

# Agenda

- Case Study
- Unit & Module testing
- Integration Testing Strategies
- Regression & Smoke Testing
- High Level Testing
- Functional Regression Testing - Example
- Test Planning
- Oracle Problem
- Summary

# Case studies

## How to test it ?



旅行保温杯  
(腾讯测试面试题)

- GB/T 11416-2002 《日用保温容器》
- QB/T 2332-1997 《不锈钢真空保温容器》
- GB 9684-1988 《不锈钢食具容器卫生标准》
- GB/T11681-1989 《不锈钢食具容器卫生标准的分析方法》

**相关强!**



智能手机电容屏保护玻璃  
(产品元件测试)

- GB 11614-1999 《浮法玻璃》
- GB 15763.2-2005 《建筑用安全玻璃-钢化玻璃》

**相关弱!**

**第一步：标准及相关性调查**

# Case studies

## How to test it ?

测试必须综合考虑目的、成本

FURPS+是测试系统化思维框架，包括安全！

- 2009年第2季度，上海市质量技术监督局对本市生产销售的保温容器产品质量进行了专项监督检查。
- 标准**涉及项目**：不锈钢真空保温容器的保温效能、容量、耐冲击、异味、橡胶件的耐热水性、手把的安装牢度、密封性、不锈钢内胆卫生要求（铅、镉）、瓶塞的装配吻合度以及标识项目……。
- 本次**抽查项目**：保温效能、容量、热水泄漏、耐冲击以及标志项目
  - 保温效能：核心指标。功能测试（F / P）
  - 容量：实用指标，容量与口径直接关系效能。实用测试（U）
  - 热水泄漏：涉及使用安全，并影响使用环境的美观。安全测试（Safety）
  - 耐冲击：该项目关系产品使用寿命。可靠性测试（R）
  - 标识：国家标准规定：产品名称、容量、口径、制造厂名和厂址、采用的标准号、使用方法及使用中的注意事项等需明确标注。（S）

假设你是驴友，您还需要旅游保温杯产品有哪些特性？

# Case studies

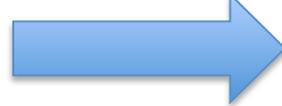
## How to test it ?

- 没有文字的需求说明书能测试吗？
  - 能！
  - 使用FURPS+(Safety & Security)框架去思考
  - 调查或体验用户使用中的常见问题（Failures）
  - 问题分析 → 质量场景



旅行保温杯  
(腾讯测试面试题)

How to test it ?



智能手机电容屏保护玻璃  
(产品元件测试)



智能手机电容屏保护  
玻璃（产品元件测试）

# Case studies

## How to test it ?

- 国家标准主要项目
  - 厚度偏差（F）
  - 机械加工尺寸偏差（供需方自定义）（F）
  - 弯曲度（F）
  - 抗冲击（R / U）
- 手机用户、手机生产商关注的质量要素
  - 划痕数量、深度检测（玻璃磨平工艺）（U）
  - 耐磨性，硬度（玻璃强化工艺）（R / P）
  - 丝印对准度，厚度（白色、黑色或彩色印刷）（U）
  - 色彩纯度（例如：白色一般都偏黄）（U）
  - 手机玻璃破碎条件与伤害（Safety）
  - 产品识别与防伪（S）

# Case studies

How to test it ?

## 腾讯测试面试题 2

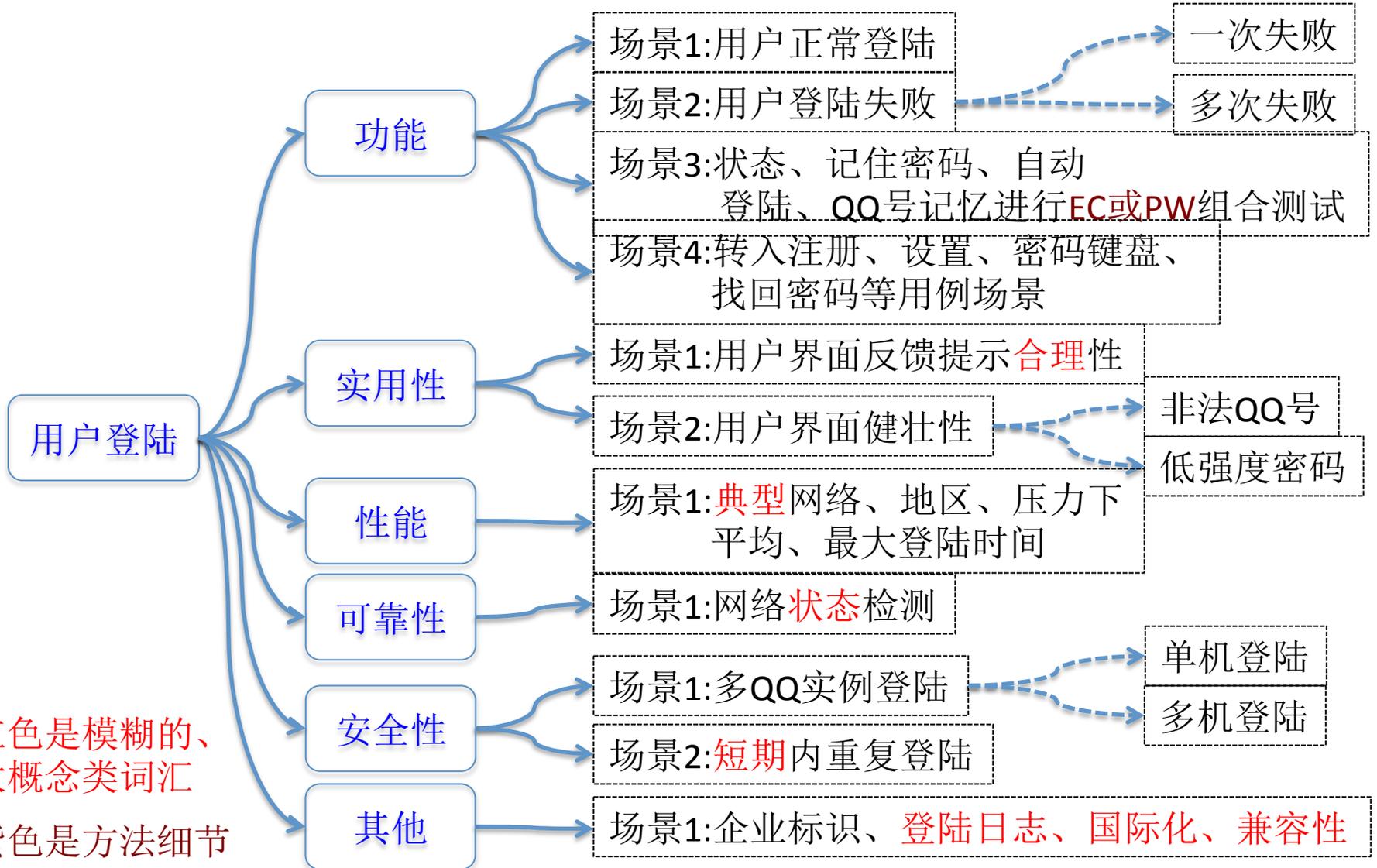


- 1、有需求说明或功能说明书吗？  
有。……
- 2、没有！？  
啊？……，Running……



# Case studies

## Mindmap – A power tools for exchanging ideas



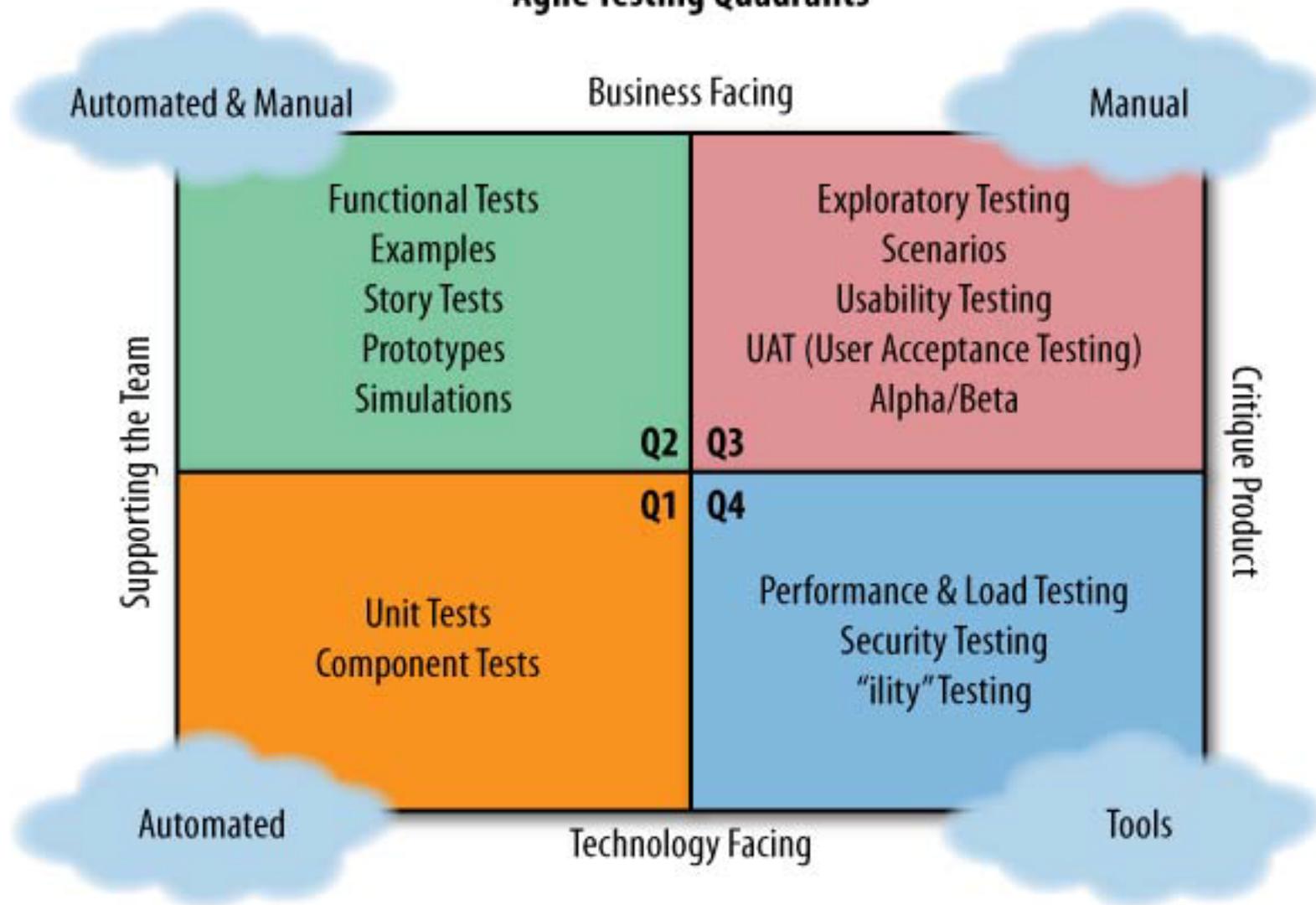
红色是模糊的、大概概念类词汇

紫色是方法细节

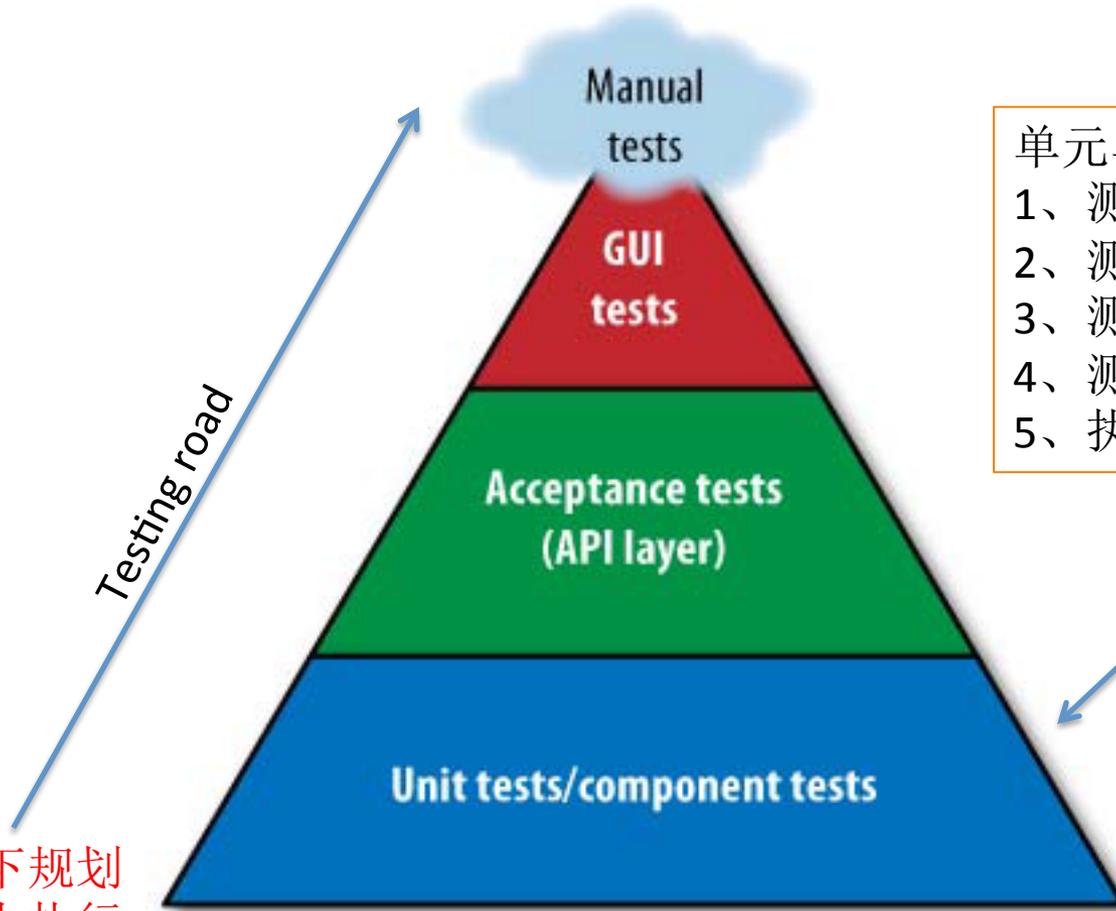
# Case studies

Planning – next jobs?

## Agile Testing Quadrants



# Test Strategy Pyramid

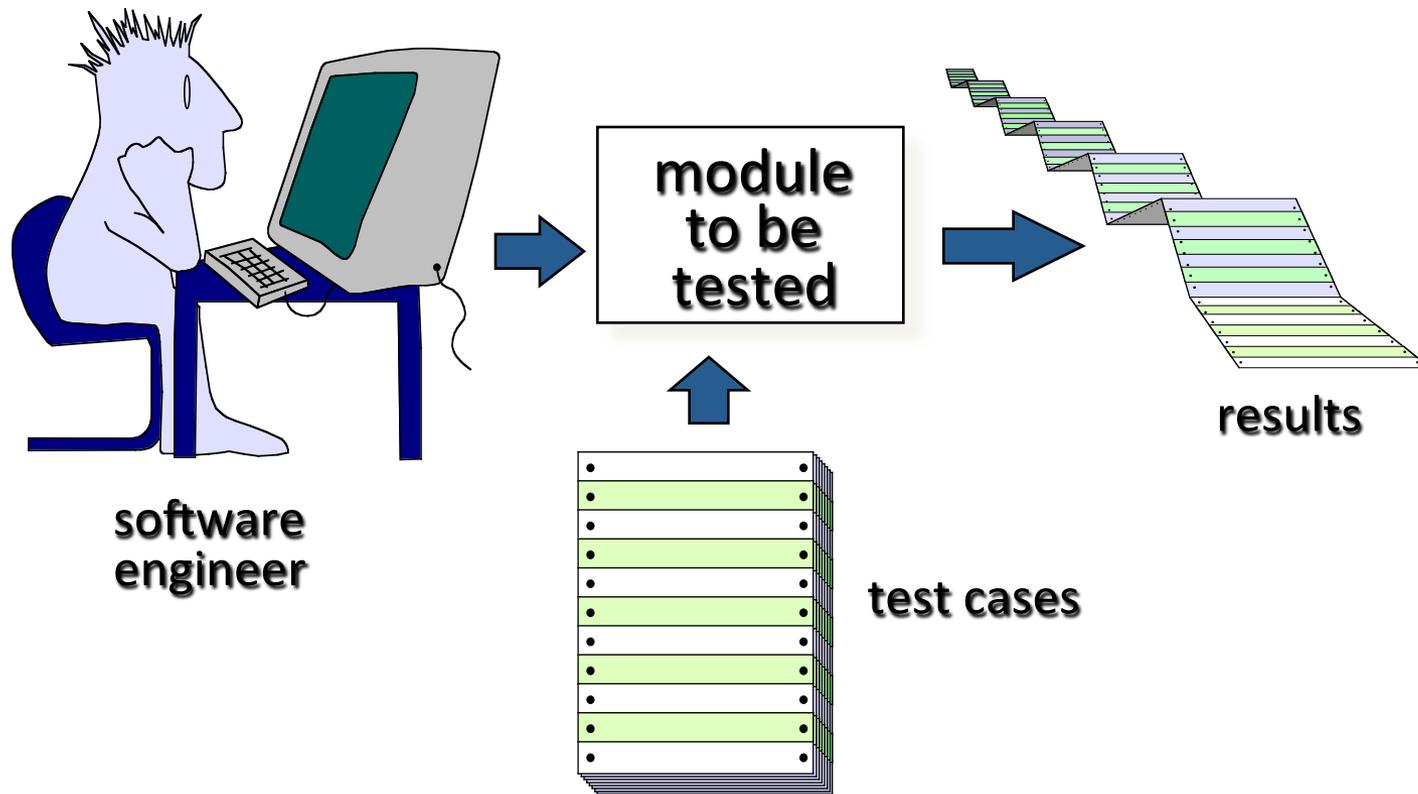


自顶向下规划  
自下向上执行

- 单元与模块测试实践
- 1、测试入 / 出口准则制定
  - 2、测试手段：黑盒 / 白盒
  - 3、测试工具：自动化
  - 4、测试评估：代码覆盖
  - 5、执行人员：程序员 / 测试

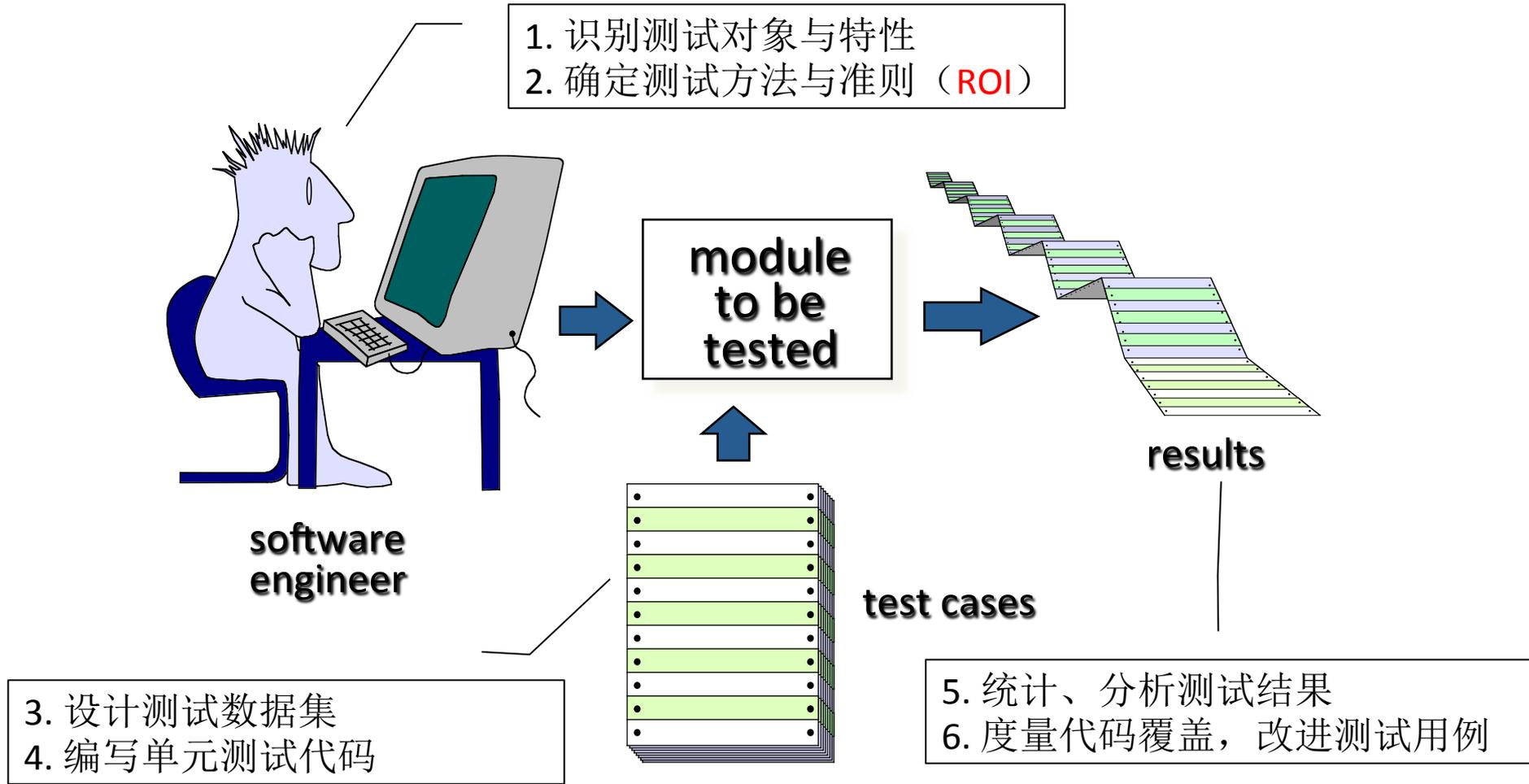
# Unit & Module testing

## Unit Test



# Unit & Module testing

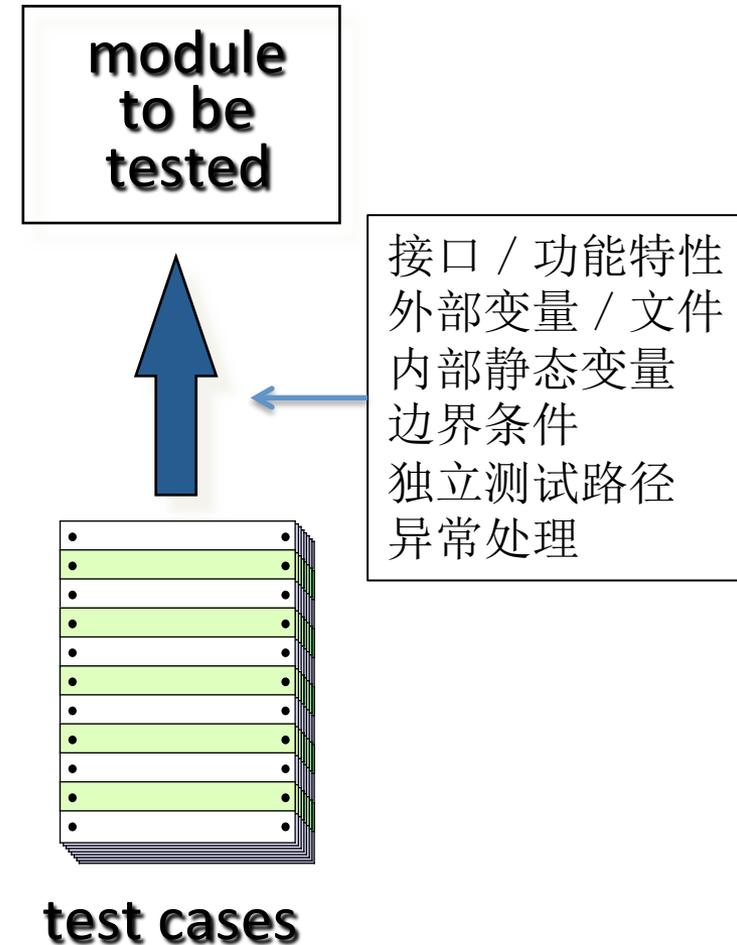
## Unit Test Process



# Unit & Module testing

## Unit Test Criterion – Economic-based

- 识别测试对象
  - 接口方法（是否确定的）
  - 类 / 模块
- 用例设计方法与准则选择
  - 确定的(deterministic)
    - 按基于说明书的方法测试用例
    - 选择黑盒 / 白盒准则
  - 不确定的(non-deterministic)
    - 识别测试动作或数据序列
    - 状态模型
- 测试手段选择
  - 通过测试（静态数据集）
  - 异常测试（静态数据集）
  - 随机测试（动态数据集）

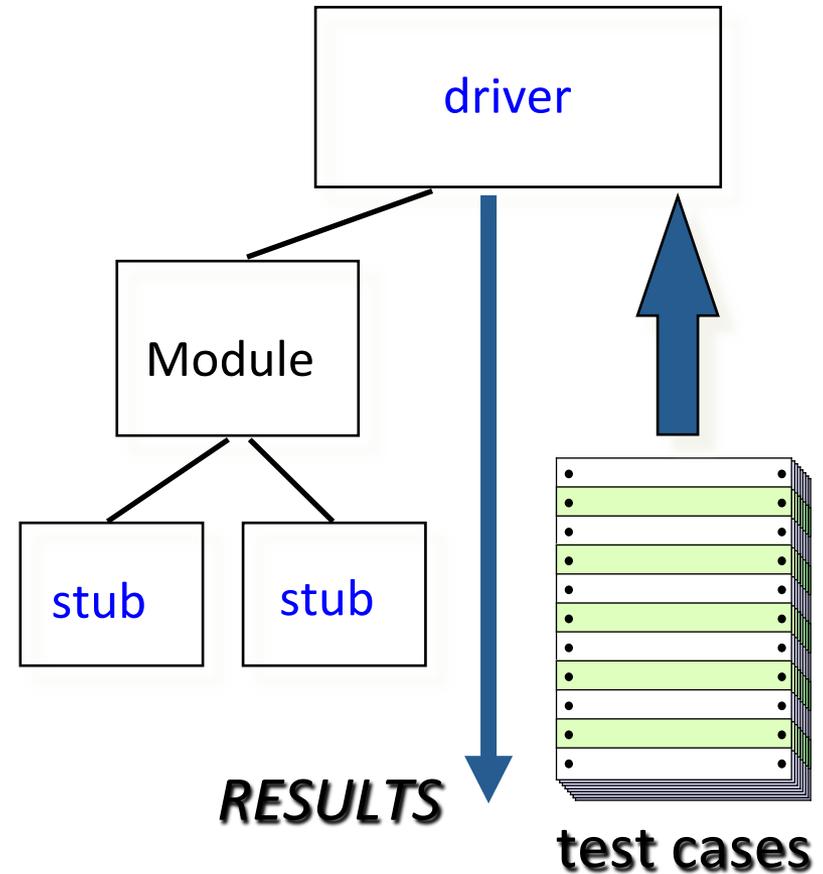


红色：都是高成本的测试方法或准则，请慎重选择！

# Unit & Module testing

## Unit Test Environment

- 驱动(driver)
    - 测试程序
    - 加载Prefix value
    - 加载测试数据（如xml）
    - 执行测试
    - 卸载测试夹具
    - 打印报告
  - 桩(stub)
    - 被测试模块依赖的方法
    - 例如：三角形面积计算程序
      - 桩：三角形判断程序
- ```
int tritype(int a, int b, int c)
{ if (a<=0) return -1; else return 1}
```



敏捷开发：Stub, Mock对象是最常用实践

# Unit & Module testing

## Unit Test - example

- 类contains方法的黑盒测试

```
Class Booklist implementation List {  
    Boolean add( Book book);  
    ... ..  
    Boolean contains(Book book);  
    ... ..  
}
```

- 单元测试步骤

- 识别测试对象(确定的测试吗? )

- Boolean contains(list, book)

- 测试特征

- Characteristics numbers of list (0,1,k)
- Characteristics book (null, book instance, shallow copy of the book, subclass instance)
- Characteristics result (true, false)
- Characteristics found at (first, middle, last)

要点：尽可能把问题表示成确定的测试  
类测试要考虑equals, instance of等特殊算子。

# Unit & Module testing

## Module Test - example

- 类BookDAO接口API测试

- Class BookDAO extends com.xxx.persistant.DAO {
  - void Create(Book book);
  - void Update(Book book);
  - void Delete(Book book); // use id only
  - Book FindBookbyID(string ISBN);
  - Booklist FindBookbyQuery(string whereClause)
  - ... ..
- }

- 模块测试步骤

- 识别测试对象
  - BookDAO类

- 设计方法与准则

- Happy序列：每种方法一次
- 对每个方法输入 / 功能等价划分，选择合适组合策略，覆盖相关准则
- 异常序列：ID重复、Delete before Update 、 ... ..

测试序列：

```
book = new Book("__XXX@__")
```

```
Create(book)
```

```
AssertEqual(FindBookbyID(book.id), book) //有BUG
```

```
//修改book
```

```
Update(book)
```

```
AssertEqual(FindBookbyID(book.id), book)
```

```
AssertEqual(FindBookbyQuery("ID=id")[0], book)
```

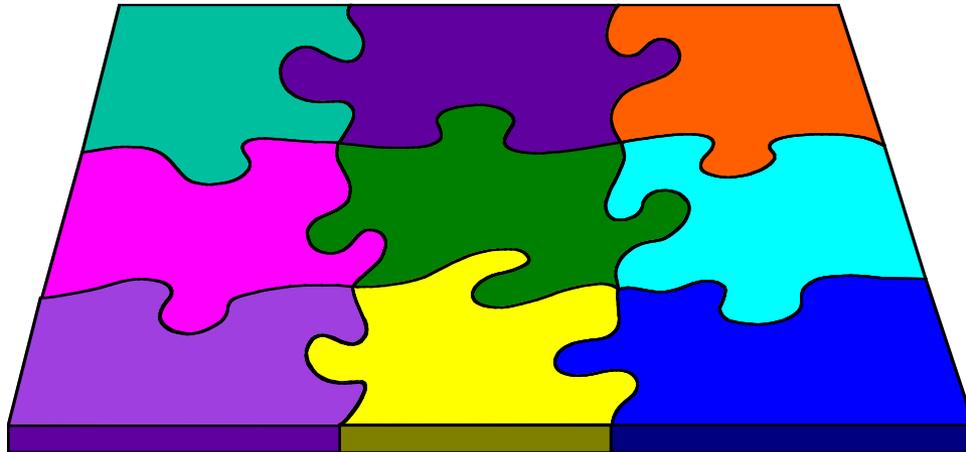
需要更强准则的测试吗？如单元测试、随机测试  
程序员编写 / 测试编写哪个成本低？

# Integration Testing Strategies

the “big bang” approach

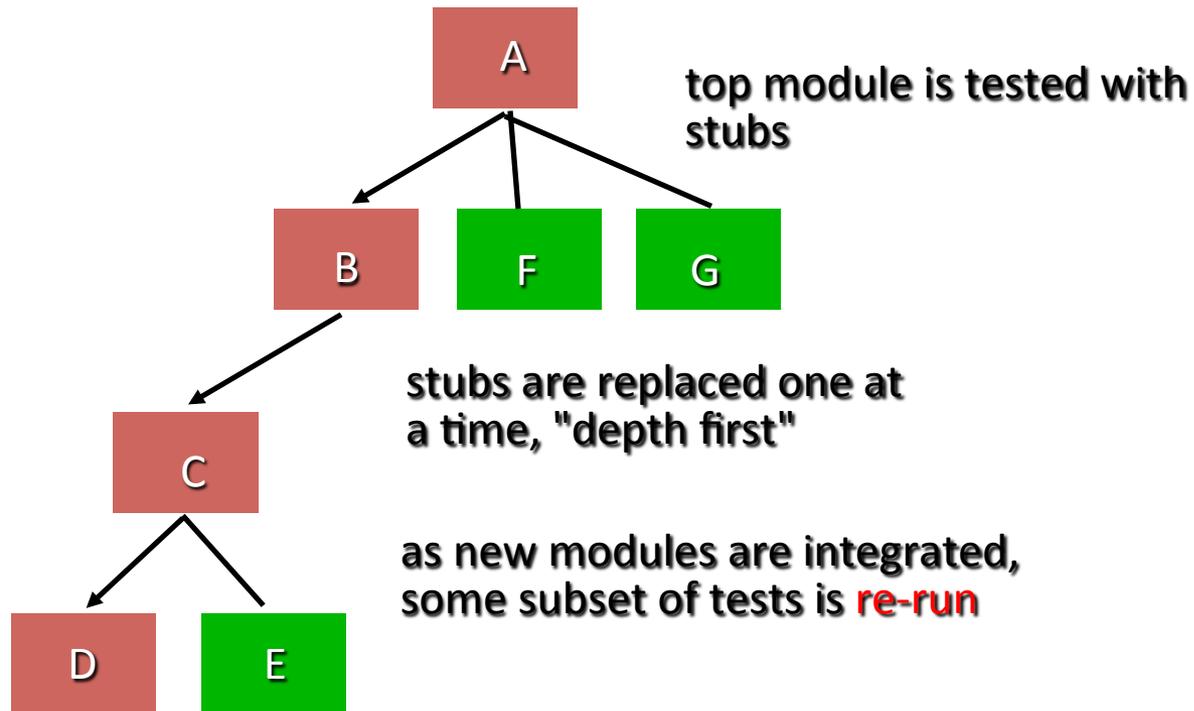
Options:

- the “big bang” approach
- an incremental construction strategy



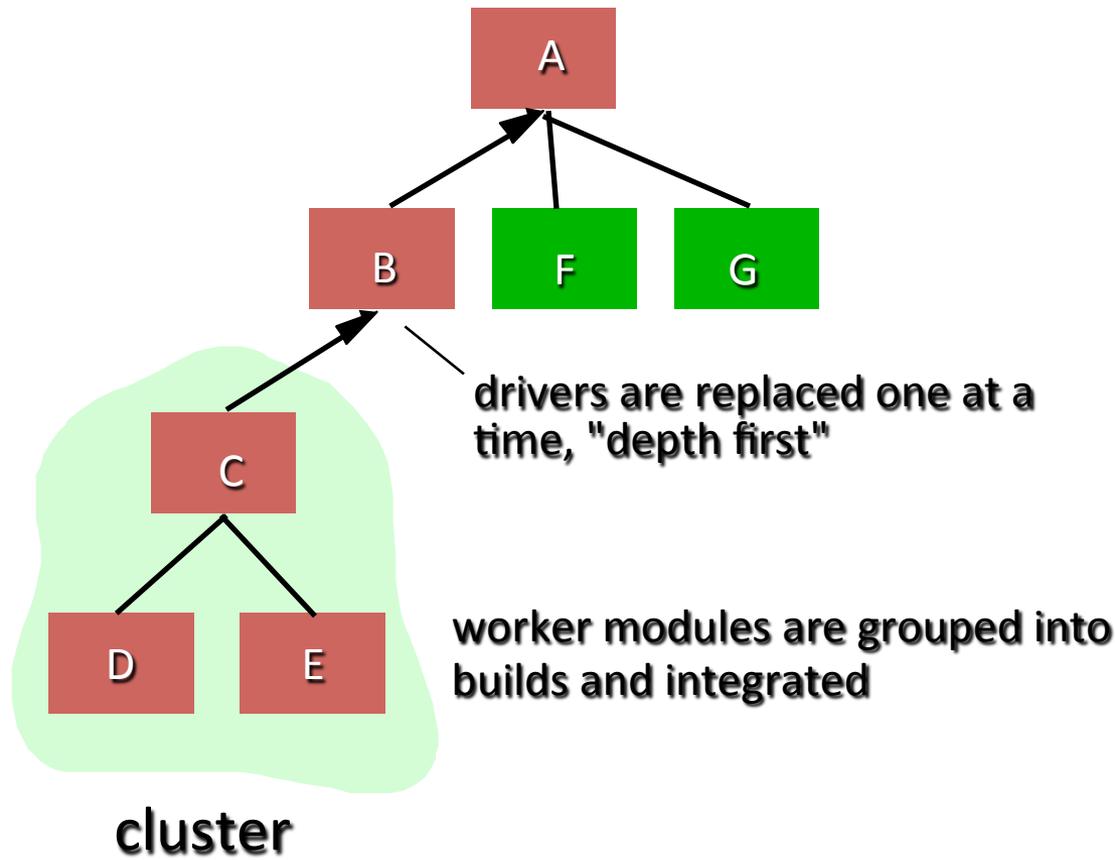
# Integration Testing Strategies

## Top-Down approach



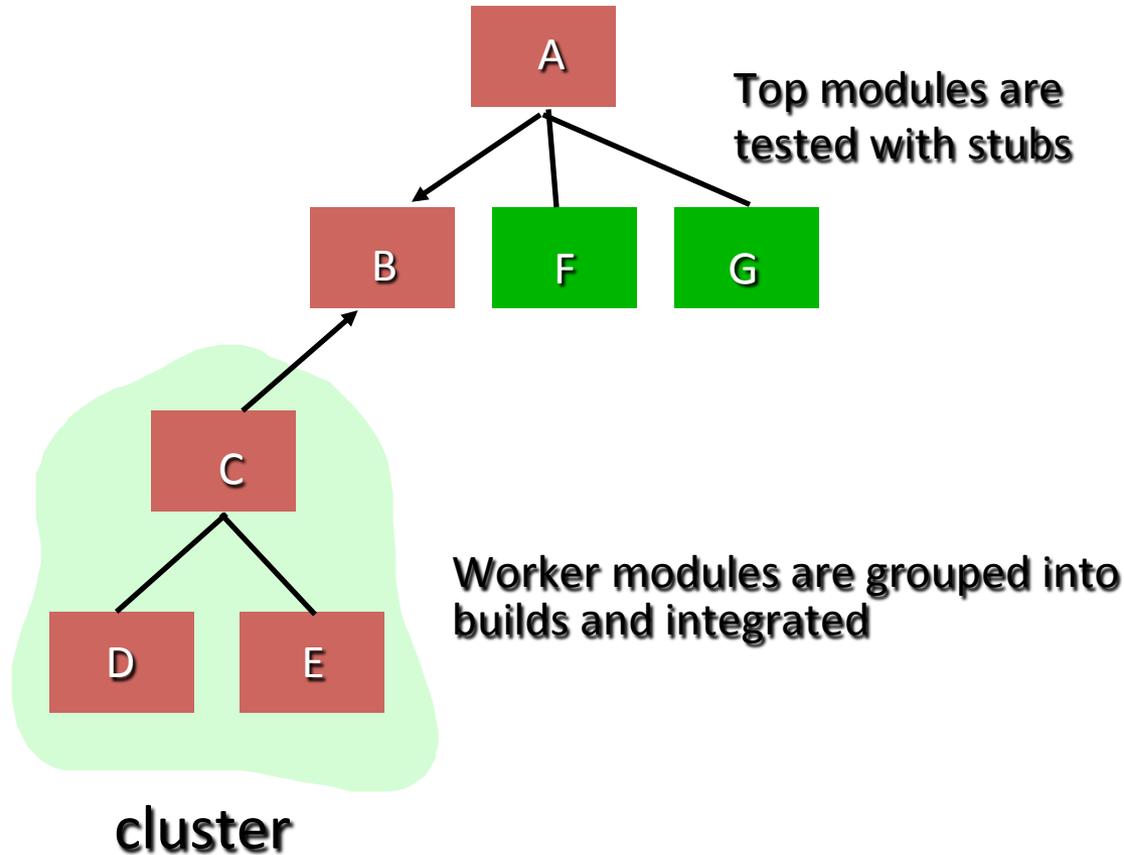
# Integration Testing Strategies

## Bottom-Up approach



# Integration Testing Strategies

## Sandwich approach



# Integration Testing Strategies

## Summary of Integration Approach

|          | “Big Bang” | Top-Down          | Bottom-Up                 | Sandwich                  |
|----------|------------|-------------------|---------------------------|---------------------------|
| 成本       |            |                   |                           |                           |
| 测试驱动     | High-level | High-level        | Level Level<br>High Level | Level Level<br>High Level |
| “桩”程序    | —          | 多                 | 无                         | 适中                        |
| Re-Run次数 | 依赖产品品质     | 多                 | 适中                        | 适中                        |
| 质量控制     |            |                   |                           |                           |
| 关注点      | 产品外在特性     | 产品外在特性            | 内部功能实现                    | 兼顾                        |
| 测试数据     | 接近真实数据     | 早期受stub限制 (happy) | 早期不能测试<br>人机交互            | 兼顾                        |
| 用例设计     | 使用质量场景     | 使用质量场景            | 设计说明<br>→ 质量场景            | 设计说明<br>质量场景              |
| 适用开发场景   |            |                   |                           |                           |
|          | 小规模低质量     | 高层、界面原型           | 底层、核心模块                   | 所有软件开发                    |

# Regression Testing

## Conception

### Definition

**The process of re-testing software that has been modified**

- Most software today **has very little** new development
  - Correcting, perfecting, adapting, or preventing **problems** with existing software
  - Composing new programs from **existing** components
  - **Applying** existing software to new situations
- Because of the deep interconnections among software components, **changes** in one method can cause problems in methods that **seem to be unrelated**
- Not surprisingly, most of our testing effort is **regression testing** (回归测试)

# Regression Testing

## Automation and Tool Support

- **Too many** tests to be run by hand
- Tests must be run and evaluated **quickly**
  - often overnight, or more frequently for web applications
- Testers do not have time to **view** the results by inspection
- Types of **tools** :
  - **Capture / Replay** – *Capture* values entered into a GUI and *replay* those values on new versions
  - **Version control** – Keeps track of collections of *tests*, expected *results*, where the tests *came from*, the criterion used, and their past *effectiveness*
  - **Scripting software** – Manages the process of obtaining test *inputs*, *executing* the software, obtaining the *outputs*, *comparing* the results, and generating *test reports*
- Tools are plentiful and inexpensive (often **free**)
  - Selenium, QTP, Rational Functional Test, Junit ... ..

回归测试用例自动化率是企业测试水平的重要指标

# Regression Testing

## Managing Tests in a Regression Suite

- Test suites **accumulate** new tests over time
- Test suites are usually run in a **fixed, short**, period of **time**
  - Often **overnight**, sometimes more frequently, sometimes less
- At some point, the number of tests can become **unmanageable**
  - We cannot finish running the tests in the time allotted
- We can always add **more computer** hardware
- But is it **worth** it ?
- How many of these tests really need to be run ?

# Regression Testing

## Policies for Updating Test Suites

- Which tests to keep can be based on several policies
  - Add a new test for every **problem report**
  - Ensure that a **coverage criterion** is always satisfied
- Sometimes harder to choose tests **to remove**
  - Remove tests that **do not contribute** to satisfying coverage
  - Remove tests that have **never found a fault** (risky !)
  - Remove tests that have found the **same fault** as other tests (also risky !)
- **Reordering** strategies
  - If a suite of N tests satisfies a coverage criterion, the tests can often be reordered so that the first N-x tests satisfies the criterion – so the remaining tests can be removed

# Regression Testing

## When a Regression Test Fails

- Regression tests are evaluated based on whether the result on the new program P is **equivalent to** the result on the previous version P-1
  - If they **differ**, the test is considered to have **failed**
- Regression test failures represent **three possibilities** :
  - The software has a fault – *Must fix the fix*
  - The test values are no longer valid on the new version – *Must delete or modify the test*
  - The expected output is no longer valid – *Must update the test*
- Sometimes **hard to decide** which !!

# Regression Testing

## Change Impact Analysis

- When a **small change** is made in the software, what portions of the software can be **impacted** by that change ?
- More directly, which tests need to be re-run ?
  - Conservative approach : Run all tests
  - Cheap approach : Run only tests whose test requirements relate to the statements that were changed
  - Realistic approach : Consider how the changes propagate through the software
- Clearly, tests that **never reach** the modified statements do not need to be run
- Lots of **clever algorithms** to perform CIA have been invented
  - Few if any available in commercial tools

# Smoke Testing

## Conception

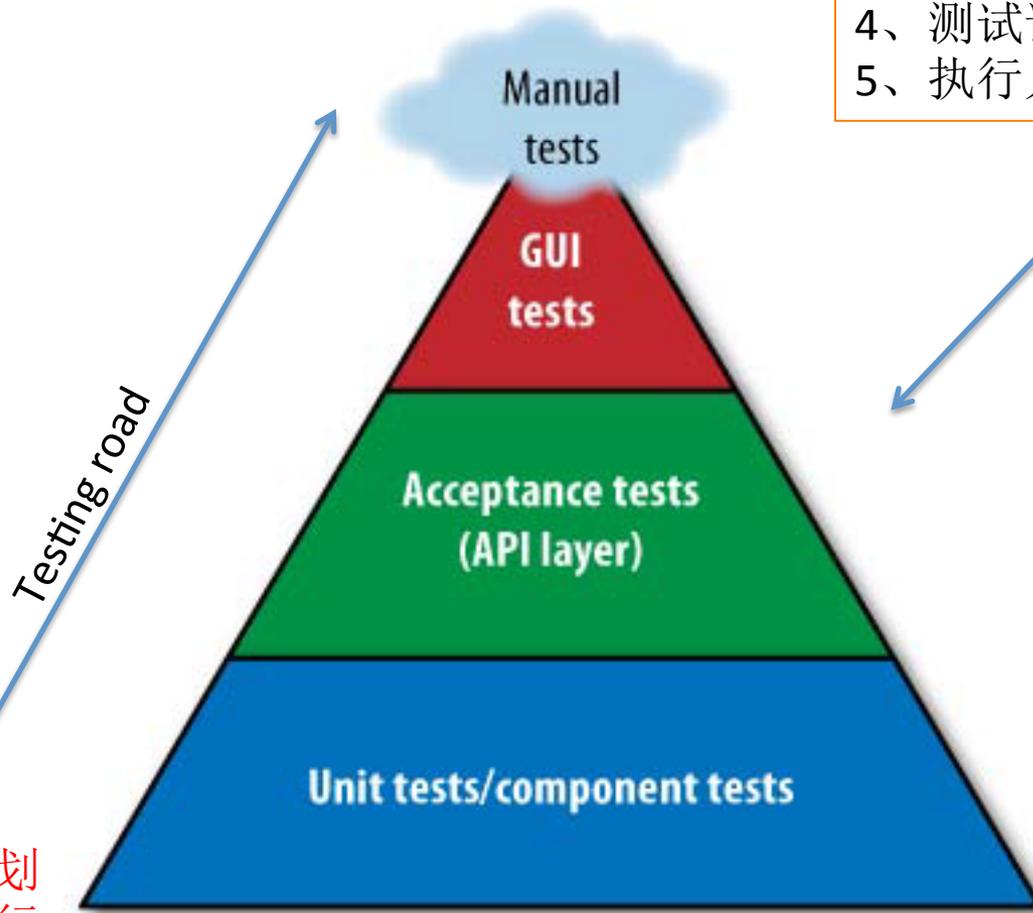
- The term smoke testing originated in the hardware industry
- The term derived from this practice:
  - After a piece of hardware or a hardware component was changed or repaired, the equipment was simply powered up. If there was no smoke, the component passed the test.
- A common approach for creating “daily builds” for product software
- Smoke testing steps:
  - Software components that have been translated into code are integrated into a “build.”
  - A series of tests is designed to expose errors that will keep the build from properly performing its function.
  - The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.

# Smoke Testing

## VS. Regression Testing

- 共同点
  - 都是bug fixed后开展的测试
  - 冒烟测试用例集是回归测试子集，去除了不合适每日构建测试相关的测试用例
- 不同点
  - 回归测试关注修改不影响系统的功能，或修改没有引入新的BUG
  - 冒烟测试关注修改模块或新增特性被实现
  - 回归测试测试用例覆盖要求高，测试周期和成本较高
  - 冒烟测试覆盖要求低，可选择比较弱测试准则，和易于发现错误的测试用例作为测试集合
  - 通常产品迭代结束执行回归测试
  - 冒烟测试配合daily build，及时发现新代码中问题

# Test Strategy Pyramid



## 高层测试实践

- 1、测试入 / 出口准则制定
- 2、测试依据：用户需求
- 3、测试工具：手工 / 自动化
- 4、测试评估：按测试设计
- 5、执行人员：测试 / 用户

自顶向下规划  
自下向上执行

# High Level Testing

## Overview

- Validation testing
  - Focus is on software requirements
- Alpha/Beta testing
  - Focus is on customer usage
- System testing
  - Focus is on system integration
- Recovery testing
  - forces the software to fail in a variety of ways and verifies that recovery is properly performed
- Security testing
  - verifies that protection mechanisms built into a system will, in fact, protect it from improper penetration
- Stress testing
  - executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- Performance Testing
  - test the run-time performance of software within the context of an integrated system
- Compatibility Testing , Localization... ..

# Functional Regression Testing

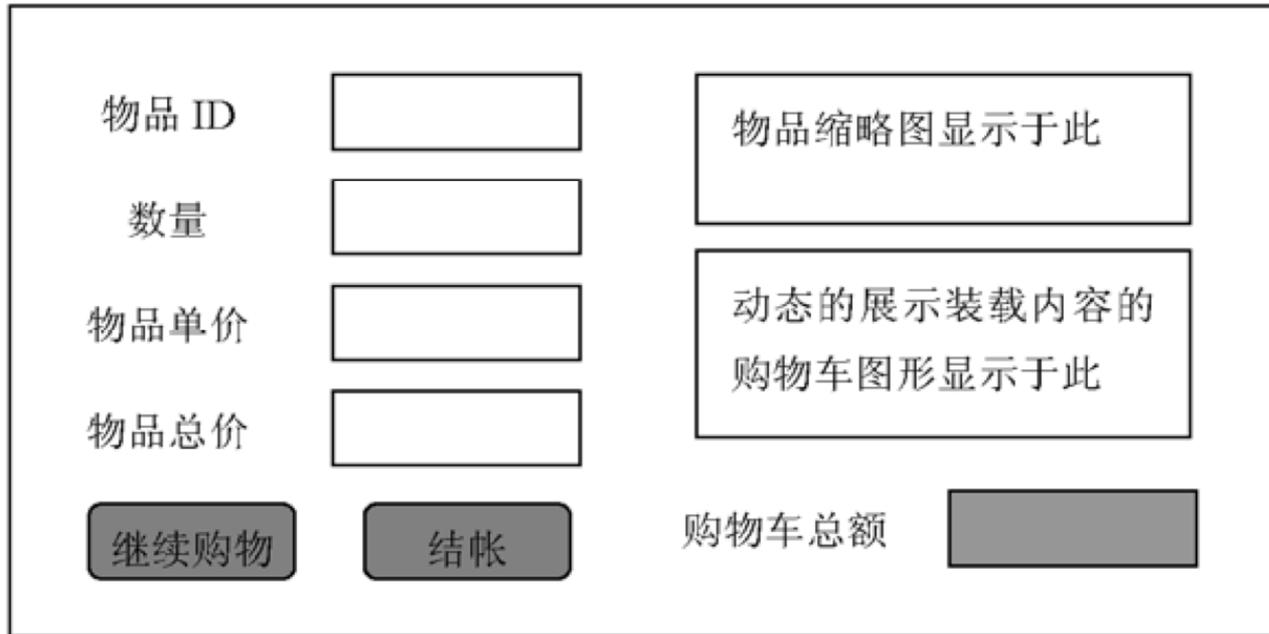
## practical example -process

- 选择测试对象
- 确定测试准则、系统和数据环境
- 定义、录制功能测试用例
- 执行测试用例
  - 构建系统
  - 执行单元、模块测试
  - 执行功能测试
  - 输出测试报告
- 工具
  - 自动构建工具（buildbot）
  - 单元测试工具（JUnit）
  - 功能自动测试工具（Selenium, QTP, Rational Functional Tester）
  - 手动测试

# Functional Regression Testing

## practical example - AuT

- 测试对象，一个简单订单网页界面



物品 ID

数量

物品单价

物品总价

购物车总额

物品缩略图显示于此

动态的展示装载内容的购物车图形显示于此

- ID: 00001—99999之间；输入ID自动显示单价、物品图片
- 数量: 0—99之间；输入0,则从购物车中删除该商品；物品总价 = 单价 \* 数量
- 购物车最大总额: 999.99

# Functional Regression Testing

## practical example – test case design

- 测试方法（黑盒）
  - 等价划分
  - 边界值
  - 数据组合策略
- 用例模板如右图：
  - 请放开思路
  - 模板可修改

| 测试编号  | 1 | 2 |  |  |  |
|-------|---|---|--|--|--|
| 输入、动作 |   |   |  |  |  |
| 物品ID  |   |   |  |  |  |
| 数量    |   |   |  |  |  |
|       |   |   |  |  |  |
| 预期结果  |   |   |  |  |  |
| 物品单价  |   |   |  |  |  |
| 购物车内容 |   |   |  |  |  |
| 购物车总价 |   |   |  |  |  |
|       |   |   |  |  |  |

### 测试思路：

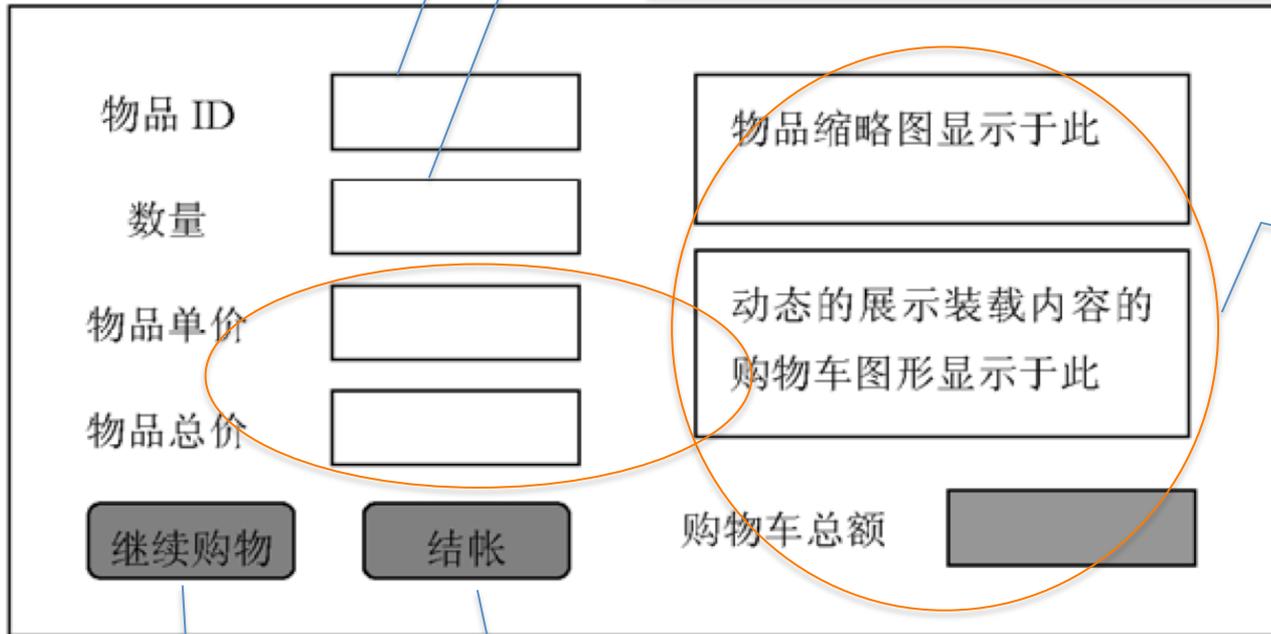
- 1、考虑功能需求
- 2、判定确定性
- 3、识别功能性、实用性质量场景
- 4、黑盒方法应用
- 5、构造测试序列

# Functional Regression Testing

practical example – Interface-based approach

1、（空，非法输入，（物品存在，不存在））

2、（空，非整数，（<0，0，1..99，>99））



3、不能接受输入

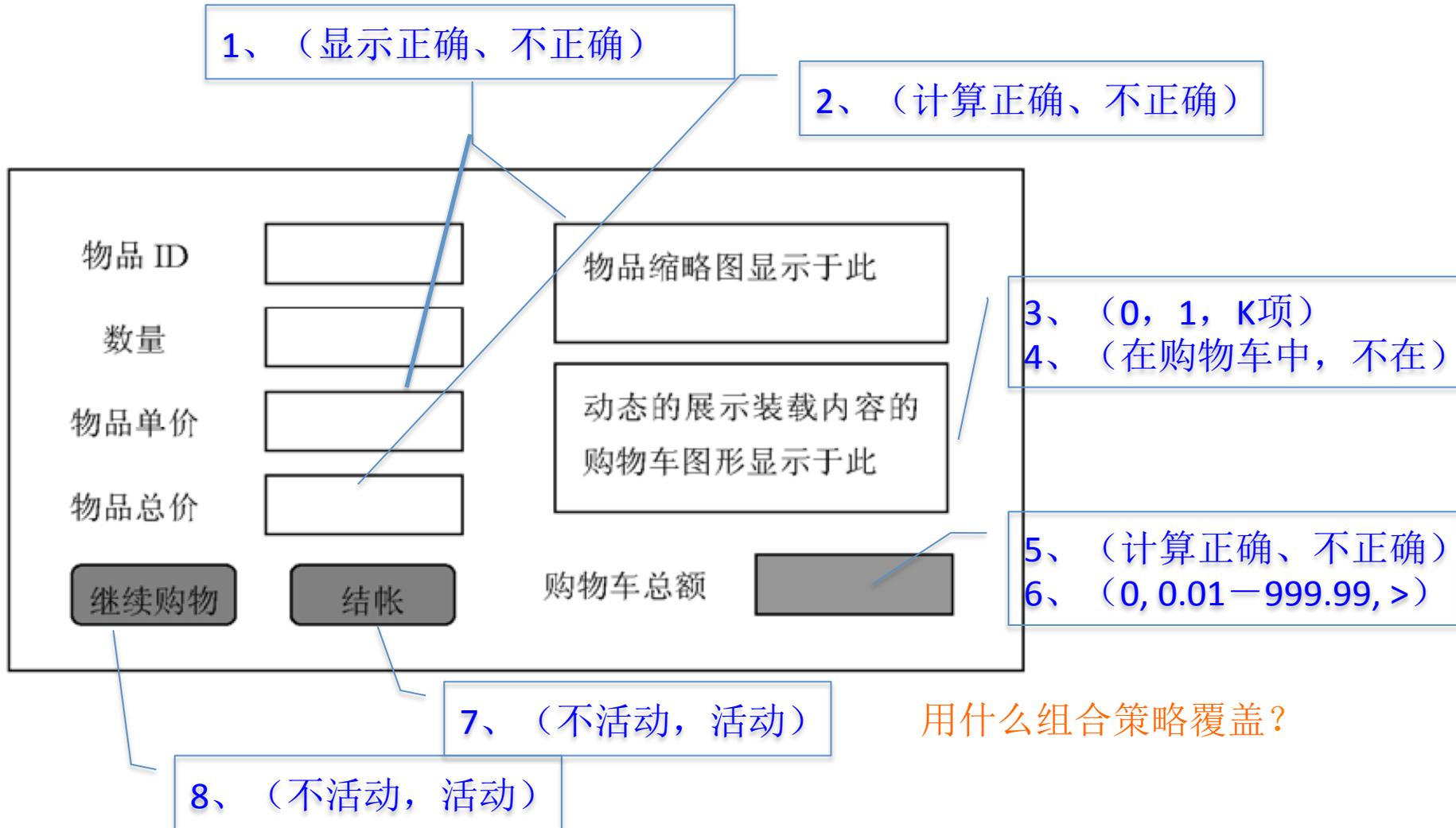
4、动作

5、动作

用什么组合策略？  
有多少可能的组合？  
合法的输入组合是？  
边界条件？

# Functional Regression Testing

practical example – Functionality-based approach



# Functional Regression Testing

## practical example – Test Sequence and Data(1)

- Happy path
  - 前置条件=初始状态?
  - P1: 输入存在的商品编号=1, 检查?
  - P2: 数量=1, 检查?
  - P3: 按继续, 检查?
  - P4: 按结帐,
  - 后置条件=?

确保该物品单价 < 999.99 / 100

|                                                                             |                                                                             |                                       |
|-----------------------------------------------------------------------------|-----------------------------------------------------------------------------|---------------------------------------|
| 物品 ID                                                                       | <input type="text"/>                                                        | 物品缩略图显示于此                             |
| 数量                                                                          | <input type="text"/>                                                        | 1 <input checked="" type="checkbox"/> |
| 物品单价                                                                        | 1 <input checked="" type="checkbox"/>                                       | 动态的展示装载内容的<br>购物车图形显示于此               |
| 物品总价                                                                        | 2 <input checked="" type="checkbox"/>                                       | 3 <input checked="" type="checkbox"/> |
| <input type="button" value="继续购物"/>                                         | <input type="button" value="结帐"/>                                           | 购物车总额 <input type="text"/>            |
| 0 <input checked="" type="checkbox"/> 2 <input checked="" type="checkbox"/> | 0 <input checked="" type="checkbox"/> 3 <input checked="" type="checkbox"/> |                                       |

# Functional Regression Testing

## practical example – Test Sequence and Data(2)

- Test to pass (手工实现)
  - 前置条件=初始状态?
  - P1: 输入存在的商品编号=1, 数量=5, 按继续, 检查?
  - P2: 输入存在的商品编号=1, 数量=99, 按继续, 检查?
  - P3: 输入存在的商品编号=999, 数量=2, 按继续, 检查?
  - P4: 输入存在的商品编号=99999, 数量=99, 按继续, 检查? 应不能加入购物车
  - P5: 修改输入数量=1, 检查?
  - P6: 按继续, 修改输入存在的商品编号=888, 按继续, 检查?
  - P7: 修改输入数量=0, 检查? 按结帐
  - 后置条件=?

到目前为止, 哪些测试需求没有实现?

|                                     |                                   |                            |
|-------------------------------------|-----------------------------------|----------------------------|
| 物品 ID                               | <input type="text"/>              | 物品缩略图显示于此                  |
| 数量                                  | <input type="text"/>              |                            |
| 物品单价                                | <input type="text"/>              | 动态的展示装载内容的<br>购物车图形显示于此    |
| 物品总价                                | <input type="text"/>              |                            |
| <input type="button" value="继续购物"/> | <input type="button" value="结帐"/> | 购物车总额 <input type="text"/> |

# Functional Regression Testing

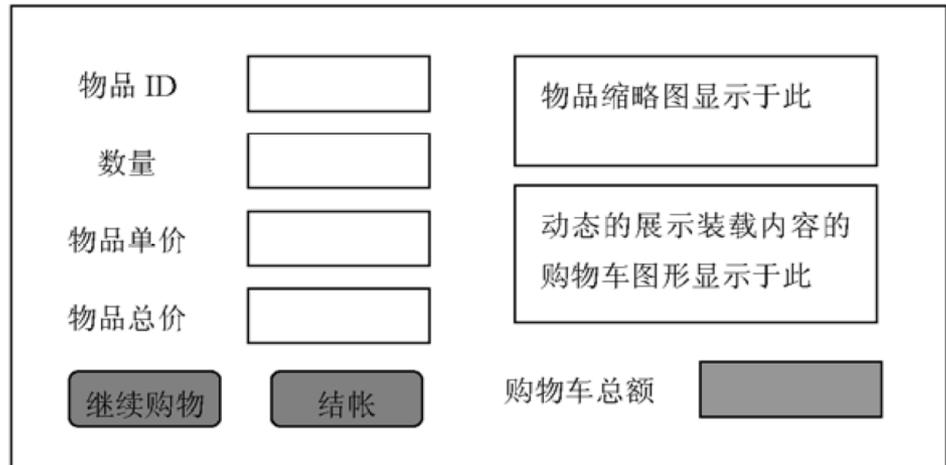
## practical example – Test Sequence and Data(3)

- Test to fail (手工实现)
  - 前置条件=初始状态?
  - P1: 输入存在的商品编号=1, 数量=5, 按继续, 检查?
  - P2: 输入数量=(空, 100, 大于100), 检查?
  - P3: 输入数量=5, 修改商品编号(-1, 空, 0, 100000, 含非法字符, 很长的ID), 检查?
  - P4: 输入存在的商品编号=1, 数量, 按继续, 检查?
  - 后置条件=?

到目前为止, 哪些测试需求没有实现?

- 1、购物车总额的边界没有测试哦!
- 2、物品不在购物车中, 数量=0!  
在哪里添加需要测试呢?

请按孙老师的模板, 写出测试数据!



|                                     |                                   |                            |
|-------------------------------------|-----------------------------------|----------------------------|
| 物品 ID                               | <input type="text"/>              | 物品缩略图显示于此                  |
| 数量                                  | <input type="text"/>              | 动态的展示装载内容的<br>购物车图形显示于此    |
| 物品单价                                | <input type="text"/>              |                            |
| 物品总价                                | <input type="text"/>              |                            |
| <input type="button" value="继续购物"/> | <input type="button" value="结帐"/> | 购物车总额 <input type="text"/> |

# Functional Regression Testing

## practical example – Automatic

- 参见IBM功能测试过程（见网站）
- 自动测试与手工测试的区别
  - 手工测试
    - 尽可能少的测试案例，实现覆盖目标
    - 每个案例依据上下文密集设置检查内容
  - 自动测试
    - 尽可能保证每个测试案例的独立性，少的检查点（减少编程量！）
    - 每个案例用于发现一种或一类BUG，使测试报告易于理解
- 自动测试用例的一种组织方案
  - Happy path test
  - Test to pass
    - 购物车项目从1个变0个不能结帐（很容易出现的错误）
    - 购物车项目可以正确添加到多个，以及输入边界测试
    - 购物车项目可以正确修改、删除，以及购物总额边界测试
  - Test to fail
    - 输入项划分正交测试，检测非法输入处理；处理后能正常使用；

# Test Plans

documentation?

- The most common question about testing is

“ How do I write a test plan? ”

- This question usually comes up when the focus is on the **document**, not the **contents**
- It's the contents that are important, not the structure
  - Good testing is more important than proper documentation
  - However – documentation of testing can be very helpful
- Most organizations have a list of topics, outlines, or templates

# Test Plans

## Standard Test Plan

- ANSI / IEEE Standard 829-1983 is ancient but still used

### Test Plan

**A document describing the scope, approach, resources, and schedule of intended testing activities. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency planning.**

- Many organizations are required to adhere to this standard
- Unfortunately, this standard emphasizes documentation, not actual testing – often resulting in a well **documented vacuum**

# Test Plans

## Types of Test Plans

- Mission plan – tells “why”
  - Usually one mission plan per organization or group
  - Least detailed type of test plan
- Strategic plan – tells “what” and “when”
  - Usually one per organization, or perhaps for each type of project
  - General requirements for coverage criteria to use
- Tactical plan – tells “how” and “who”
  - One per product
  - More detailed
  - Living document, containing test requirements, tools, results and issues such as integration order

# Test Plans

## Test Plan Contents – Strategic Testing

- Purpose
- Target audience and application
- Deliverables
- Information included
  - Introduction
  - Test items
  - Features tested
  - Features not tested
  - Test criteria
  - Pass / fail standards
  - Criteria for starting testing
  - Criteria for suspending testing
  - Requirements for testing restart
  - Hardware and software requirements
  - Responsibilities for severity ratings
  - Staffing & training needs
  - Test schedules
  - Risks and contingencies
  - Approvals

# Test Plans

## Test Plan Contents – Tactical Testing

1. Purpose
2. Outline
3. Test-plan ID
4. Introduction
5. Test reference items
6. Features that will be tested
7. Features that will not be tested
8. Approach to testing (criteria)
9. Criteria for pass / fail
10. Criteria for suspending testing
11. Criteria for restarting testing
12. Test deliverables
13. Testing tasks
14. Environmental needs
15. Responsibilities
16. Staffing & training needs
17. Schedule
18. Risks and contingencies
19. Approvals

# Test Planning

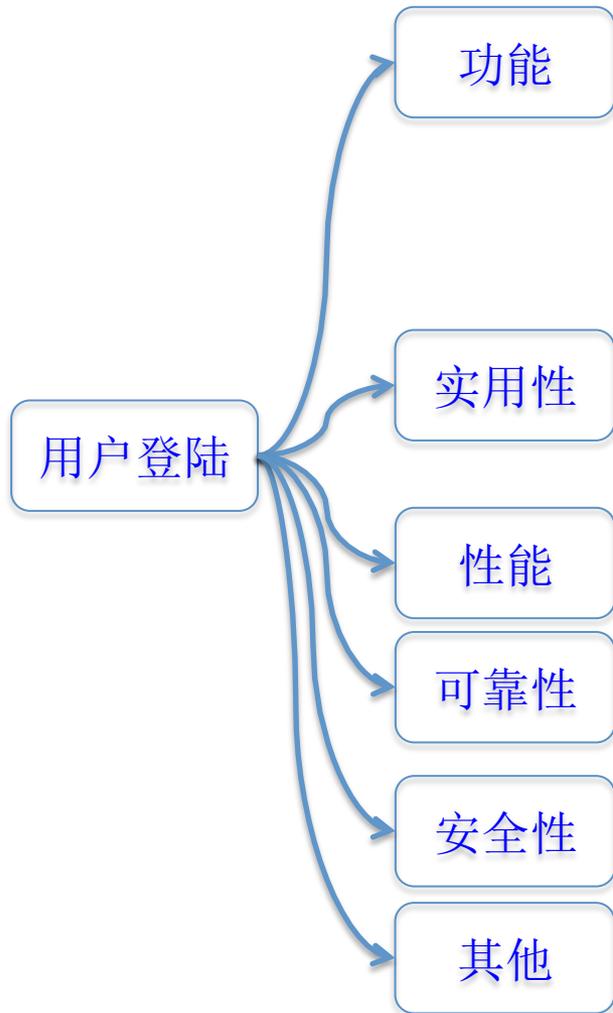
## Example

腾讯测试面试题 2



# Test Planning

Example - Strategic plan - “what” and “when”



| What       |   | When   | Notes  |
|------------|---|--------|--------|
| 概要、目标      | ✓ | 项目启动   | 商业质量目标 |
| 策略         | ✓ | 项目启动   | 测试组织   |
| 功能测试       | ✓ | 每个迭代   | 回归测试   |
| 组件测试       | ✓ | 每天     | 冒烟测试   |
| 核心组件       | ✗ | 每个迭代   | 白盒测试   |
| 集成测试       | ✓ | 每天     | 冒烟测试   |
| 系统测试 (GUI) | ✓ | 每个迭代   | 回归测试   |
| 性能、压力测试    | ✓ | 2-3迭代后 | 关键指标   |
| 互操作性测试     | ✗ | 2-3迭代后 | 架构完成后  |
| 安全测试       | ✓ | 发布阶段前  | 80%完成  |
| 国际化测试      | ✗ | 2-3迭代后 | 架构完成前  |
| 部署、兼容测试    | ✓ | 2-3迭代后 | 架构完成后  |
| 健壮性测试      | ✓ | 发布阶段前  | 80%完成  |

# Test Planning

## Example - Strategic plan - “what” and “when”

- 案例：功能测试策略

| 属性     | 如何考虑该属性                                                   | Notes                |
|--------|-----------------------------------------------------------|----------------------|
| 介绍     | 质量目标：功能覆盖100%<br>测试目标：如自动化率70%                            |                      |
| 交付物    | 每个迭代测试用例规格说明                                              | 参考开发计划               |
| 关键质量情景 | 新特性；产品的兼容性，<br>安全提示，（见需求规格XX章）                            | 参考需求说明               |
| 测试方法   | 黑盒（组合覆盖准则，活动图覆盖准则、状态图覆盖准则、边界覆盖准则、逻辑测试...）<br>自动化测试回归工具... | 应大致说明不同准则的使用条件，以减低成本 |
| 交付条件   | 代码覆盖率60%；所有用例通过                                           |                      |
| 测试环境需求 | 自动构建+测试框架                                                 | 需要软件与设备              |
| 培训     | 业务培训，测试技术培训                                               |                      |

在一定质量标准下，策略（high Level）文档差不多。但存在新产品、升级、维护差别

# Test Planning

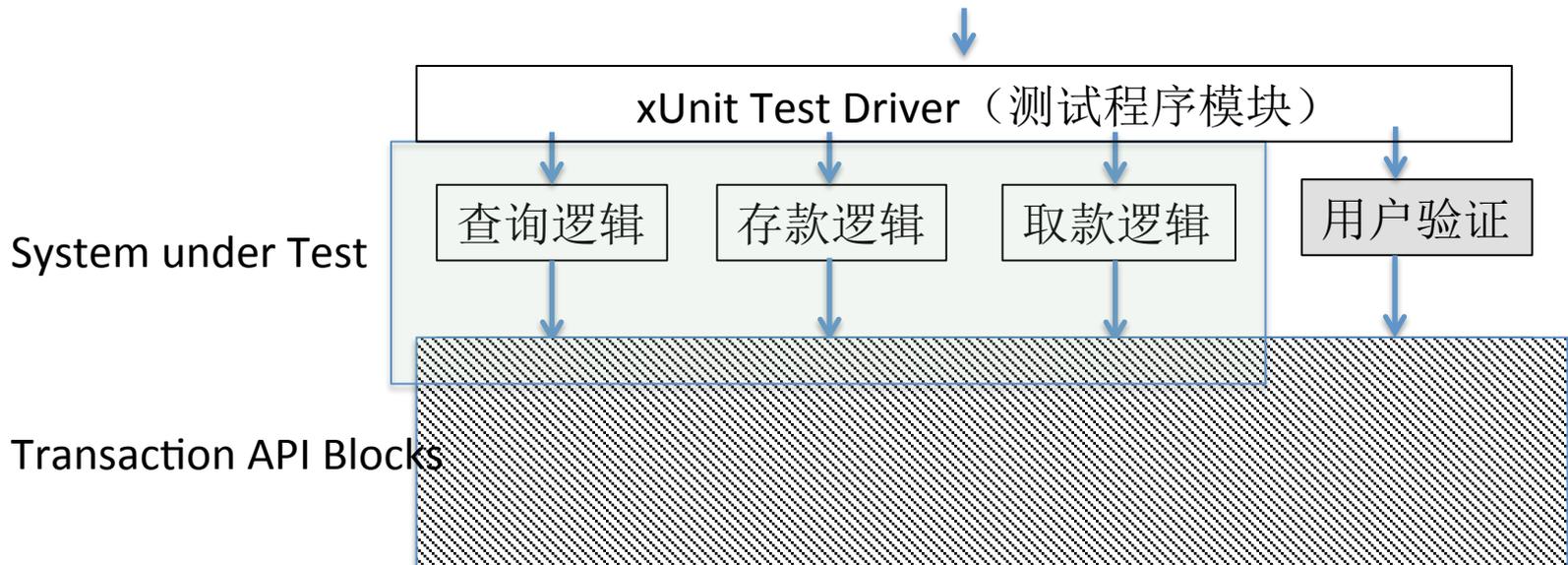
## Example – Tactical Plan- “how” and “who”

- 测试工作计划要点
  - 按项目开发迭代计划逐步完善
  - 按策略说明定义当前阶段目标和测试人员与方案
  - 测试内容按当前系统风险情况调整
- 工作计划组织
  - 1..2: 描述本阶段的质量目标，实施的测试策略内容
  - 3: Project-XXX-Test-YYY。XXX项目编号号，YYY开发计划编号或迭代编号
  - 4..15: 建议按不同策略内容分别编写需要的内容
  - 16: 本阶段涉及的组织，成员的职责
  - 17: 将不同的测试策略，使用甘特图描述测试培训、需求分析、设计、实施、总结等测试活动的汇总安排
  - 17,18: 风险识别与干预措施；负责人签名

# Test Planning

## Example – Tactical Plan- “how” and “who”

- ATM开发第一次迭代计划
- Purpose
  - 对ATM基础业务逻辑完成Happy path单元与集成测试
- Plan ID: P-2012-03-Test-001



ATM程序单元与集成测试结构图，见PPT05

# Test Planning

## Example – Tactical Plan- “how” and “who”

| 主要属性          | 属性描述                           | Notes            |
|---------------|--------------------------------|------------------|
| 介绍            | 基础业务模块单元与集成测试                  |                  |
| 参考文献          | 存取款业务单元API设计说明<br>交易API安装使用说明  | 最好具体到章节          |
| 测试特性<br>不测试特性 | 测试帐户存取款happy操作<br>用户验证过程       |                  |
| 测试实现          | 方法：黑盒，Happy动作序列<br>工具：JUNIT    | 用例设计001-01       |
| 通过准则<br>挂起准则  | 代码覆盖率60%；所有用例通过<br>交易API环境安装失败 |                  |
| 测试交付物         | 用例设计<br>用例程序，Junit报告           | 设计文档尽可能采用doc工具生成 |
| 测试环境          | 验证Mock对象；POS开发机及服务器            | 常需要测试框架图         |
| 培训需求          | 用例设计探讨（2小时以内）                  |                  |

# Test Planning

## 课堂讨论 - 功能测试案例设计与计划安排



# Identifying Correct Outputs

## Oracle Problem

- Oracle (神谕)
- Test Oracle Problem
  - 判断一个测试是否通过
- With **simple software** methods, we have a very clear idea whether outputs are correct or not
- But for most programs it's **not so easy**
- This section presents **four general methods** for checking outputs:
  1. Direct verification
  2. Redundant computation
  3. Consistency checks
  4. Data redundancy

# Identifying Correct Outputs

## Direct Verification with Automatic (1)

- Appealing because it eliminates some **human error**
- Fairly **expensive** – requiring more programming
- Verifying outputs is deceptively **hard**
  - One difficulty is getting the **post-conditions** right
- **Not always possible** – we do not always know the correct answer
  - Flow calculations in a stream – the solution is an approximation based on models and guesses; we don't know the correct answers !
  - Probability of being in a particular state in a Petri net – again, we don't know the correct answer

# Identifying Correct Outputs

## Direct Verification with Automatic (2)

- Consider a simple **sort** method
- **Post-condition** : Array is in sorted order

|               |    |    |   |    |
|---------------|----|----|---|----|
| <b>Input</b>  | 8  | 92 | 7 | 14 |
| <b>Output</b> | 1  | 2  | 3 | 4  |
| <b>Output</b> | 92 | 14 | 8 | 7  |

Oops !

Oops !

- **Post-condition** : Array sorted from lowest to highest and contains all the elements from the input array

|               |    |    |    |    |
|---------------|----|----|----|----|
| <b>Input</b>  | 87 | 14 | 14 | 87 |
| <b>Output</b> | 14 | 14 | 14 | 87 |

Oops !

- **Post-condition** : Array sorted from lowest to highest and is a permutation of the input array

# Identifying Correct Outputs

## Direct Verification with Automatic (3)

```
Input : Array A
  Make copy B of A
  Sort A
  // Verify A is a permutation of B
  Check A and B are of the same size
  For each object in A
    Check if object appears in A and B the same number of times
  // Verify A is ordered
  for each index i except the last in A
    Check if A [i] <= A [i+1]
```

- This is almost as **complicated** as the sort method under test !
- We can easily make **mistakes** in the verification methods

# Identifying Correct Outputs

## Other Methods

- Redundant Computation
  - Write **two programs** – check that they produce the same answer
  - Very **expensive** !
- Consistency Checks
  - Check part of the answer
  - Check if a probability is negative
- Data redundancy
  - Check for “**identities**”
  - Testing  $\sin(x) : \sin(a+b) = \sin(a)\cos(b) + \cos(a)\sin(b)$ 
    - Choose a at random
    - Set  $b=x-a$
    - Note failure independence of  $\sin(x)$ ,  $\sin(a)$
    - Repeat process as often as desired; choose different values for a

# Identifying Correct Outputs

## Consistency Checks

- Check part of the answer to see if it makes sense
- Check if a **probability** is negative or larger than one
- Check assertions or invariants
  - No duplicates
  - Cost is greater than zero
  - Internal consistency constraints in databases or objects
- These are only partial solutions
- Consistency Checks do not always apply, but are very useful within those limits

# Systematic-Approach Testing

## Summary

- Give a testing object or system
  - If has standards, choose some for your goal
  - Build usage scenarios with FURPS + Safety + Security
- Give a testing software system
  - Plan test it High level and Low level
- Test Strategic
  - Unit Test & Module Test
    - Driver and stub
    - Testing a Function or a Class or a Module
  - Integration Approach: bottom-up, Top-down, Sandwich
- Functional test (GUI based)
  - Identify Inputs and Functionality
  - If you have activity or state diagram, design with graph test
- Smoke Testing vs. Regression Testing