解决一个，带来一百个新的！
  - 不值得修复

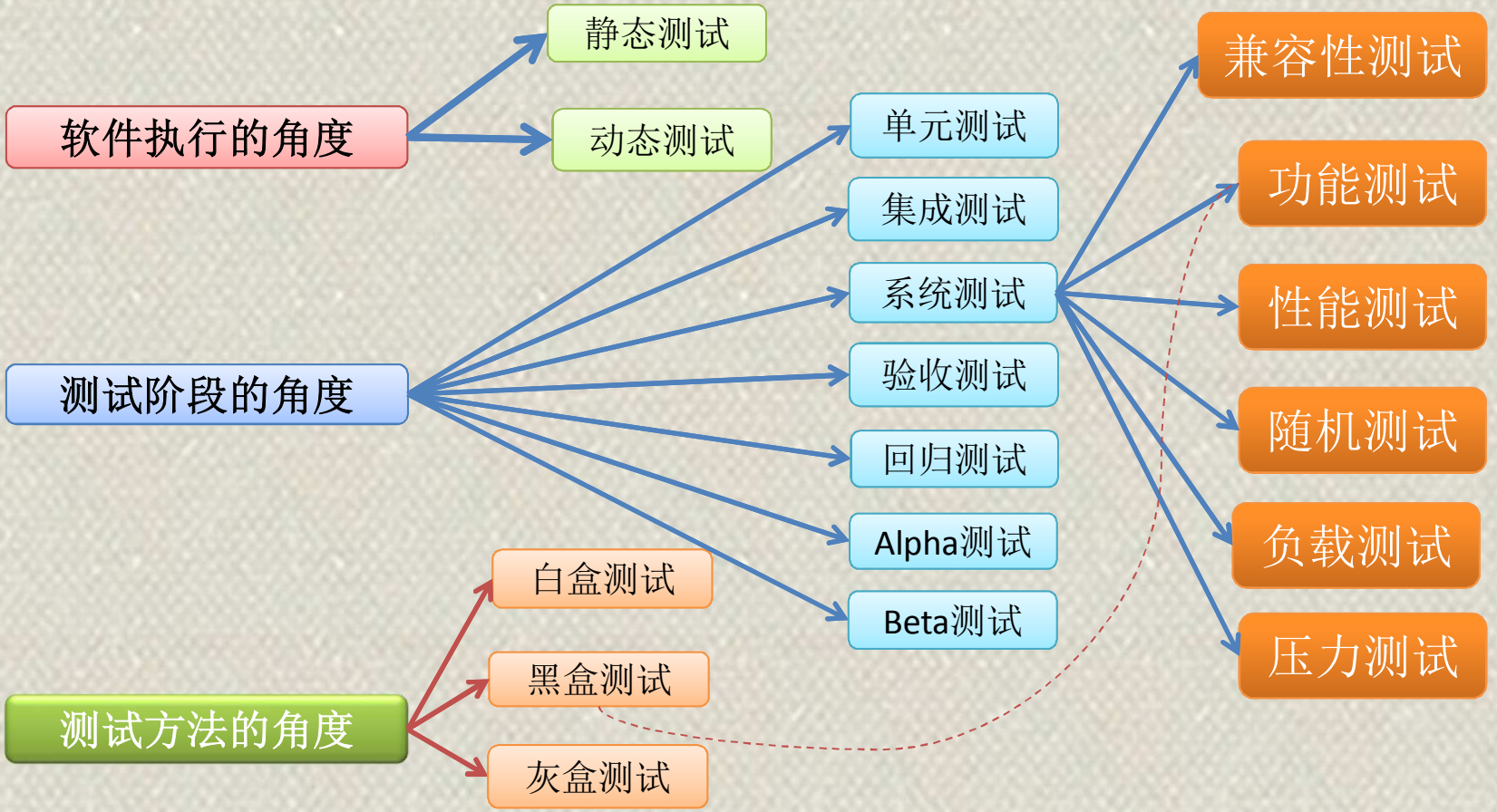
- 找到的**BUG**越多，潜在的**BUG**就越多
- 只能找到一些**BUG**，不可能找到全部

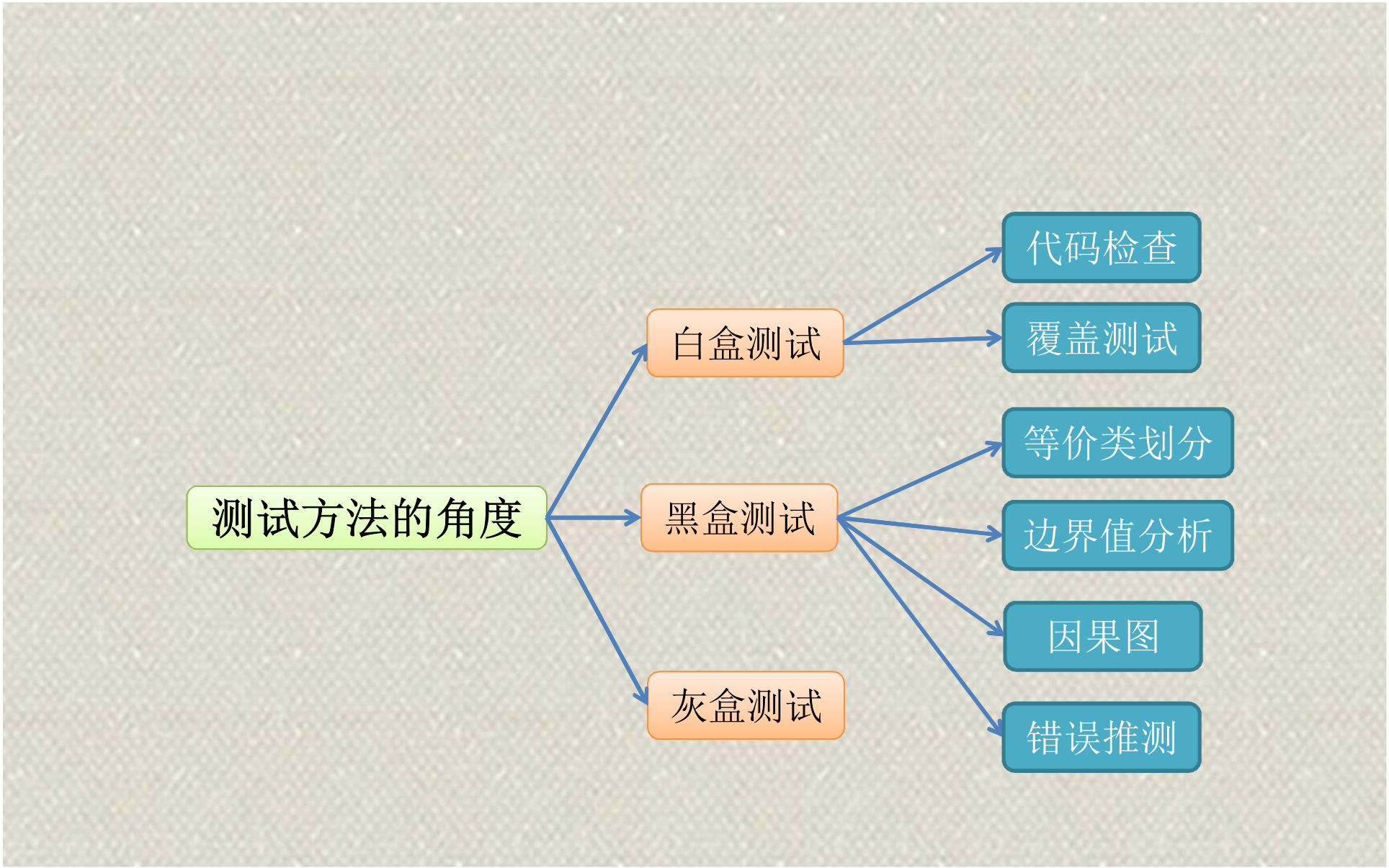
那怎么办

- 找到方法进行“有效”地测试
- 发现问题不断完善

目录

大体的分类方法


软件执行的角度
静态测试
动态测试
测试阶段的角度
单元测试
集成测试
系统测试
验收测试
回归测试
Alpha测试
Beta测试
测试方法的角度
白盒测试
黑盒测试
灰盒测试
兼容性测试
功能测试
性能测试
随机测试
负载测试
压力测试


测试方法的角度
白盒测试
黑盒测试
灰盒测试
代码检查
覆盖测试
等价类划分
边界值分析
因果图
错误推测

Static & dynamic

- 静态测试：不运行程序, 核对文档/代码
- 动态测试：执行程序，对“执行过程”和“执行结果”进行分析

Load, Stress & Performance

- 测试产品在不同负载下的表现
- 在各种极限情况下对产品进行测试
- 0.1秒计算出结果/20秒计算出结果

单元测试

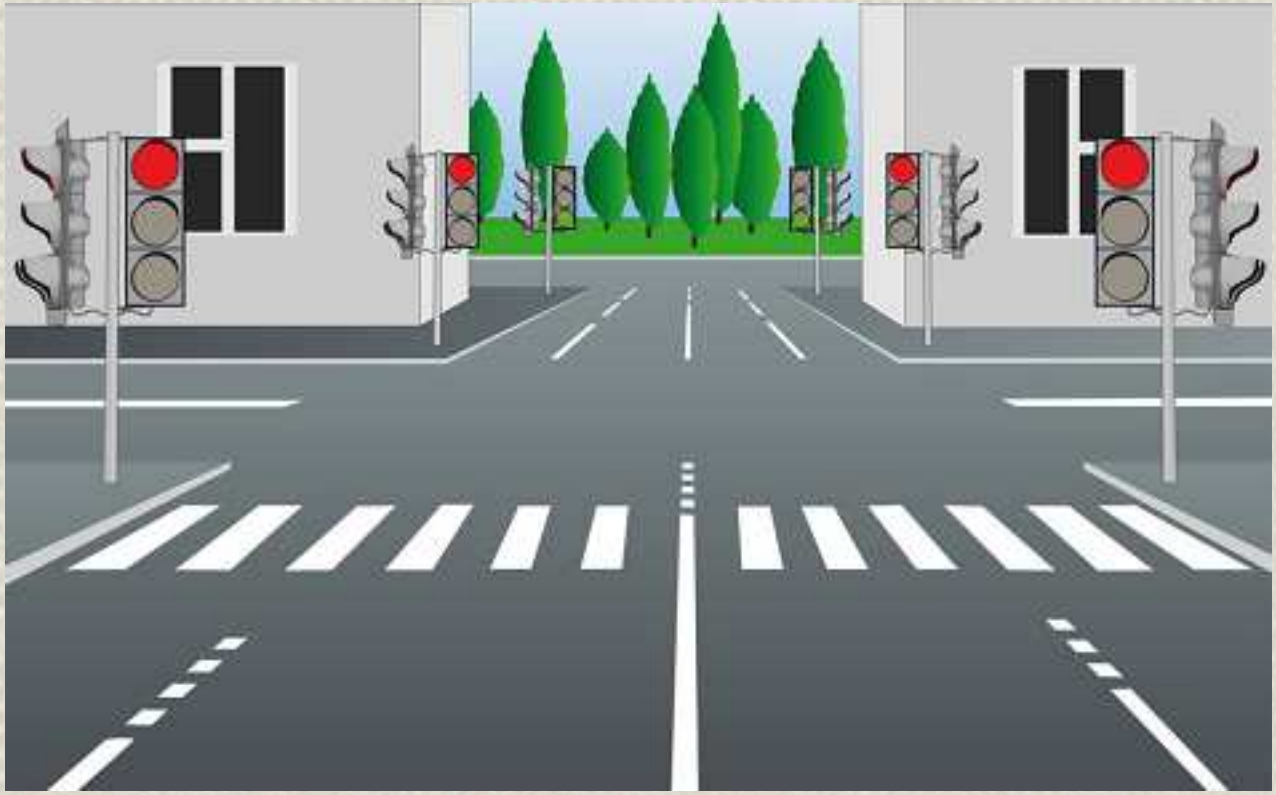
- 测试每个模块→子系统的独立的功能
- 部分有问题，整体是否有问题？

单元测试

集成测试

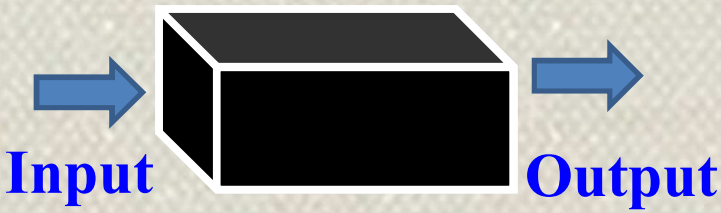
- 把模块集成到一起看，测试整体的功能
- 单个模块测试有效，集成后，这个模块会不出问题？

集成测试

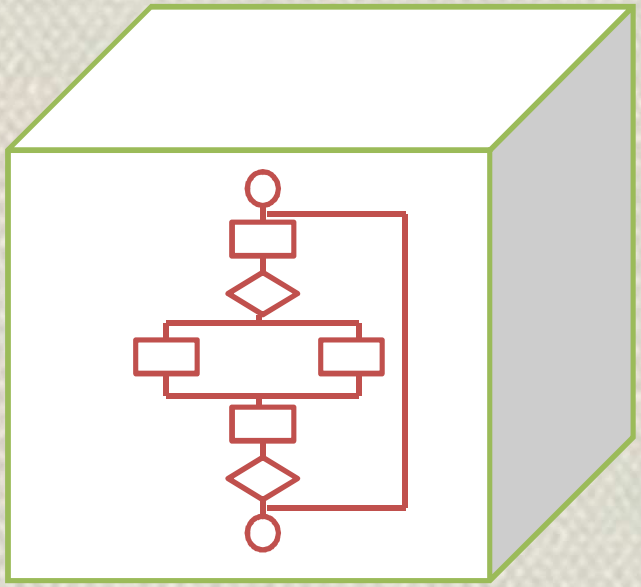


集成的复杂性

Black Box



White Box



```
int fac(int n)
{
        if (0 == n)  return 1;

        return  n * fac(n - 1);
}
```

覆盖性测试

```
String howold(int age)
{
    if(age < 2)
       return "Baby";
    if(age < 14)
       return "Child";
    if(age > 18)
       return "Adult";
    else
       return "";
}
```

- age = 1
- age = 2?
- age = 13
- age = 14?
- age = 16 ?

边界值检测

```
short doubleme(short n)
{
   return (short)(n * 2);
}
```

- n=0;
- n = 32767 ?
- n = 16384 ?
- n = -16384;
- n = -16385;

目录

测试工作的专业性

- 要懂编程和必要的计算机知识
  - If/else/switch的覆盖性测试
  - 边界值分析
  - 数值会不会溢出
  - 硬盘为什么是性能的瓶颈

测试工作的专业性

- 要懂特定领域软件相关的专门知识
  - 驱动软件和一般的软件有什么差别
  - 电商网站的订单流程
  - 流媒体的节目搜索
  - 短视频推送的基本原理

测试工作的专业性

- 能有效地运用分析方法
  - 思路清晰，避免重复无效的工作
  - 正确的方法，增加有效的工作量

测试工作的专业性

- 能设计专业的测试用例
  - 代表性强
  - 覆盖性广
  - 明确无歧义

测试工作的专业性

- 会使用增加效率的工具
  - 可以完成人工无法完成的任务
  - 大量重复的工作让机器去做
  - 工具是方法的实现

测试工作的专业性

- 做具有说服力的报告总结
  - 软件测试评价
  - 软件测试报告
  - 数据最具说服力

目录

测试用例

- 对一个测试点进行测试的全部过程描述
  - 工作环境
  - 输入
  - 操作过程
  - 预期结果

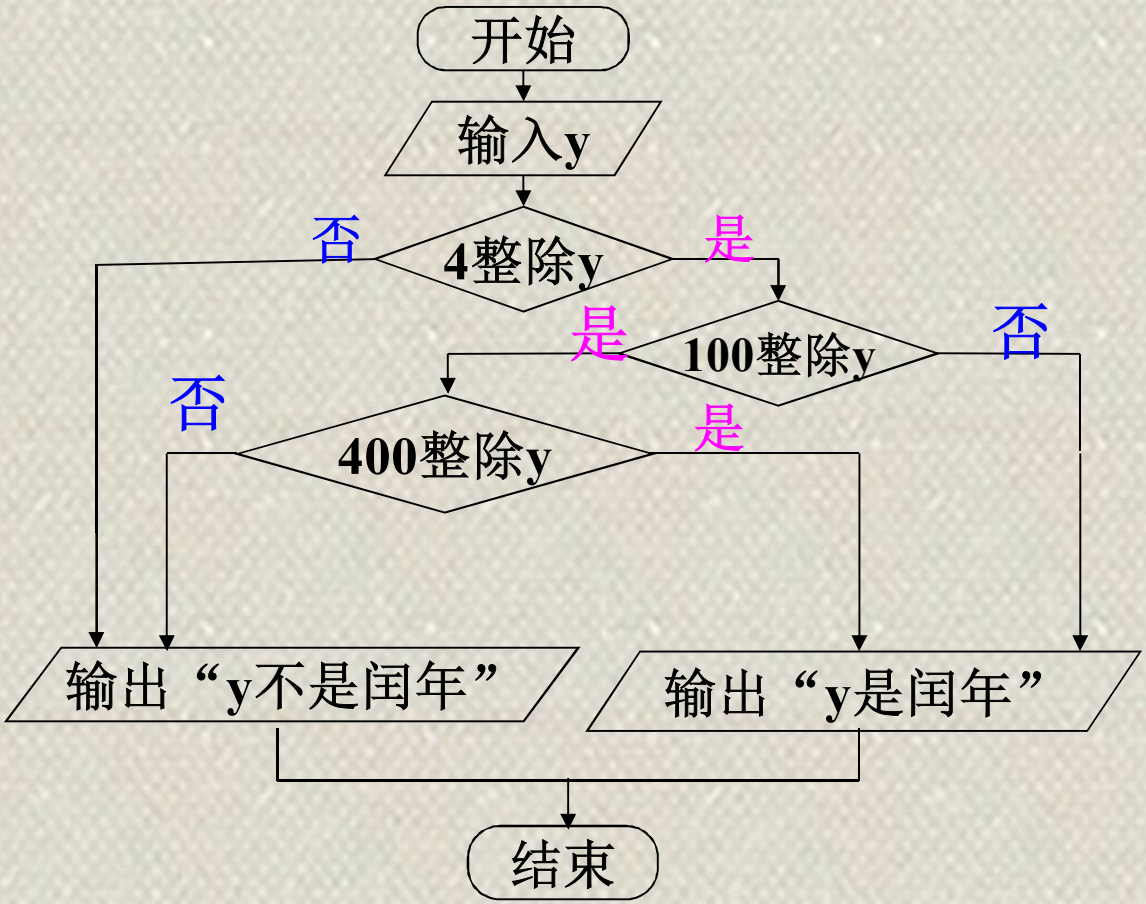
水杯测试用例

- 常温下
- 倒入120摄氏度的水

水杯测试用例

- 常温下
- 倒入80摄氏度的水
- 反复摇晃5分钟

测试用例



一个用例

用例编号	001	用例类型	单元测试
用例名称	判断是否为闰年		
模块名称	日期判定模块		
用例概述	根据输入的年限，判断是否为闰年，而且判断是否有容错技术		
测试环境			
测试目标	判断是否为闰年，而且判断是否有容错技术		
用户需求	判断输入年限是否为闰年		
前置条件	需要用户输入某年限		
后置条件	无		
特殊说明			

用例的测试过程

步骤	测试内容	测试输入数据	操作描述	测试预期结果	测试结果	测试完成后功能描述
1	语句覆盖	2000	输入某年限	1	1	判断是闰年
2	语句覆盖	2001	输入某年限	0	0	判断是闰年
3	条件覆盖	1917	输入某年限	提示年限输入错误	1	没有容错技术
4	条件覆盖	2004	输入某年限	提示年限输入错误	0	抛出异常
5	条件覆盖	1900	输入某年限	1	1	判断是闰年
6	条件覆盖	1600	输入某年限	0	0	判断是闰年
7	基本路径	123457	输入某年限	0	0	判断是闰年
8	基本路径	1267	输入某年限	1	1	判断是闰年

测试人		测试时间	2012.3.7
备注	闰年判别函数测试		

测试用例的重要性

- 合理分析测试的过程
- 测试工作的操作指南
- 保证测试的有效性（可以重复进行）
- 方便反复验证

合理的测试用例

- 公元前10000年和公元后1000005年
- 是不是每个数字都要测试一次

等价类划分

- 进行完全的测试验证是不可能的
- 测试代表性数据

有效等价类

- 输入2000
- 输入2005

无效等价类

- 输入“MM ”(罗马数字2000)
- 输入：0.2 （小数）
- 输入：“贰壹叁伍”
- 输入：-500
- 输入：null