# Lec11 Testable Code

1. Segment code 段码

方法应当执行定义良好的功能（不要把几个函数功能写在一块，分开来写得清楚）

1. DRY (Don’t repeat yourself)

不要复制粘贴代码；不要多个方法有近似的功能

Disadvantage:两倍代码空间；两倍的测试维护量；错误也可能是两倍

* 对于重复的代码，创建一个新方法并调用它
* 对于两个（或多个）相似的方法，将它们合并为一个
* 如果两个代码在功能上相似只是类型不同怎么办？ polymorphism（子类，继承）

P13

* generic classes and methods：参数化类型的类和方法

如Java ArrayList<Integer> 由 Integer 类型参数化

1. Move TUFs out of TUCs

No Test-Unfriendly Features (TUFs) inside Test-Unfriendly Constructs (TUCs)

3.1 Feature (用stubs来伪造)

- ？因为设置功能需要很长时间才能正常工作，测试时间过长（I/O），导致不必要副作用

- 例子：打印到console; 从数据库读取、写入；读/写文件系统；访问不同的程序或系统；访问网络

3.2 Constructs: 使用stubs或overriding难以伪造的方法

- stub: 使用mockito替换模拟对象中的方法

- override：覆盖fake class中的方法，该子类是真实类

- 例子：无法覆盖：对象构造函数析构函数；private；final；static

3.3 总而言之，就是不要把你试图fake的code放进很难fake的方法

1. Make it easy to satisfy preconditions(容易满足先决条件)

4.1 尽量不要依赖外部数据 – global variables; 从全局数据结构提取的值；从文件或数据库读取的值；任何没有作为参数传入的值；side-effect

- 例：依赖外部嵌入，开发人员可能不小心改变precon；正确方式应该是值通过参数传递，即pure function

4.2 如果一定需要外部数据：尽量用重构的；剩下的数据进行隔离 -保持尽可能的纯净

1. Make it easy to reproduce

易于重现；不依赖随机数据

1. Make it easy to localize（dependency injection）

Passing a dependent object as argument（模拟对象使测试更容易）

6.1 改进代码从何开始？建立一个测试基础设施以确保没有任何问题

- 从编写固定测试开始：为确定existing behavior现有行为而进行的测试，通常是单元测试（seam）

- 现有行为可能与预期行为不同，修改之前需要确定

- 在旧代码中寻找seam：两个代码模块相遇的地方；不需要修改源代码。Seam 是一个可以注入假对象以本地化单元测试的地方

- 为什么不可以修改源代码？因为固定测试需要原样测试，修改源代码可能改变行为

- 重构遗留代码STEPS：

1. 为重构类编写固定测试

2. 利用seam启动单元测试

3. 重构方法

4. 运行固定测试确保现有行为没有改变

5. 对每个要重构的方法重复步骤 2 和 3

# Lec12使用 Selenium 测试 Web 的系统

1. System testing: 将整个系统作为一个整体进行测试(希望自动化系统测试)
   1. CLI的自动化测试：创建一个“输入脚本”并重定向到标准输入，将标准输出重定向到文件并与预期输出进行比较

GUI的自动化测试：GUI不都是CLI，步骤与observed behavior vs. expected behavior一样

* 1. GUI APPs和基于文本的apps很像：gui有输出文本，只是将其呈现为最终用户的图形表示

例子：请求url时从web获取html文本，经由web浏览器呈现为图形元素；xml从移动应用服务器获取，由手机操作系统呈现为图形元素

因此，测试与txt一致（假设大操作背景没问题）

* 1. 如果直接将其当作textbasedapp会导致：1. Tests are unreadable：不确定试图检查html大量元素中的哪个 2. Tests are fragile：一点更改也可能破坏测试 3. 不改变功能的JavaScript代码更改（例如注释）可能会破坏测试

1. Solution: **Web Testing Frameworks** web测试框架
   1. 将html视为纯文本过于简单；它应当形成层次结构：有内容有属性

* 提供 API 将 HTML 解析为树状数据结构并进行搜索
* 提供 API 来模拟点击或输入等事件，测试 JavaScript 代码
* 自动化测试脚本是通过调用API来模拟用户交互编写的
* 需要为每种类型的 Web 浏览器实现 Web 驱动程序
  1. Web Driver: 用于指示 Web 浏览器行为的界面, 主要用于允许编写自动化测试；Web 浏览器编写者为他们自己的 Web 浏览器实现驱动程序
  2. 当使用web测试框架时

1. Tests are readable: 从 Web 驱动程序 API 调用中可以明显看出正在测试哪些 HTML 元素
2. Tests are not fragile: html中的细微变化不会影响html解析树; 更改页面的不相关部分不会破坏测试
3. JavaScript 代码经过功能测试: 框架允许模拟触发 JavaScript 函数调用的事件; 框架允许直接调用 JavaScript 函数作为脚本的一部分。不改变功能的 JavaScript 代码更改不会破坏测试
4. Selenium：一个开源的web测试框架

Selenium = WebDriver + Grid + IDE （WebDriver：适用于 Chrome 和 Firefox 的驱动程序；Grid：并行运行 Selenium 测试的网格计算服务器）

# LEC13 PERFORMANCE TESTING 性能测试

1. 定义

* Merriam Webster: 执行的能力
* Cambridge：完成工作的能力如何
* Macmillan：速度/效率如何

对于QA, performance是non-functional requirement

* 狭义上是程序的速度；广义上是程序的有效性-广义是主要讨论的

速度很难定义web browser & web server（需要多个指标来量化）

Performance Indicators（被测系统性能的定量分析）

-狭义：response time(按下按钮需要多久)；吞吐量(系统一次可以处理多少用户)

-广义：availability(系统能正常运行多久)；utilization(程序使用多少内存，多少能量)；

1. KPI：关键绩效指标（要选择真正重要的KPI）

Performance target：KPI应达到的量化指标

Performance threshold(性能阈值)：KPI应达到的最低值

Ex. KPI：吞吐量

性能目标：每秒100个用户请求

性能阈值：每10个用户请求

1. Performance Indicators: Categories
   1. **Service Oriented**（衡量服务质量QoS）

两个子类别：

**Response Time**：系统响应用户请求的速度有多快（加载网页需要多久）

**Availability**：用户可以访问系统服务的时间百分比是多少（一年运行多少天）

* 1. **Efficiency Oriented**（衡量系统利用计算资源的程度）

两个子类别：

**Utilization**：软件使用了多少计算资源（为一个网页提供服务需要多少CPU block？）

**Throughput**：在给定的时间内可以处理多少个请求（一个web服务器服务每秒可以访问多少网页）

* 1. 两者对比
* Service oriented测量当前系统的QoS

用户是否对response time和availability满意

* Efficiency oriented衡量实现QoS所需的资源

CPU utilization: CPU需要达到目标响应时间

Memory utilization: 保证可用性而不崩溃所需的内存

Server throughput: 服务器需要一致地实现目标响应时间

Server throughput: 服务器需要在高需求期间保证可用性

* Efficiency oriented指出了软件中的问题领域

给定当前的资源集，软件的哪个方面会降低QoS？

1. 测试面向服务的性能指标

**LEC14 Property-Based Testing基于属性的测试**

什么是测试？对照观察到的行为检查预期行为

1. Stochastic Testing随机测试

* 使用随机生成的输入值进行测试
* 并不测试所有值，测试大量随机值获得良好覆盖率
* 又被称为monkey testing
* 测试人员应充分考虑如何生成输入值；值源于分布，分布影响覆盖率；测试人员应选择最有可能发现缺陷的分布

1. Test Oracle Problem

* 我们需要一个预言家告诉我们预期的输出值是什么（经过测试的是观测输出值）
  1. Property-Based Testing: 使用output的属性进行测试
* Property: 输出值中始终保持正确的东西；又称为invariants
* 不直接输出output
* 想法: 编写一次property, 并对所有（至少一大组）输出值进行测试
* Invariant只检查行为的一个子集

Ex. 以sort()方法为例，invariant是：1. 输出数组的大小与输入数组的大小相同；2. 输入数组中的每个元素都在输出数组中；3. 输出数组中没有不在输入数组中的元素；4. 输出数组中的值总是在增加或保持不变；5. 幂等元：sort(arr)=sort(sort(arr))

* 1. Property-Based Testing的优缺点
* Dis: 并不总是保证每个输出值都是正确的
* Ad: 可以在不提供预期输出值的情况下检查行为; 支持随机测试; 甚至可以嵌入到源代码中，以便在运行时检查值
  1. Stochastic Testing的优缺点
* Dis: 1. 测试是不可重复的；2. 由于性能测试，并不总是保证输出值的正确性
* Ad: 1. 只需编写一个测试用例，就可以检查多个输入值（测试用例运行次数越多，覆盖率就越高）；2. 输入值不受先入为主的成见影响；3. 基于property testing，考虑invariants
  1. 随机测试与重复性
* Reproducibility可复现性 vs. Repeatability可重复性

Reproducibility可复现性：使类似的defect再次出现

Repeatability可重复性：完完全全重复自身，一模一样

* Stochastic tests are reproducible 随机测试**可复现**

测试人员只需记录随机生成的导致缺陷的输入值；

测试人员可以将输入值传递给编码器以重现缺陷；

* Stochastic tests are not repeatable 随机测试**不可重复**

每次测试运行时，都会通过设计测试一个新的输入值；

即使存在缺陷，测试也可能通过；

很难确定缺陷是何时产生的；

* 1. 随机测试不应成为主要测试方法
* 尽可能使用可重复的测试(tests that use concrete values)
* Test all edge cases and corner cases that are frequent sources of defects(第一道)
* 使用随机测试作为第二道防线（捕捉测试人员甚至没有想到要测试的意外缺陷）

1. QuickCheck: Tool for Stochastic Testing
   1. 两个简单步骤：1. 指定允许输入的property；2. 指定必须保持的输出property- invariant；
   2. @property

QuickCheck then runs randomized test cases for us!

* 1. Shrinking
* 查找可能出现的最小故障；帮助跟踪实际问题；

A “toy” failure is a great thing to add to a defect report；

* 1. @Property(trials=1000) public void testSort(int[] arr) {

// 试1000次

1. Fuzzing Testing 模糊测试

一种以“byte stream”输入为重点的随机测试形式

* 1. Idea: 用“模糊”或“嘈杂”的字节流来填充程序，看看它是否正常 – 从而使程序暴露缺陷和安全漏洞
  2. A byte Stream可以变化为
* 图像查看器或视频播放器的输入文件
* 连接到web服务器的网络数据包
* 由web浏览器解释的JavaScript代码
* 程序的配置文件
  1. Fuzz testing听起来愚笨，它生成一些随机静态数据，并将其强制输入到程序中，直到失败

事实上它并不愚笨：必须更加小心如何生成输入；byte Stream是导致复杂行为的复杂输入形式；与简单的整数输入不同，该输入可以相对轻松地涵盖所有行为

LEC 15 Static Analysis 静态分析

1. Dynamic test: 代码由测试执行，这是我们目前所做的

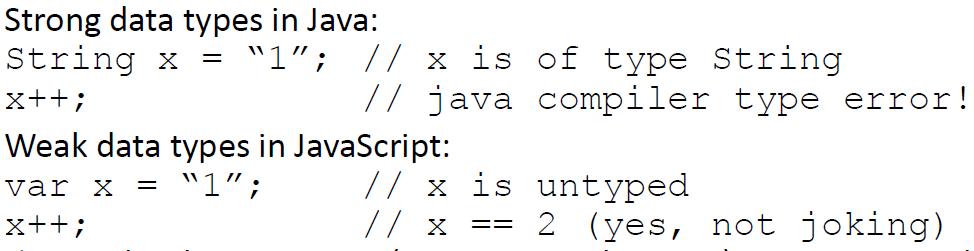
Static test：代码并不由测试执行，通过对代码的分析从而发现defect

* 1. Kinds of Static Tests: Code review / walk through; Compiling; Code coverage; Linters; Bug finders; Formal verification
  2. WHY?
* 比dynamic更加容易：不需测试用例；无需设置软件/硬件来运行程序
* 比dynamic更好定位缺陷：dynamic只告诉某个输入存在缺陷，static分析代码并准确告知修复哪一行代码
* 经常发现dynamic遗漏的缺陷：dynamic受其测试用例的限制，可能会遗漏某些行为；static分析整个代码以查找缺陷
* 为什么不只用static?

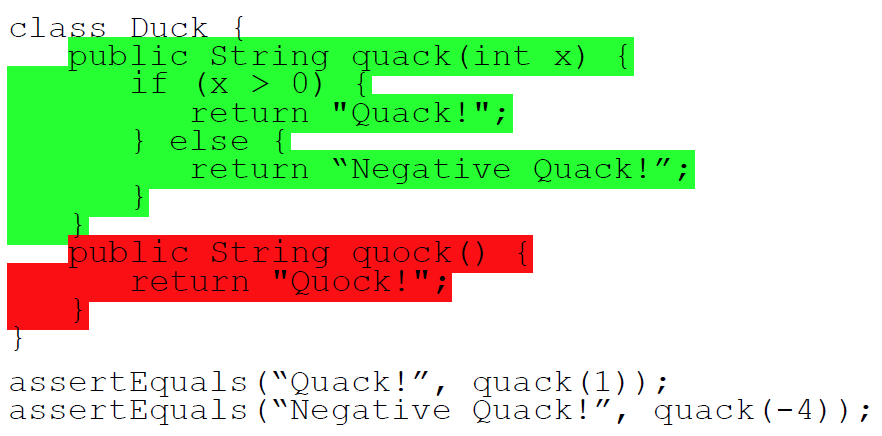
不能发现所有缺陷：即便做了code review和编译，同样可能存在bug

false positives: 认为通过代码审查发现了一个bug，但它不是bug

1. Compiler: 编译器的第一项工作是将源代码翻译成机器代码; 第二项工作是对源代码执行静态检查(error&warning)

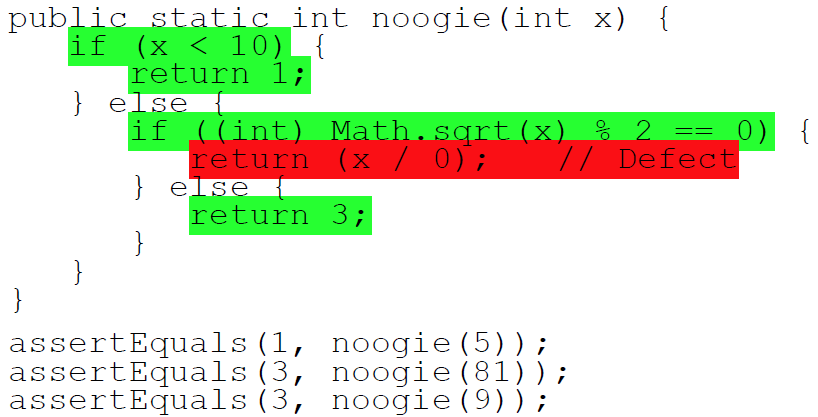
* 尽可能使用好编译器，注意warning
* 语言决定编译器静态分析的有效性；汇编语言没啥用

1. Code coverage：特定测试套件覆盖了多少代码库

需要相当多的静态分析来度量代码覆盖率；在运行前分析代码和检测计数器

有一个方法没测到，所以方法覆盖率是50%

Statement coverage是3/4(if条件句也算)

method coverage：100%

statement coverage:4/5

* 其他种类的codecoverage

1. Branch coverage分支覆盖率: 覆盖的分支方向（如if语句）的百分比
2. Condition coverage条件覆盖率：覆盖的布尔表达式的百分比
3. Path coverage路径覆盖率：覆盖的方法中的路径的百分比
4. Parameter value coverage参数值覆盖率：覆盖的（常用）参数值的百分比
5. Entry/Exit coverage进入/退出覆盖率：覆盖的方法调用/返回的百分比
6. State coverage: 如果程序表示为有限状态机，则覆盖的状态百分比

最普遍的还是指Statement Coverage

100%的测试率不能保证没有defect

1. Linters
2. Bug finders
3. Formal verification

LEC 15 EXTRA **Type Checking类型检查**

1. 类型

* A set of values: integers, strings, objects
* A set of operations: 操作符, 方法调用

两种类型可能具有相同的一组值，但允许的操作不同

LEC16 **Static Analysis 2**

1. **Formal verification** (Kind of Static Test)

证明程序是否有缺陷

1. Theorem Proving 定理证明：用数学方法从前置条件推导后置条件
   1. Hoare Logic霍尔逻辑定理证明：通过数学推导证明霍尔三元组

给定先决条件和程序，后决条件始终为真

{Precondition} Program {Postcondition}

{ x == y } x = x + 3 { x == y + 3 }

* 1. Advantage
     1. 可以证明具有无限状态的大型程序

这是model check无法实现的，它可以无限循环

* + 1. 引导程序员对程序有更深的理解
  1. Disadvantage
     1. 需要大量人工操作
     2. 证明可能会很长，超出正常人的理解极限
  2. Industry Reception 行业接受度
* 工业界需要它，但对此抵制。工业界更希望按钮式一键解决方案

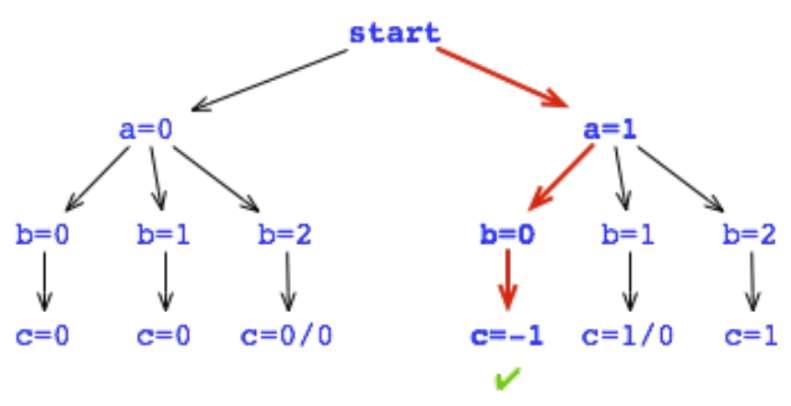
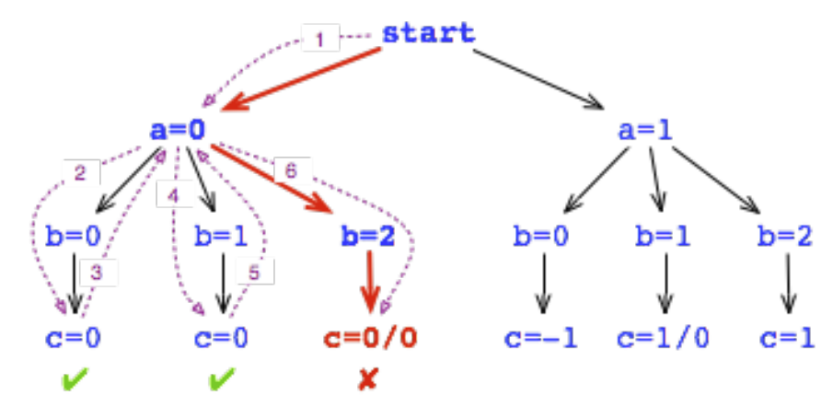
1. Model Checking 模型检查：给定一个系统的有限状态模型，详尽地检查该模型是否满足给定的规范
   1. Does implementation satisfy specification （系统模型是否实现系统属性？）

* Implementation：system model

系统模型可以是源代码，也可以是抽象模型

* Specification：system property

property-based testing 中的属性

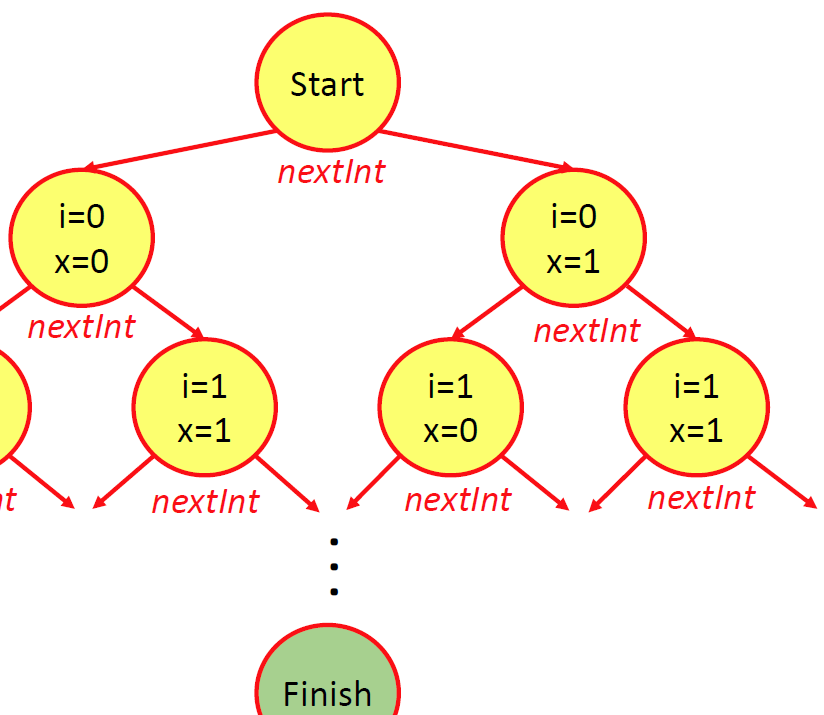
* 1. 一些系统属性中的例子：
* Assertions (invariants)；嵌入源代码或部分基于属性的单元测试；
* 与内存相关的属性；没有越界的数组访问；没有空引用；
* Thread related properties
* Security related properties
  1. model checking与Property-Based Testing的对比
* 相同：模型检查也测试property，而不是输出值
* 不同：通过随机测试，测试了（一些）随机输入值；通过模型检查，将彻底检查所有状态

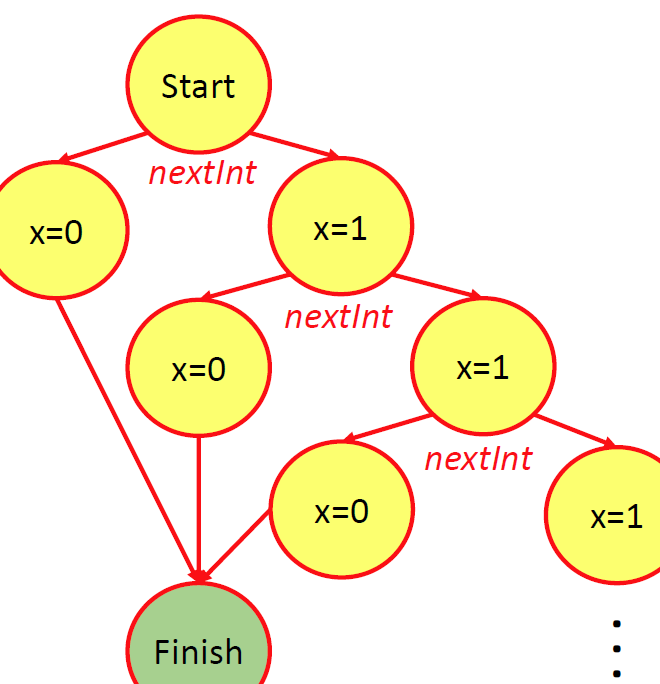
对于随机测试而言：所有测试之后，并不是所有路径都被覆盖了，那么可能永远也找不到bug；对于model check：能够彻底搜查

1. State Explosion Problem：状态爆炸问题 – 可能会以很多种状态结束

* 无法使用model check：可能会遇到内存限制（无法包含整个状态图）；可能会遇到时间限制（无法在分配的时间内浏览整个图表）

Ex1. 指数级的潜在状态数



Ex2. 右翼无限迭代创造无穷多的状态

* 1. Solution: Match & Backtrack
* assume we visit states in Depth First Search (DFS) order
* 例子见slide 21 22
* 逻辑：当有选择时就会出现状态分歧 – match: 当下一个状态与之前状态相匹配，意味着重复，需要backtrack； - backtrack：回到最近的先前的未探索的路径
* 这么做以后仍然可能无限大，但是在有限的时间内能发现更多的defects
  1. ？什么是程序状态
* 程序状态的定义取决于编程语言
* C：基本是整个内存空间
* Java: 由JVM维护的抽象状态 \_\_\_ JVM维护一个栈（用于本地变量）和一个堆（用于对象）；该状态比整个内存空间小得多
* 编程语言的选择对于模型检查也很重要

1. **Java Path Finder (JPF)** 一个java的模型检查器

* 它是一个model checker，搜索程序中所有可能的路径（包括在多线程程序中通过交错的路径）
  1. how？
  2. 验证API
* 枚举输入值

1. JPF with Junit（在JPF上运行junit）

* 枚举一大组输入值及其所有组合
* 对于具有随机值生成的不确定程序，枚举所有随机值以证明测试用例始终通过
* 对于由于线程交错而导致的不确定程序，枚举所有可能的线程交错

Verify

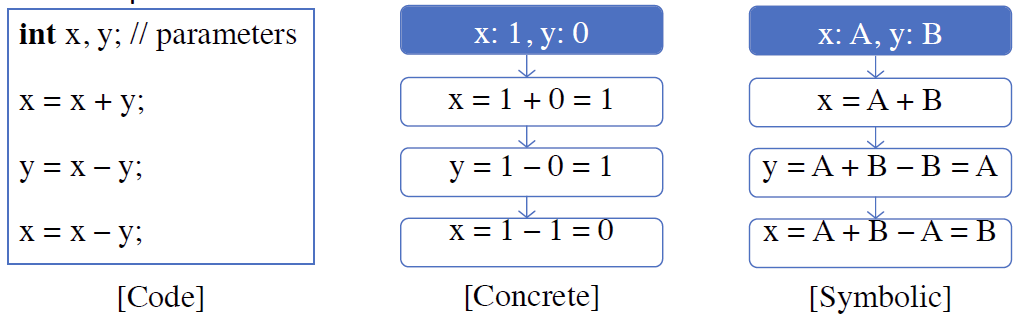
LEC17 **Static Analysis 3**

1. Symbolic Model Checking符号模型检验

* 模型检验可以分为两种
  1. Enumerative model checking枚举模型检验
* 上一章所学的，可能导致state explosion
  1. Symbolic model checking 符号模型检验
* 使用符号进行model check，从根本上避免state explosion

1. Symbolic execution符号执行：在执行期间为变量指定符号表达式而不是实际值

* x = A + 1, y = A \* B
  1. Symbolic expression符号表达式：使用符号值的表达式
* A + 1, A \* B
  1. Symbolic value 符号值：表示输入值的数学符号
* A, B, …, X, Y, Z
* 程序变量：小写lower case – x y z
* 符号值：大写upper case – A B C D (表示给定给代码的输入值; 可以是来自用户输入的值; 可以是来自命令行参数的值; 如果测试方法，可以将值传递到参数中)
  1. 例子 \_符号模型检验像人类一样对待程序
* 输入是符号值，而不是具体的数据值，变量是符号值上的符号表达式



* 1. 如果路线存在分歧？- 建立Path Condition (PC)

LEC18 **Pairwise and Combinatorial Testing** 成对和组合测试

1. 变量交互

1.1 例：测试一个文字处理器，假设有十种可能的字体效果，它们是可以组合的；如果想要详尽测试需要1024次，但这是必要的，但是随着种类增多是指数级增长

* Defect是由变量之间的相互作用产生的吗？如果不是，我们可以单独测试
* 如果由变量之间的相互作用产生：小于10不必测试所有组合，2的话测试
  1. 结论：变量的相互作用会导致缺陷，单一变量缺陷覆盖率只有17%~68%

一个缺陷最多只涉及六个变量，六个交互绝对可以到100%

测试所有的possible pairs，就能发现大多数缺陷，仅由两个交互变量即可覆盖53%~97%

1. Pairwise Testing成对测试

* 测试所有可能的交互对 //all pair testing

Bold / Italic,

* 测试一对中所有可能的交互

Not-Bold / Not-Italic；Bold / Not-Italic；Not-Bold / Italic；Bold / Italic

例. 详尽测试需要1024次，但如果是成对测试呢？

- 一共有十个变量，所以可能有45个pairs；一个pair内有四种结合；只需180次test

- 缩减到8次test：制作covering array覆盖阵列



LEC19 **Smoke and Exploratory Testing**冒烟和探索性测试

1. Smoke Testing

1.1 在送水或其他流体之前，先将烟气送入管道以查找泄漏（先搞点简单的小问题看看会不会出爆炸性bug）

最少的测试以确保系统准备好进行严格的测试（优点在于避免浪费时间精力）

* 1. 别名：Confidence Testing；Sanity Testing；Build Verification Testing (BVT)
  2. BVT 是回归测试的一种形式, 也是Integration / system tests

通常在集成服务器上完成，在虚拟机内

* 1. BVT 通过/失败会发生什么？
* Pass, 通过（自动）电子邮件通知 QA 团队进行进一步测试
* Fail, 代码存储库处于损坏状态。需要：立即将存储库恢复到已知的安全版本，立即修补错误并再次运行 BVT

推荐的 BVT 时间为 ~ 10 分钟

1. BVT实施

2.1 Scripted / Unscripted

- 如果编写脚本，则在每个构建中运行相同的预先制定的测试计划

- 如果没有脚本，经验丰富的测试人员会选择一组与代码更改相关的测试

2.2 Automated / Manual

BVT通常是scripted and automated

sometimes unscripted and automated

1. Exploratory Testing 探索性测试

该功能的确切要求可能尚未确定 - 没有测试计划的测试

又叫“ad hoc” testing

* 1. 如何实施

没有测试计划，也没有预期的行为；本能地知道要测试什么；能地知道什么是缺陷（或需要改进）

从主要功能到edge / corner cases

* 1. 将问题尽可能记录到错误管理系统，使错误未来可以重现，也为写test case做基础
  2. 优点：快速：可以专注于快速发现缺陷

灵活：更新测试没有开销

Improved tester’s knowledge

* 1. 优缺点：不受管制：质量在很大程度上取决于测试人员

不可重复：缺陷不可再现

未知覆盖率：测试后很难判断覆盖率

LEC 20 **Security Testing** 安全测试

1. Security Testing Hardware也很难

为什么难？

* 对手很积极
* 关于安全漏洞的信息迅速传播
* 你需要保护所有的，对手只需要攻破一个
* 资金支持不足