# CS1632 Software Quality Assurance

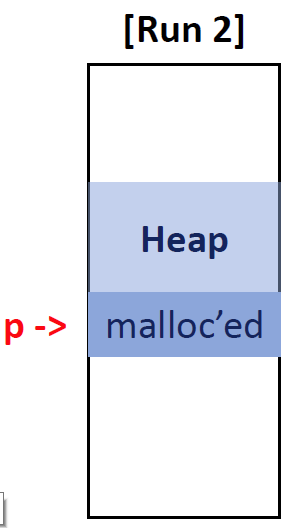
Lec 2 Nondeterminism 非确定性

**非确定性**：程序的输出不由其输入决定。（output在预期外，概率低）

它使得测试困难，出现以下问题：1、surprise defects（测试过程中未发现的缺陷突然出现）；2、unreproducible defects(测试时出现，debug时候不出现)

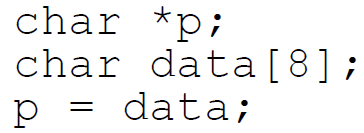
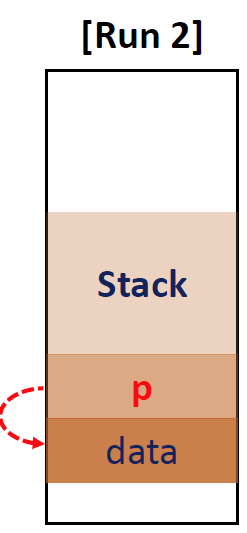
**Nondeterminism by mistake**（本身是缺陷，需要消除）

它源于代码错误：Memory errors；Datarace errors

程序运行时行为未定义，导致调用错误。

* **Memory errors**

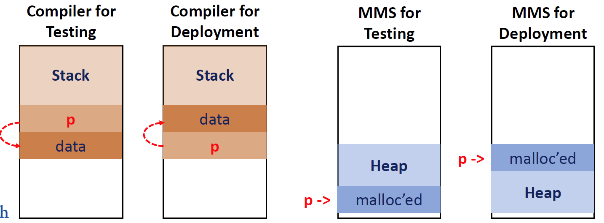
 Malloc会返回随机地址，因为Address Space Layout Randomization (ASLR)，防止黑客猜出内存分布

 栈地址也是随机的，ASLR也应用于栈

指针地址从来不是程序输出的，除非出于调试或诊断目的打印它们

！当有Memory errors时，地址可能会错误地泄漏到程序输出

内存错误在于访问非法内存地址

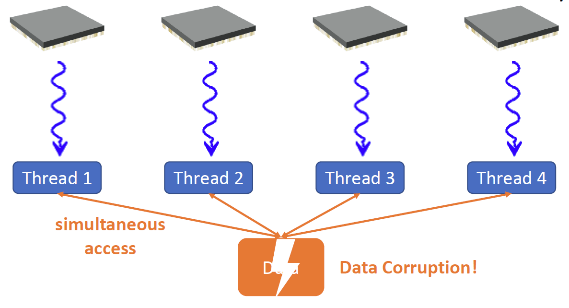
* Buffer overflow: access beyond the bounds of an array
* Dangling pointer: access to already freed memory pointed to by pointer

ASLR是可以关掉的，从而地址固定

但不使用ASLR，仍可能出错，问题是：客户端使用不同版本的编译器；使用不同的runtime memory management system（MMS is nondeterministic on parallel mallocs）

Solution: Google Address Sanitizer (ASAN)

它包含了Address Sanitizer (ASAN); Thread Sanitizer (TSAN); Memory Sanitizer (MSAN)

通过仪器工作,程序中插入额外指令以进行监视, 检测代码在代码执行期间/后报告问题

* **Datarace errors**

A parallel program can use all CPUs

A Parallel Program Runs 1 Thread per CPU Thread: a unit of code that runs on 1 CPU

Simultaneous Access leads to Data Corruption

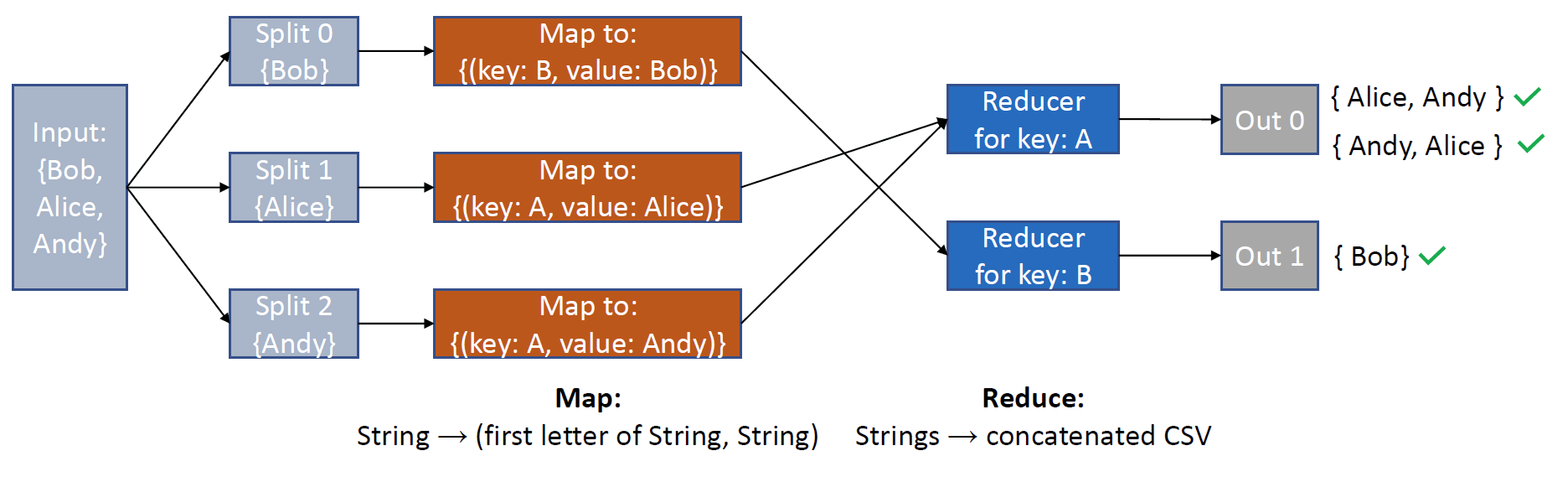
**Nondeterminism by design**（程序员设计，必须处理

• Thread interleaving (examples / solutions)

• Random number generation (examples / solutions)

输出都是正确的，少的约束速度更快

？并行的MapReduce可以提速 – 先分离，根据key分类，再将其约束，最后输出



Reducer以到达顺序连接，这导致非确定性输出，任何non-commutative都有这个属性；确定性的mapping order是被控制的

Nondeterminism is Inherent in Parallelism不确定性是并行性所固有的

？why 1. 通过并行加快运行速度（有时输出都是对的；对于优化问题，非确定性输出可能更好-给定相同运行时间&NP问题）；2.引入随机性（随机事件；加密）

# 真Lec 2 测试理论与术语

Expected behavior vs observed behavior（期望发生与事实发生；defect为这两不一致）

Expected behavior is also known as **requirement**

测试每一个组合是不现实的 相当于测试integer的范围

即便详尽测试，也可能出现问题

* Compiler Issues

编译器有bug是非常少见的，但出现编译错误很正常

* Parallel programming issues

不同线程在独立的cpu上运行，它们同时增加计数导致相互竞争-data race

这就导致每次运行可能都导致不同结果，99%都是正确的、

**synchronized(this)**解决这个问题，锁定代码区域。保证不同过程，相同结果

**Test Coverage**

理想中应该是*test\_coverage = defects\_found / total\_defects*

只能使用估计值 *statement\_coverage = statements\_tested / total\_statements*

因为资源有限，只能最大化statement coverage → 执行所有必需程序行为的测试

**Equivalence Class Partitioning**

将input划分为equivalence classes：例如sqrt方法，可以分为*{nonnegative\_numbers, negative\_numbers}*，前者返回平方根，后者返回NaN

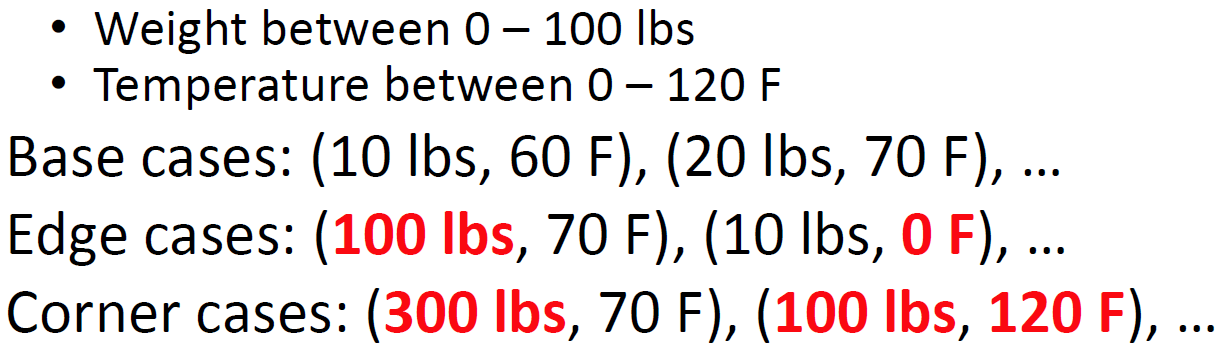
每个input value都属于且仅属于一个equivalence class

Input value可以是string、object

测试每一个等价类：从每个类至少选取一个值，涵盖预期所有行为

**Interior and boundary values**

bug在等价类边界上比在中间更普遍。(由于一错再错Off-by-one Error)

但是有很多边界都是隐形的

Base case: 正常操作时equivalence classes的interior value

Edge case: 正常操作时equivalence classes的boundary value

Corner case: 远远超出正常操作的参数值，或同时发生多个edge case

明确边界来自requirement；隐形边界来自语言、领域知识

Black-box testing: 在不了解内部结构或源代码的情况下进行测试；通过用户界面从用户角度执行测试；可由不知道如何编程的非专业人员执行

Ex. 使用web浏览器测试网站；通过实际玩游戏来测试游戏；针对API端点测试脚本；任何类型的beta测试

White-box testing: 明确了解内部结构和代码库的测试；从开发人员的角度执行测试；精心编制测试输入，以执行特定的代码行；测试可能涉及调用特定方法- 单元测试

Ex. 精心设计输入以练习特定的程序路径- 选择要求中没有但源代码中有的边界值；故意选择导致异常的输入（并观察是否正确处理）

Ex. 从测试脚本显式调用方法- 测试函数是否返回正确的结果；测试instantiate a class是否会创建有效对象

Grey-box testing: 测试时了解一些内部结构和代码库；知识来自部分代码检查或设计文件；从用户的角度执行，但以知识为依据

Ex. 检查代码并注意使用了冒泡排序。然后编写一个涉及大量输入的面向用户的测试。

Ex. 查看web应用程序中的代码，并注意到用户输入未正确清理。然后编写一个面向用户的测试，尝试SQL代码注入。

Ex. 阅读设计文件，注意大量数据通过的网络连接。然后编写一个面向用户的测试，强调这种连接

**Static vs dynamic testing**

Dynamic testing = code is executed; 依靠良好的输入实现良好的覆盖率

* Observed results are compared with expected results

Static testing = code is not executed; 依靠分析代码来发现缺陷

# Lec3 Requirements

* The specifications of the software

需求并非一成不变，需要进行管理和记录

Requirements Verification：我们是否正确构建了软件？（又称testing）

1.从每个测试用例的需求中得出预期行为；2.将预期行为与观察到的行为进行比较

Requirements Validation：我们是否构建了正确的软件？

1.仔细研究需求，确保其有意义；2.采访利益相关者，查看需求是否符合实际需求

3.采访开发者，看看需求在技术上是否可行；4.采访测试人员，看看需求是否可验证

内部问题

* Completeness check：SRS是否涵盖软件的所有方面？

-需求应涵盖系统的所有方面：未涵盖的内容可能会有不同解释，如希望某件事以特定方式发生，则应指定

* Consistency check：SRS是否包含任何逻辑冲突？

-需求必须在内部保持一致，不得相互矛盾

* Ambiguity check：SRS是否有解释空间？

-需求不应当公开解释：例子. 错：当数据库存储无效日期时，应将其设置为默认值。对：当数据库存储无效日期时，应将其设置为1970年1月1日。

坏：系统应在出现错误时及时正常关闭。对：系统应在出错时将所有未决请求存储在命令行上可配置的检查点文件中，然后在5秒内关闭。

外部问题

* Validity check：SRS是否符合用户需求？

-需求必须与利益相关者的需求保持一致：会有一些常见的误解

* Realism check：SRS是否可以切实实施？

-需求必须符合当前技术的实际情况；必须在给定的预算日期内切合实际

* Verifiability check：SRS是否可以进行可行性测试？

-测试需求必须可行：应记录详细确切；应在给定的预算和日期内测试

记录得不够详细：错：计算器子系统应执行所有运算；对：计算器应包括加法、减法、乘法、并在MININT和MAXINT之间除以任意两个整数。

交付日期内不可测：坏：系统应在4137年内处理100 TB的数据集；好：系统应在4小时内处理1 MB数据集。

Requirements should say **WHAT** to do,not HOW to do it! 用户关心的是软件的功能，而不是它是如何发生的

**Functional and Non-Functional Requirements**

Functional Requirements：指定系统的功能行为，系统应在输入Y时执行X。

**Non-Functional Requirements（Quality Attribute）**：指定系统整体质量，而不是特定行为，系统在运行期间应为X。

* 例子. 要求1-应保护系统，防止未经授权的访问；要求2-系统应易于扩展和维护要求3-系统应可移植至其他处理器架构；要求4-系统应在99.999%的时间内可用

Quality Attribute Categories：

* Performance：每秒事务数，响应时间
* Reliability：平均故障间隔时间
* Robustness：系统可以同时处理多少个故障
* Portability：目标系统的数量，或移植需要多长时间
* Usability：训练所需的平均时间
* Accessibility：可以使用系统的人口百分比

质量属性是很难测试的，solution：与利益相关者就可量化问题达成一致

# Lec4

Testing is done by executing a test plan

**Test Plan**: A list of related *test cases* that are run together

**Test case**: test plan中能测试独立行为的最小单元

- Preconditions: 测试前的系统状态；环境/全局变量值；屏幕状态、数据库状态

- Execution Steps

- Postconditions: 测试后系统的预期状态；环境/全局变量已设置；打印到屏幕的输出、发送的网络数据包

Ex. 先决条件：空购物车；执行：点击“购买小工具”；后置条件：购物车显示小工具

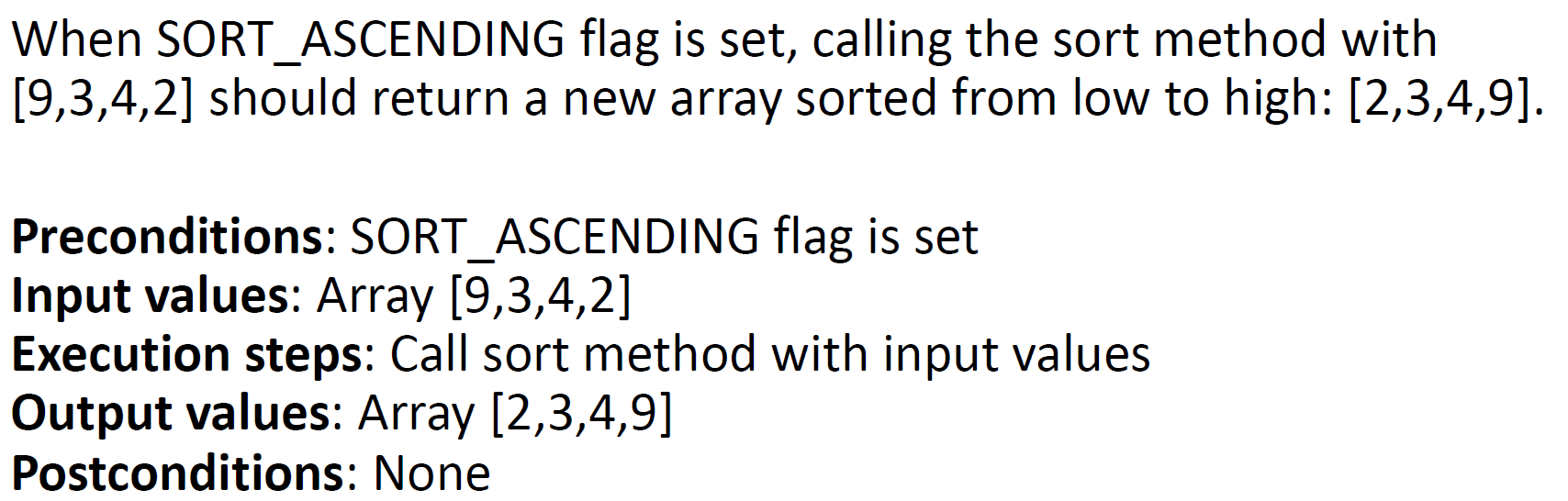
还可以添加**Identifier**: A way to identify test case（通常label）

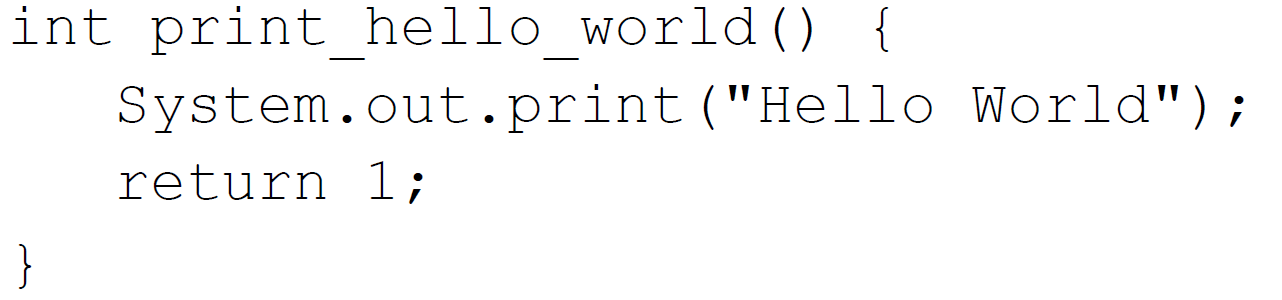
如果做**method unit test**，还需要添加

输入值：作为方法参数传递的值；输出值：方法的预期返回值

输入值和precondition的区别？ -除了argument以外能影响方法的一切东西都是precondition（如全局变量，方法读取的文件内容）

输出值和postcondition的区别？ -除了返回值以外方法影响的一切东西都是postcondition（方法修改的文件内容）





综上，一个test case包含：Identifier, Test Case, Preconditions, Input Values, Execution Steps, Output Values, Postconditions（标识符、测试用例、前置条件、输入值、执行步骤、输出值、后置条件）

Test Plan: List of test cases for a (sub)system 包含很多个test case

A group of test plans make up a **test suite（测试套件）**

regression回归：由看似不相关的增强或缺陷修复引起的先前工作功能的失败

创造Test Suite自上而下开始

将系统细分为功能或子系统，为每个功能创建test plan；对于每个功能，决定要测试的方面；对于每个方面，决定要测试哪些输入或用户交互，为每个输入创建一个test case，命中不同的基础、边缘和角落case以获得良好的测试覆盖率

Test run: Actual execution of a test case / test plan / test suite

The purpose of a test run is to obtain *observed behavior*

Pass or fail after comparing observed behavior with postcondition

Test Run Status：1.PASSED：完成并获得预期结果；2.FAILED：完成但获得意外结果；3.PAUSED：测试在执行中暂停；4.RUNNING：仍在执行中；5.BLOCKED：因为未满足precondition而未完成；6.ERROR：running test本身出现问题

良好的测试用例：除了测试覆盖率，还有独立性和复现性

不可复现说明

1.不完整的preconditions（操作系统状态、数据库状态、文件系统状态、内存状态）

2.不精确的执行步骤

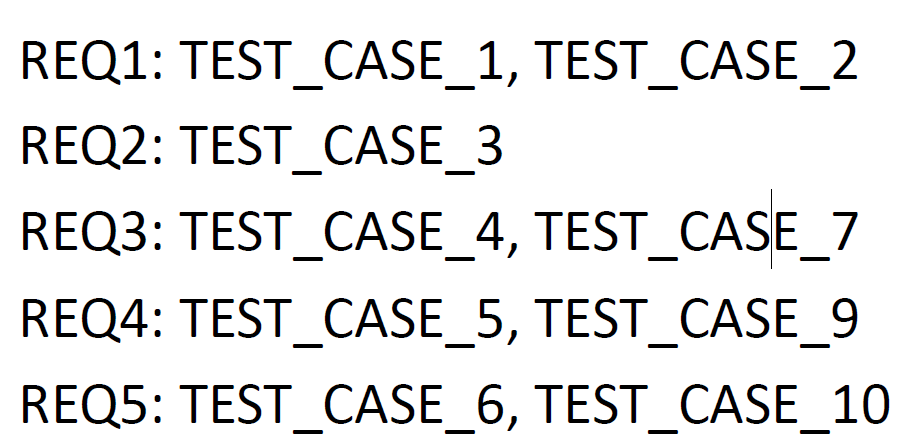
测试的非依赖性

不完整的precondition导致测试具有依赖性（依赖先前的test case来满足部分先决条件）

依赖性会导致1.乱序 2.有选择性

**Traceability Matrix 追溯矩阵**

描述requirement和test case之间关系的表格（多对多）



# Lec5 Defects 缺陷

1. observed behavior ≠ expected behavior, **DEFECT**出现

Defect: 对要求的违反; 开发者向客户支付损失（违反pre-existing需求）

Enhancement：对要求的计划改进；为了增加功能，客户得加钱（（没违反）

Ex1.程序在断电时丢失数据-程序是数据库，这是defect；程序是纸牌游戏，则是enhancement

Ex2.程序在一秒内无响应-程序是实时游戏，这是defect；程序是批处理文件复制工具，则xx

1. Explicit requirement：1- Software Requirements Specification（SRS）所记录的 2- functional and non-functional requirements (quality attributes)

Implicit requirement：SRS中未记录但预期仍需要的；这与程序领域有关

1. Defects是编程错误的结果

它可能来自于外部，但我们只关注内部defects

Defect可能很微小，未经测试的软件一定会带来defect（known & unknown），不一定影响使用

1. 严重缺陷（Faulty data错误数据；system crashes系统崩溃；extreme resource usage极端的资源使用；issues with core features核心功能问题）
2. Reporting defects 写bug report：
   1. SUMMARY：简明描述问题，通常为一句话，如 “在列表添加两个节点后，CPU固定在100%”
   2. DESCRIPTION：测试人员发现的所有内容的详细描述，问题的细节
   3. REPRODUCTION STEPS：Preconditions + Steps to Reproduce Defect

首先，列出先决条件；接下来，列举重现缺陷所需的*步骤*；通常最好是over-specifying出错

* 1. EXPECTED BEHAVIOR & OBSERVED BEHAVIOR；
  2. IMPACT：对不同利益相关者的影响
  3. SEVERITY：问题有多严重？（CRITICAL；MAJOR；NORMAL；MINOR；TRIVIAL）

问题发生时有多严重？多久发生一次？有解决办法吗？

* 1. PRIORITY：应首先处理哪些defect的顺序
  2. NOTES：有助于理解和解决问题的技术和详细说明。它包含：Stack traces堆栈跟踪；Log file excerpts日志文件摘录；Environment；可能有助于开发人员修复此缺陷的任何内容

1. Tracking Defects

* 一旦报告缺陷，就需要对其跟踪，确保及时修复，验证修复可纠正缺陷且不会导致regression
* 必须以系统的方式完成。在不同的解决阶段，通常会出现数百个错误，通常在bug tracking system的帮助下完成
* Defect应有以下信息以便track：Identifier标识符: usually numbered, not named；Source来源: associated test case；Version of software found(fixed)
* Lifecycle of a defect：Discovery发现；Recording记录；Triage鉴别分类；sub-triage；Fixed修复；Verified核实
* 在上述步骤，triage是最重要的，又被称为defect review：1.Validity of defect； 2. Final severity； 3. Final severity；4.将缺陷分配给特定开发人员
* Sub-triage适用于较大的项目：先对其进行系统级分类，筛选出非缺陷和重复缺陷，将缺陷分配给子系统；子系统内缺陷的优先级划分，再分配给开发人员

1. Verification：修复应当实际上解决报告的缺陷，而不引起其他问题（regression testing）

如果修复不正确，则迭代回修复阶段；如果修复正确，关闭bug report

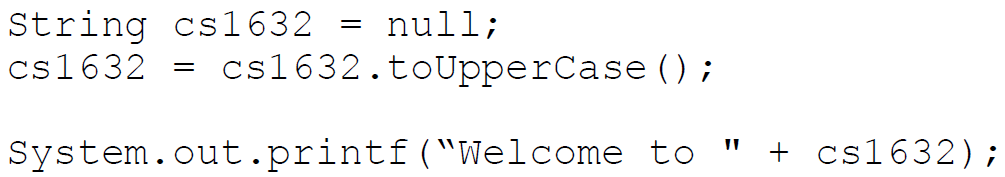
Add test case for bug to test suite (可选)

# Lec6 Breaking software

1. – 软件的happy path：A case where user inputs typical values; expected usage

* Software breaks on the unexpected cases: 1. Corner cases or edge cases; 2. subsystems go down; 3. 软件缺乏资源（内存/带宽/CPU周期）; 4. 恶意用户故意试图破坏软件

1. – Logic Errors: 程序的逻辑有错误

* OFF-by-one Errors: 某个unit指定的值不正确 （它是逻辑错误的子集，因为code逻辑上有问题）
* Null pointer error: 程序取消对空指针的引用

数据库或网络连接失败 & 在填写表单中将文本框留空将导致空指针

* Rounding / Floating Point Errors: 由于浮点数不精确导致的错误

精度误差往往会累积，严重影响最终结果

* Integration Errors集成错误: 发生在（子）系统边界之间的错误
* Missing Data Errors:

示例错误结果：Index 3 out of bounds for length 0

* Bad Data Errors: 数据格式不正确或无效（如0/0）
* Display Errors: 数据正确，但未正确显示。
* I/O Errors: 磁盘、网络或其他I/O的意外状态（报错找不到）
* Configuration error配置: 系统没有配置好来正常工作（学校网站报错）
* 除此以外，还有

Data type errors：由于不正确的隐式/显式数据类型转换而产生的错误

Permission errors：没有访问所需资源（文件、数据库表等）的权限

Version mismatch errors：库版本与软件的预期版本不同

Distributed system errors：在分布式系统的不同部分之间通信时出错(Ex. 客户端/服务器之间的数据编组/解编组错误)

Interface errors：由于开发人员误解API的行为而导致的错误

# Lec7 自动化vs手动测试

1. Manual Testing：我们至今为止所做的，一个人类执行test plan
   1. 优点

+可由非熟练员工执行

+不需要测试自动化工具

+灵活：人类可以调整测试用例以适应软件中的小变化

+宽泛：即使在测试用例中没有明确说明，人们也可以注意到软件的问题

* 1. 缺点

-时间和资源密集型

-再现性问题（例如，如果测试对时间敏感，人类不擅长）

-容易犯人为错误

-仅限于通过用户界面进行测试（例如，无法进行单元测试）

1. Automated Tests：计算机运行测试（能跑很多次）

2.1 优点

+一旦设置，运行速度更快，成本更低

+更擅长精确地再现步骤

+不可能有人为错误

+可以使用JUnit之类的工具进行单元测试

* 1. 缺点

-提前构建测试基础架构的时间

-需要熟练的员工来构建测试基础设施

-需要测试工具（并学习它们）

-脆弱性：软件中的微小更改可能导致自动测试用例失败（维护成本高昂）

-狭窄性：只测试显式脚本化的内容

**2.3 脆弱性：**机器不像人那样灵活，微小更改也会导致测试失败(如gui中的细小变动)

- 软件一旦更新，test也得随之更新，test code 成为code base的一部分

- 你花在调试测试上的时间就和你花在调试应用上的时间一样多！

**2.4 狭隘性：**

- Automated test cases only check “known unknowns”(test case指定我们应当检查的)

- do not check “unknown unknowns”

- 手动测试可以检查“unknown unknowns”, 人类在测试时会注意到一些细节

1. Mixture: 同时使用手动和自动测试: 自动化测试完成了大部分繁重的工作，手动测试以确保自动化测试不会遗漏任何内容

* 有时完全自动化是可能的，特别是如果软件没有UI，大型软件公司往往拥有足够全面的自动化

1. Automated Test Cases与Manual Test Cases相同
2. 自动化测试可以是black box和white box
   1. Blackbox testing：使用应用程序界面在系统级别进行测试

- 脚本使用各种命令行参数调用应用程序，与应用程序交互，提供用户输入并验证程序输出

-scripts for GUI apps: Selenium

* 1. Whitebox testing: unit testing

-脚本使用测试参数直接调用应用程序源代码中的方法

-脚本通过检查返回值和/或后置条件来验证每个方法

-用Junit

# Lec8 Unit Testing

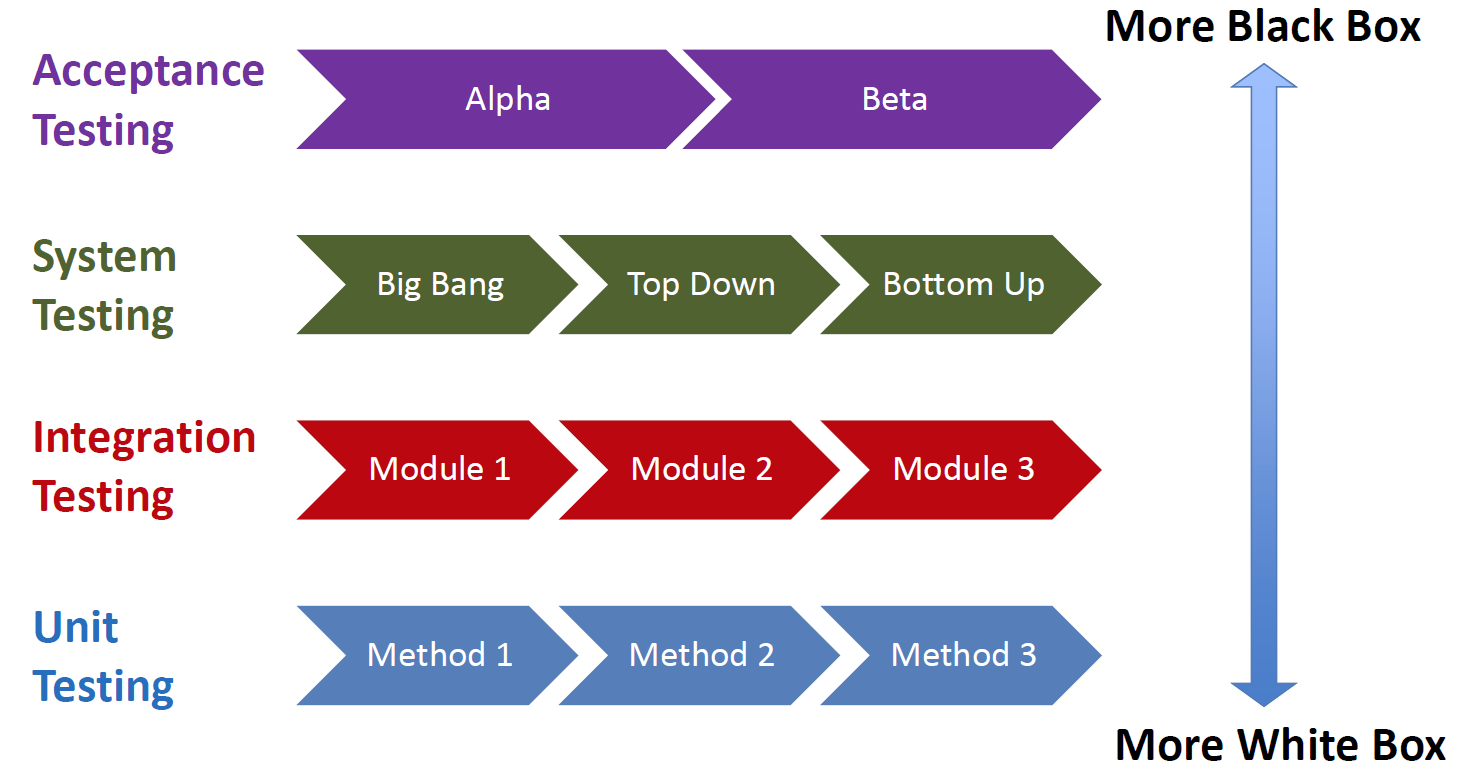
1. 测试代码中最小的单元“unit” - Functions, methods, or classes；保证unit正确工作

* 白盒测试，直接调用functions or methods
* 不能保证unit一起工作时不出问题

1. 2.1 Unit Testing : Testing smallest unit of SW (typically a method)

2.2 Integration Testing : 将unit集成到module后测试（a group of classes or package）

2.3 System Testing : 集成所有模块后的End to end testing

- Big Bang : 集成所有模块后立即测试

- Top Down: 自上而下添加模块进行增量测试(用stubs代替未添加的leaf modules模拟这些模块)

- Bottom Up: 自下而上添加模块进行增量测试(用驱动程序代替尚未添加的root模块调用leaf模块)

(为什么是增量测试？易于定位导致缺陷的模块)

2.4 (User) Acceptance Testing: 确定软件对于用户是可接受的

- Alpha 测试：发布给选定的一小群用户

可以是一组精选的具有高技术的客户，可以在内部，甚至是同一个开发团队(dogfooding)

目标：测试并最终确定软件的主要功能

* + Beta测试：向更广泛的用户发布

Closed Beta: 私人测试版，仅通过邀请

Open Beta: 公开测试版，任何希望参与的人都可以参加

目标：确保各种平台和环境的稳定性和安全性

1. 单元测试的例子：lec9 p6

Unit testing一般由开发者实施

* 1. why unit testing?
* 早期发现的问题：无需等到系统构建完成
* 更快的周转时间：开发人员可以立即调试。无需等待测试人员运行测试/归档错误报告/分配错误
* 开发人员非常了解代码，并且知道在哪里可以找到defect
* “Living documentation” : 单元测试可以被视为软件预期行为的文档，它是living的，因为测试是通过对软件运行来定期验证的
* Unit tests in sum total form a test suite：test suite作为回归测试运行（从具有非局部影响的更改中查找缺陷，可以发现由于其他单元的更改而导致的缺陷）

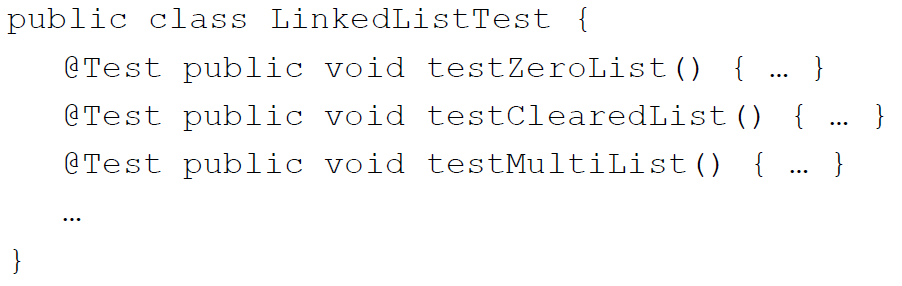
3.2 unit test的组成

- Preconditions: 代码设置（初始化变量与数据结构）

- Execution Steps:对单元测试方法的一个或多个调用

- Postconditions : 检查后置条件是否满足

Ex. LinkedList.equals () method P10

1. A JUnit @Test Method is a Test Case

* Each @Test JUnit method is a test case
* Each JUnit class is a test plan
* Collection of JUnit classes is a test suite

1. Assertion (postcondition check)

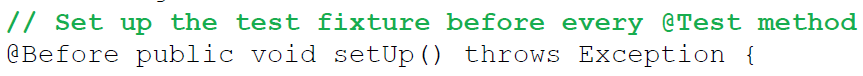
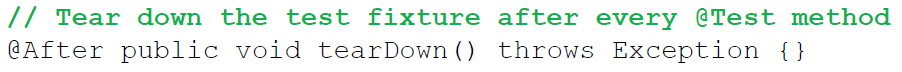
Should assert EXPECTED BEHAVIOR == OBSERVED BEHAVIOR

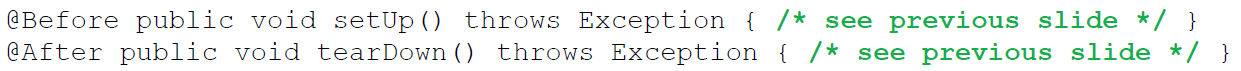
* 1. Junit assertion(具体的函数功能)
     1. assertSame (Object expected, Object actual): reference comparison
     2. assertThat (T actual, Matcher<T> matcher): a catch all assertion

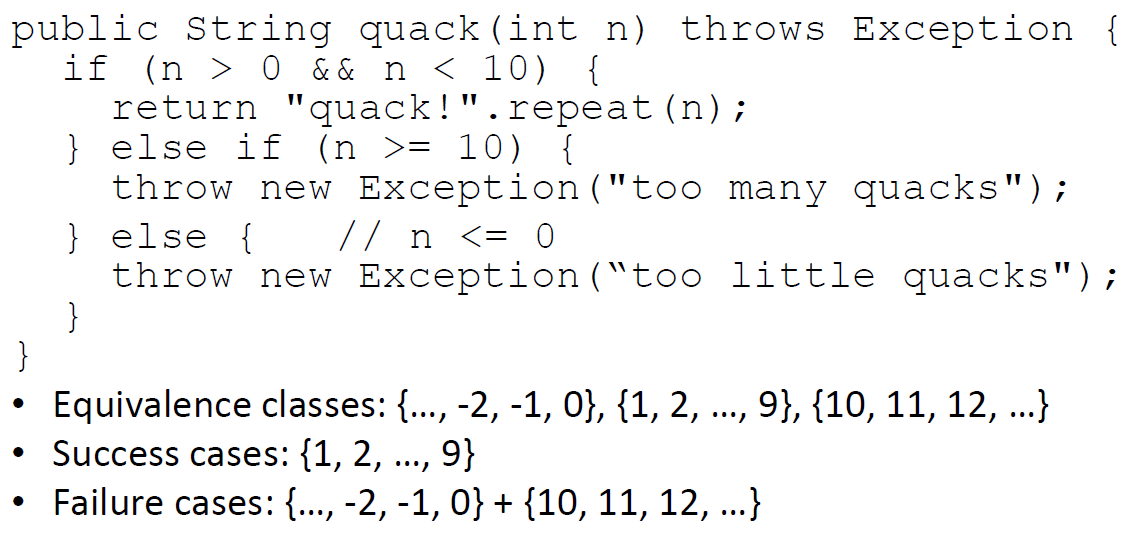
E.g. assertThat (“CS1632", anyOf (is(“cs1632"), containsString (“CS”)));

* + 1. fail(): assertion that always fails

1. Test Fixture = Baseline Preconditions （一个固定状态，用于baseline precondition）

* Test case in test plan通常需要一个通用的基线先决条件
* 内存填充了一组固定的对象; 数据库填充了一组固定的条目; 硬件设备重新初始化为固定状态
* @Before: 方法执行于每次 @Test method 前
* @After: 方法执行于每次 @Test method 后





1. 要测试方法参数的哪些值？

* 每个等价类；internal & boundary value
* Success & failure cases

1. Public vs. Private Methods

只测试公共方法；公共和私有方法都测试

* 1. 只测试public methods

私有方法作为公共方法的一部分测试；

私有方法频繁改动，它们不是public object interface的一部分，每次测试都得改框架

由于语言、框架问题，私有方法难以测试

* 1. 测试每个方法

公共私有是任意的，它们都是code的一个unit；

Unit testing means testing at the lowest level; 测试到私有方法更贴近

# Lec9 Unit Testing2

先进的单元测试技术

1. Removing Class Dependencies: Test Doubles 测试替代技术
   1. Fake Objects 用来替代 real objects来测试target class method （将外部类分割开来，模拟）

这些是目标类方法引用的外部类的对象

* 1. Goal: 不执行real object中的代码
* 意味着可以在没有实现外部类的情况下测试目标类
* 意味着如果发现缺陷，将其定位到目标类内
* 意味着外部类的任何变化都不会影响测试

1.3 警告：double应该看起来像target class的real thing；即便double不执行外部类的代码，也应当模拟真实对象

1.4 ex1. Doubled database object: 在没有安装数据库的情况下进行测试。Double实际上没有连接到数据库，它返回预先确定的用于测试的数据库条目。

Ex2. Doubled file object: 用于模拟真实文件难以实现的故障。Double实际上并不从硬盘读取文件，它模拟在真实硬盘中难以触发的文件读取失败。

Ex.3 Doubled RandomNumberGenerator ：用于可重复测试。Double 实际上不会生成随机数，它返回预先确定的数字以使软件具有确定性。

1.5 Double外部类（并非测试类）

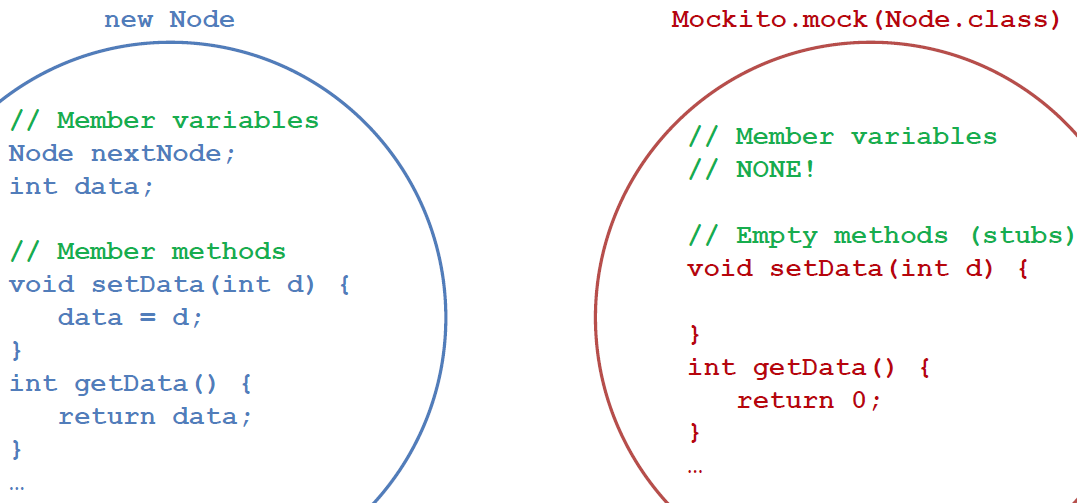
不要将test class double，一旦这样，你测试的是fake object而不是真实对象

测试类所依赖的外部类的Double对象

1.6 例子

只想要测试linkedlist，而不想测试Node(Node中的缺陷可能导致测试失败，并非真正的unit test)，

用mockito.mock测试替代，不要将linkedlist double因为那是就是测试的target class



1. Removing Class Dependencies: **Stubs**

Test doubles are **fake objects** and stubs are fake methods for fake objects(两者相生)

* 1. 默认的stubs都是空方法

Return type void : Does nothing

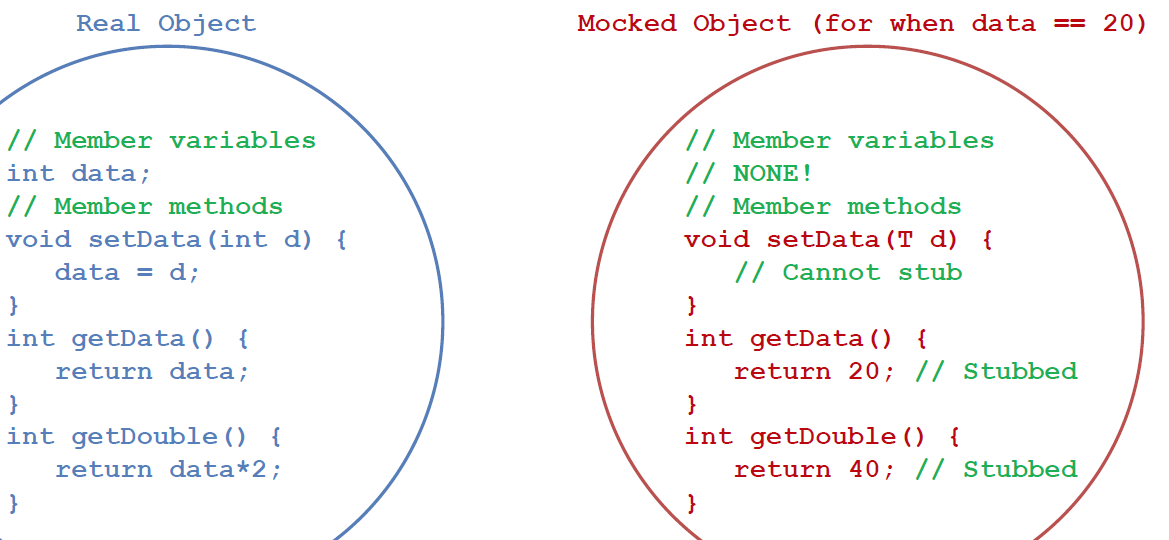
Return type int : returns 0

Return type boolean : returns false

Return type Object : returns null

原始类的代码永远不会执行，default stubs可以修改，仍然是空的，但是可以返回不同值

* 1. mocked objects have no state

stub模拟get方法

1. Behavior Verification: **Mocks 行为验证**
   1. State Verification vs. Behavior Verification

* State Verification: 测试程序的状态

状态是否因为方法调用而正确更改，完成于对postconditions的assertions而完成（我们所做的）

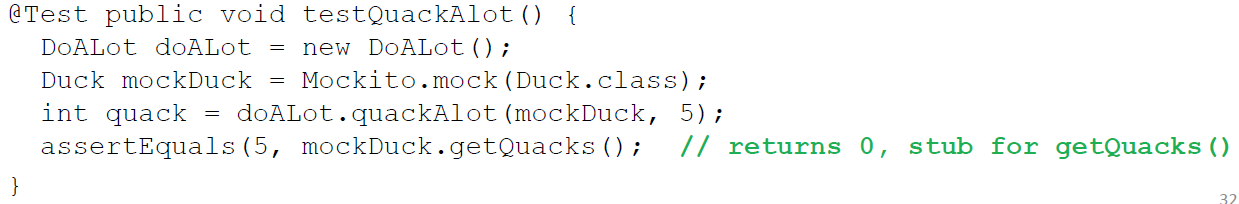
* Behavior Verification : 测试代码的行为

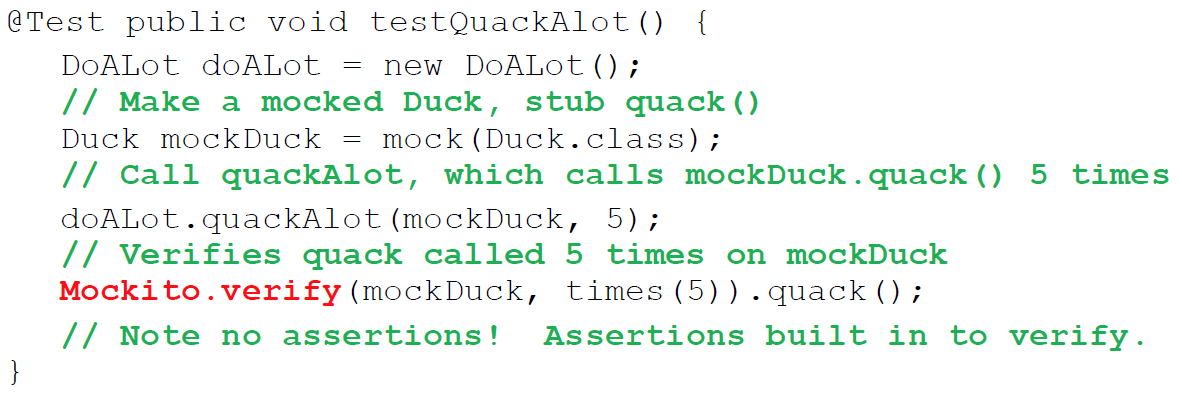
某些方法是否已被调用一定次数；是否使用正确的形参调用了方法

完成于Mockito的verify

* 1. Why behavior verification?
* 在最后，我们所重视的是state而不是behavior。State Verification以下用处：方法是否返回正确值；方法是否正确更新heap object；方式是否输出正确的消息（我们很少重视内部调用的方法）
* 尽量做state verification
* 但是如果想要验证mocked object的状态？

模拟对象没有state to begin with> 需要behavior verification

* 1. When State Verification does not Work

使用behavior verification

Mockito.verify 不直接检验 mockDuck是否有正确的state

1. Mock模拟：A test double which uses behavior verification
   1. 大多数框架对doubles和mocks有同样的框架

Mockito.mock is used to create both doubles and mocks；对象如果用了behavior verification, 它是个mock; 对象不用behavior verification, 它是个double

* But technically, a mock is a specific kind of test double.

4.2 更多junit / mockito例子

# Unit Testing Summary

好的单元测试应有以下标准：

1. 每次运行都可重复

测试应该一直都只有一个结果，否则: 如果失败，很难重现缺陷以进行调试；如果通过，不能保证缺陷以后不会重新出现

为了确保运行可以复现：preconditions必须准确和完整；execution steps不能有随机因素（测试本身无随机性- a random input value；程序内部无随机性-掷骰游戏）

1. 独立于其他测试 ：测试不应该依赖于其他测试来运行。

- run a subset of tests in a test suite

- 以不同的顺序运行（如并行）

- 其他测试失败导致连锁反应

3. 已本地化(tests only the unit) ：模拟外部类，并存根方法

4. 一次测试一个test case：不要在单个测试中测试不同的test case, 理由：

- 如果失败(assertion fires)，剩余的也不会测试；失败时很难判断哪个失败

这意味着我们每次测试只能调用我们正在测试的方法，除非调用其他方法来设置pre/检查post

Junit不是唯一的单元测试框架

Unit Testing != System Testing （exe1是系统测试；exe2是单元测试）

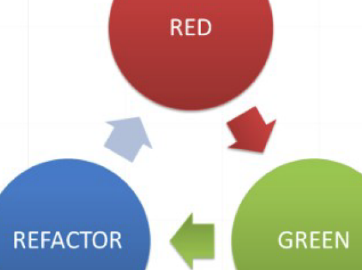
一般测试流程以上两个都得检测

# Lec10 TDD Test driven Development测试驱动开发

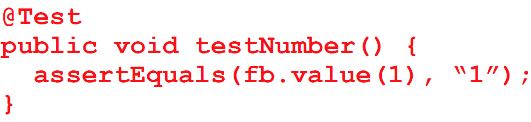
1. TDD是一种软件测试方法：在编写代码之前编写测试；只编写经过测试的代码；非常短的周转周期；尽早并经常refactor

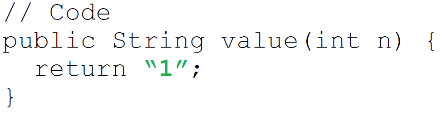
* 这是一种面向开发者的开发方法
* 这里指的test是unit test：UTDD: Unit Test Driven Development

1. The Red Green Refactor Loop

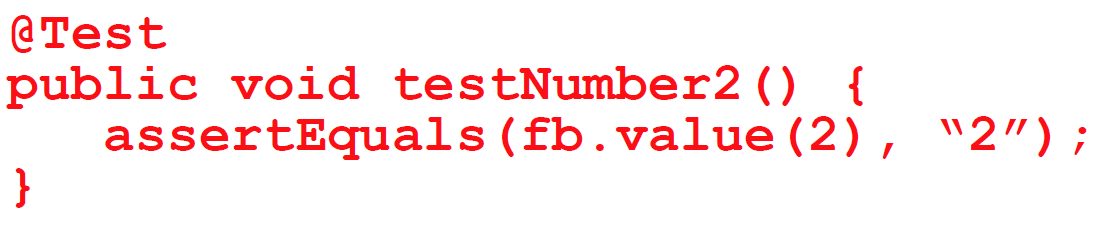
* RED: 为新功能编写测试，测试应当立即失败
* GREEN：测试通过
* Refactor：在不改变功能的情况下改进代码（代码不完美）

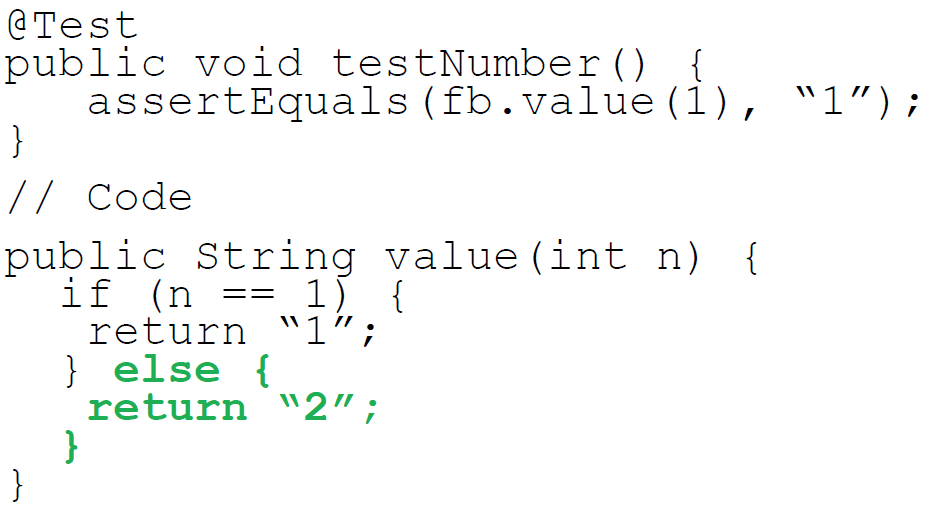
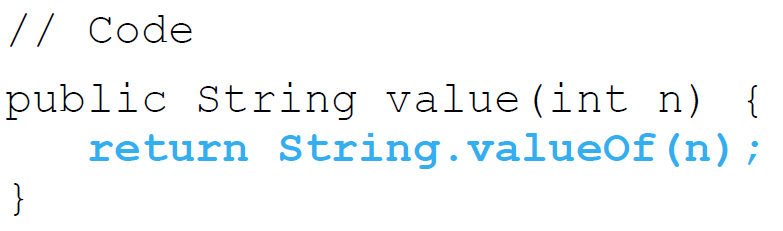
EX. Fizzbuzzin1

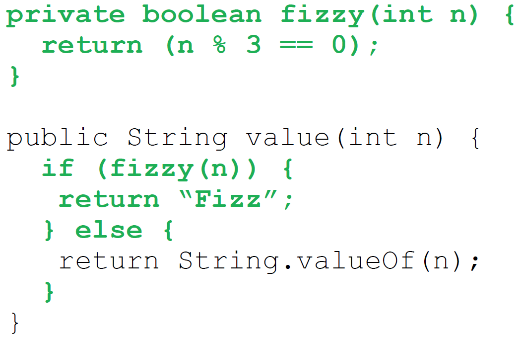


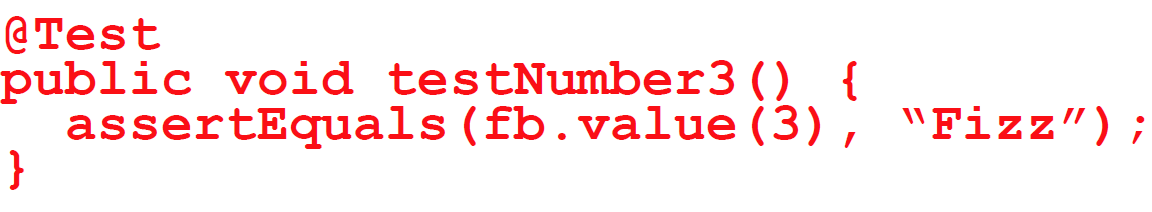


EX. Fizzbuzzin2





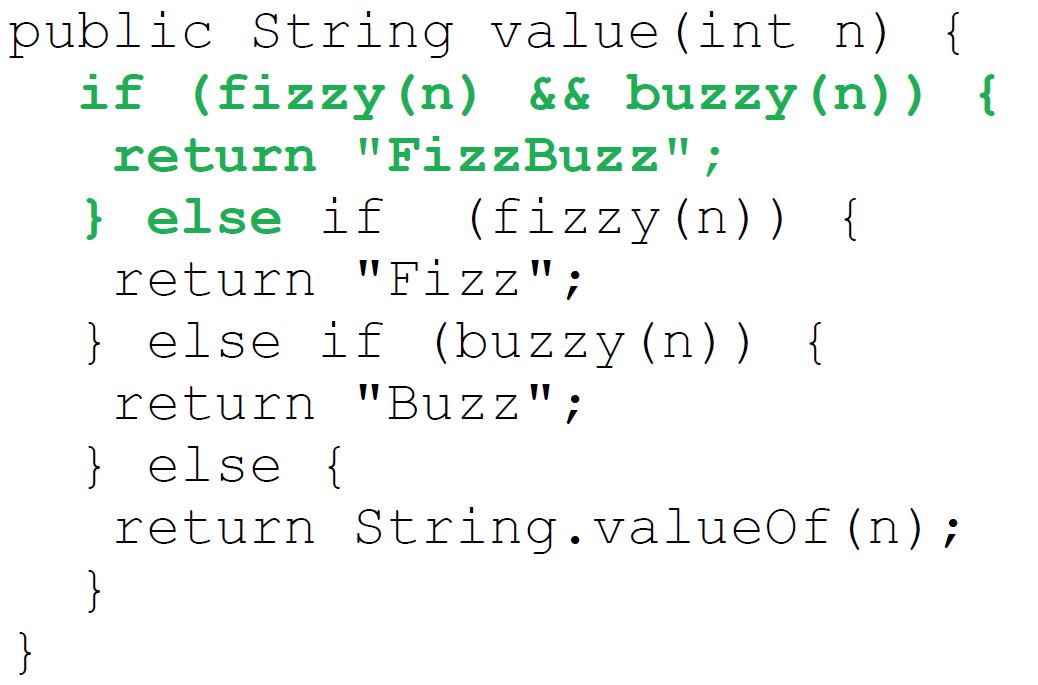


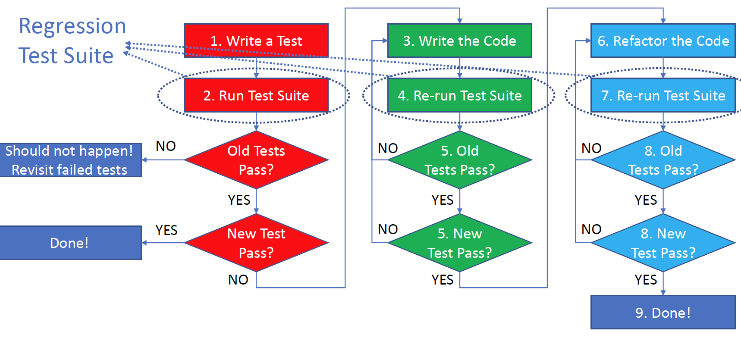
Ex. Fizzbuzzin3

EX. Fizzbuzzin5

Test加5，操作方式与3同理，创建私有方法，写循环

EX. Fizzbuzzin15



1. 建议
   1. 开发由测试驱动：需求推动测试，测试推动开发 ->编程与需求紧密相关
   2. 100%测试覆盖率：测试在编程前，从而保证defect能被立即抓取
   3. Regression test回溯测试的保证：新的旧的都要通过，以保证开发过程中不到退
   4. 小增量开发：一次只写通过下次测试的代码，每次测试只需测试小增量
   5. QA纳入开发周期：软件一边循环一遍重构，循环结束时就无需更改
2. Red Green Refactor Loop**流程图**
   1. 一次编写一个小测试，保持紧密周转周期
   2. 运行整个test suite：注意regression error；对所有组件进行连续测试
   3. 不要多写代码，确保写的代码都被测试覆盖
   4. 每次迭代都要refactoer：green确定正确，refactor改进，搞完之后regression test
   5. Fast and Independent

每次代码更改运行整个test suite，影响开发时间

* 使Unit test单独快速运行：对于易延迟的组件（databases, files, network I/O）使用test doubles和stub
* Independent:测试不依赖于其他测试的结果：仅选择与修改代码相关的单元测试；拆分测试并行运行

1. Sum
   1. benefit: 编程与需求紧密相连；100%覆盖率；以前test case失败时代码不回归；小增量；全周期
   2. drawback: - 测试成为项目开销的一部分：维护成本与测试时间

- 难以进行大量、复杂的设计：TDD相对短视

- 关注unit test错失一些别的：mock缺乏integration test；关注代码而不关注用户体验

5.3 TDD是测试优先开发，关注于unit testing，其他test-first不是

# Lec10补充BDD Behavior Driven Development行为驱动开发

行为驱动开发BDD：它是一种TDD，但是更适应用户需求

1. Agile Software Development: 快速适应不断变化的需求
   1. Agile: 一个能够适应需求频繁变化的开发过程

* Stresses adapting to user needs quickly vs. negotiating a contract
* Stresses efficient communication vs. comprehensive specification
* Stresses iterative design vs. rigid plan
  1. some practice
* Continuous Delivery (CD): 频繁交付软件以获取用户反馈
* Test Driven Development (TDD): 允许通过持续测试实现持续交付
* Behavior Driven Development (BDD) : 一种更适合适应用户需求的TDD

1. 维护需求是一种负担
   1. Software Requirements Specification (SRS): 对于lawyer而言，SRS是一份合同；对于开发者而言，SRS是一种规范

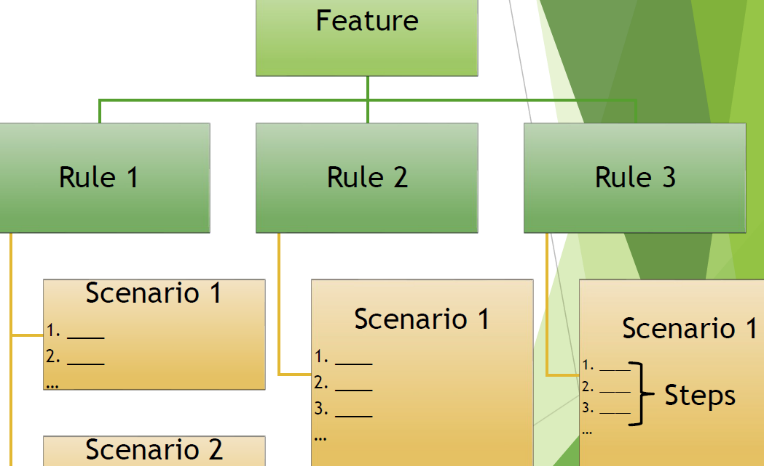
现在srs逐渐繁琐，用户看不懂

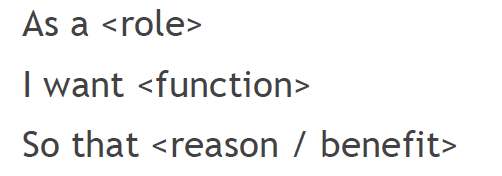
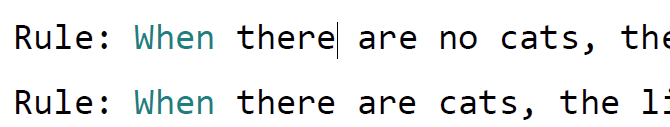
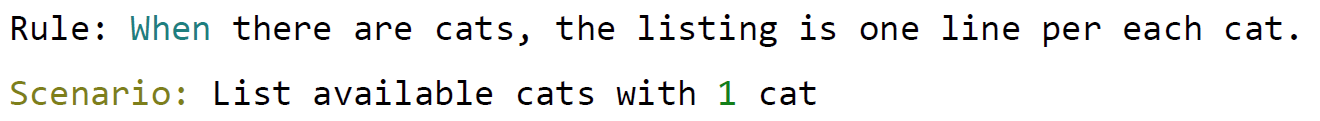
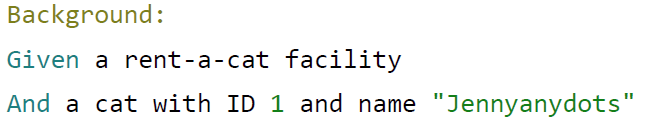
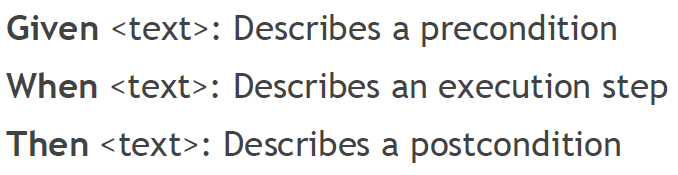
* 1. BDD：behavior是requirement和test的一体化

Software is described in terms of behaviors；Behaviors are “executable”

以代码为中心的描述很抽象，程序员才懂；行为驱动的描述使用者也看得懂（p9）

* 1. BDD解决TDD存在的问题
  2. 行为描述必须满足两个标准：
* 像英语，便于理解
* 有结构，使测试可以自动化
* 有俩，Gherkin with Cucumber framework & JBehave; 主要学习Gherkin with Cucumber but

1. Gherkin : 描述行为的特定语言
   1. 结构：feature > rule > scenario >step
   2. Feature: <text>衍生到下一个keyword的多行文本

* Text 是对特征的一行描述
  1. Rule: 多个规则可以跟在 Feature 关键字之后
  2. Scenario: follow Rule
  3. Step: One or more steps describe a scenario
  4. Background(optional): 多个给定步骤，跟在Feature之后

1. Sum
   1. Cucumber 框架将使用正则表达式匹配，使其在java中实现
   2. Pro: 采纳用户反馈，使需求易于发展（用户满意度更高）
   3. Con: 为模棱两可的要求留下空间 (specification by example)
   4. Good for user facing functionality; 不利于后端和安全开发