SOFTWARE TESTING

Software Testing

White-Box Testing (1)

School of Computer Science & Engineering
Sun Yat-sen University

Instructor: Guoyang Cai
email: isscgy@mail.sysu.edu.cn

Approaches & Technologies

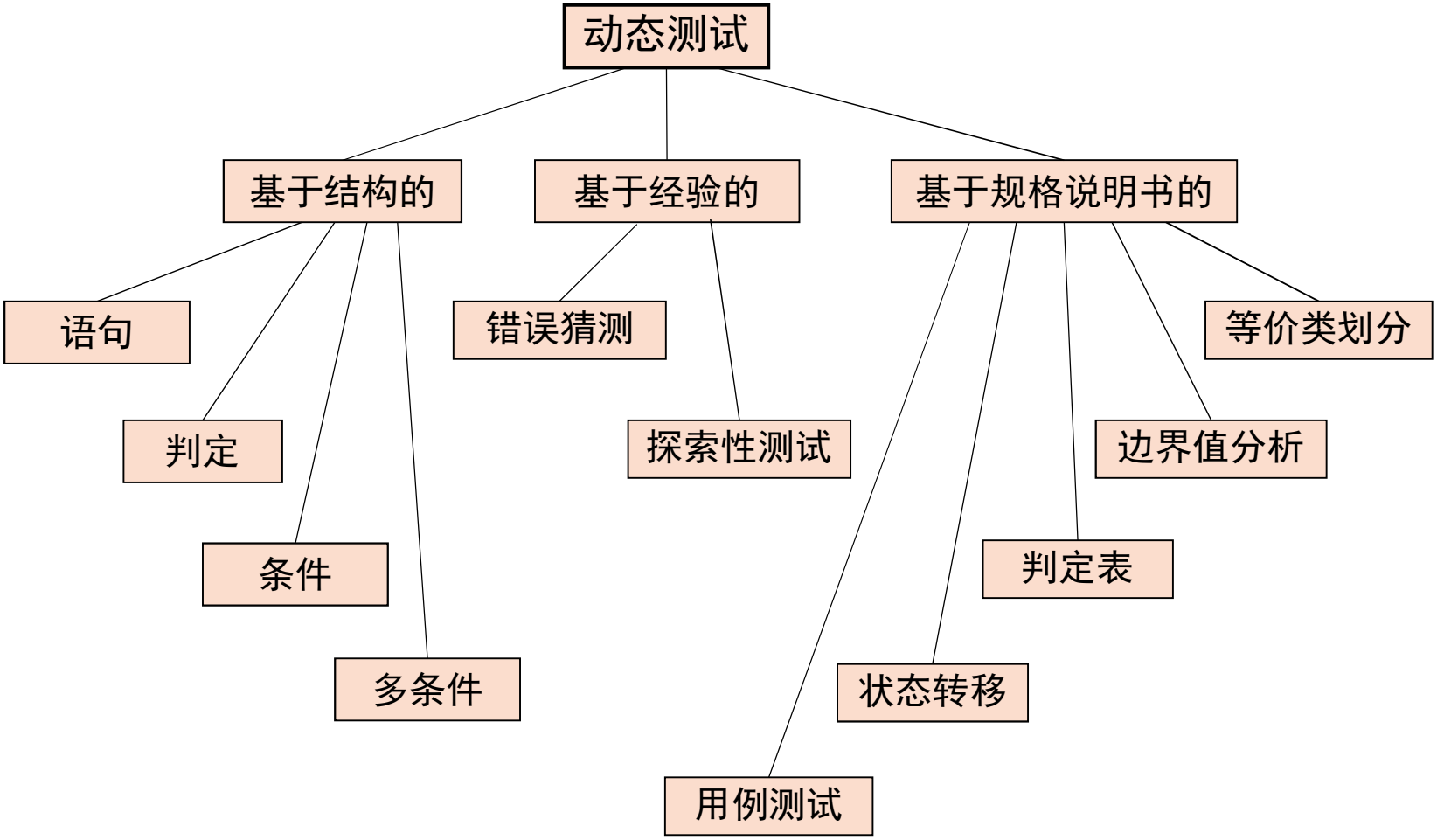
中山大學
SUN YAT-SEN UNIVERSITY

OUTLINE

- 4.1 概述
- 4.2 逻辑覆盖
- 4.3 路径测试
- 4.4 数据流测试
- 4.5 信息流分析
- 4.6 覆盖率分析
- 4.7 覆盖测试准则
- 4.8 基本路径测试

动态测试概述

- Tree structure of DYNAMIC testing techniques



- 动态测试概述
  - 动态测试基本流程
    - 运行被测程序
    - 检查运行结果与预期结果的差异
    - 分析运行效率和健壮性等性能
  - 动态测试由三部分组成
    - 构造测试实例
    - 执行被测程序
    - 分析输出结果

- 动态测试分类
  - 动态测试的分类
    - 从是否了解软件内部结构 (程序源代码) 的角度划分：
      - 白盒测试
      - 黑盒测试
      - 灰盒测试
    - 从软件开发过程的角度划分：
      - 单元测试、集成测试、功能/确认测试、系统测试、验收测试及回归测试
    - 从测试执行时是否需要人工干预的角度划分：
      - 人工测试
      - 自动化测试
    - 从测试实施组织的角度划分：
      - 开发方测试 (α 测试)、用户测试 (β 测试)、第三方测试

- 白盒测试
  - 白盒测试的概念
    - 白盒测试按照程序内部逻辑结构和编码结构来设计测试数据并完成测试，是一种典型的动态测试方法。
      - 白盒测试又称为结构测试或逻辑驱动测试。
      - 白盒测试直接分析源代码，确定测试内容和测试方法。
      - 白盒测试应该覆盖全部代码、分支、路径和条件。
  - 白盒测试的主要特点
    - 可以针对被测程序的特定部分构造测试用例
    - 有一定的充分性度量手段
    - 可以获得较多工具支持
    - 通常只用于单元测试

- 白盒测试
  - 白盒测试的基本测试内容
    - 对程序模块的所有独立执行路径至少测试一次。
    - 对程序模块中所有的逻辑判定，取“真”与取“假”的两种情况都至少测试一次。
    - 在循环边界和运行边界的界限内执行循环体。
    - 测试内部数据结构的有效性。
  - 白盒测试采用的测试方法
    - 逻辑覆盖测试
      - 包括语句覆盖、判定覆盖、条件覆盖、判定-条件覆盖、条件组合覆盖以及路径覆盖 。
    - 路径测试
    - 数据流测试
    - 信息流分析

- 逻辑覆盖概述
  - 逻辑覆盖的概念
    - 逻辑覆盖是以程序内部的逻辑结构为测试基础的一种白盒测试方法。
    - 逻辑覆盖方法建立在测试人员对程序的逻辑结构清晰了解的基础上，是一大类测试过程的总称。
  - 逻辑覆盖方法的主要分类
    - 语句覆盖
    - 判定覆盖
    - 条件覆盖
    - 判定-条件覆盖
    - 条件组合覆盖
    - 路径覆盖

语句覆盖

- 语句覆盖要求设计足够多的测试用例，使得被测程序的每一条语句至少被执行一次。
  - 例1：C 语言程序段落如下

```
func (int a, b, x)
{
    if ((a > 1) && (b == 0))
        x = x / a;
    if ((a == 2) || (x > 1))
        x = x + 1;
}
```

  - 测试用例 (a, b, x) = (2, 0, 3) 满足语句覆盖要求。
  - 语句覆盖是弱的逻辑覆盖标准。

- 语句覆盖
  - 语句覆盖的优点
    - 检查所有语句，代码覆盖率高
    - 结构简单的代码的测试效果较好
    - 容易实现自动测试
    - 如果是程序块覆盖，则不用考虑程序块中的源代码
  - 语句覆盖不能检查出的错误
    - 条件语句错误
      - 例如 `if ((a > 1) && (b == 0))`

        误为 `if ((a > 0) && (b == 0))`
    - 逻辑运算错误
      - 例如 `if ((a > 1) && (b == 0))`

        误为 `if ((a > 1) || (b == 0))`
    - 循环语句错误
      - 例如循环控制次数错误、循环条件错误。

语句覆盖

语句覆盖与编码规范

思考：有 C 语言程序段落

```
func (int a, b, x)
{
    if ((a > 1) && (b = 0))
        x = x / a;
    if ((a = 2) || (x > 1))
        x = x + 1;
}
```

- 编码中有哪些地方不符合一般的编码规范？
- 例1 的测试用例 (a, b, x) = (2, 0, 3) 不能满足语句覆盖要求。实际上由于赋值表达式 `b = 0` 的值是0，无论如何选择测试用例，语句 `x = x / a` 总不能得到执行。

- 判定覆盖 (分支覆盖)
  - 判定覆盖的测试用例
    - 判定覆盖要求设计足够多的测试用例，使得被测程序中的每一个 (判定) 分支至少通过一次。
      - 每一个判定语句的“真”值分支和“假”值分支都至少得到一次执行。
      - While 语句、switch 语句、异常处理、跳转语句和三目运算符 (a?b:c) 等同样可以使用判定覆盖进行测试。
      - 对多分支语句，例如 C 语言中的 case 语句，判定覆盖必须对每一个分支的每一种可能的结果都进行测试。

判定覆盖

判定覆盖的测试用例 (续)

例2： C 语言程序段落如下

```
func (int a, b, x)
{
    if ((a > 1) && (b == 0))
        x = x / a;
    if ((a == 2) || (x > 1))
        x = x + 1;
    else
        x = x - 1;
}
```

- 测试用例 (a, b, x) = (3, 0, 1) 和 (a, b, x) = (2, 1, 3) 满足判定覆盖要求。

判定覆盖

判定覆盖的测试用例 (续)

例2： C 语言程序段落如下

```
func (int a, b, x)
{
    if ((a > 1) && (b == 0))
        x = x / a;
    if ((a == 2) || (x > 1))
        x = x + 1;
    else
        x = x - 1;
}
```

判定表达式

- 测试用例 (a, b, x) = (3, 0, 1) 和 (a, b, x) = (2, 1, 3) 满足判定覆盖要求。

- 判定覆盖
  - 判定覆盖的评价
    - 判定覆盖的查错能力强于语句覆盖。
      - 执行了判定覆盖，实际上也就执行了语句覆盖。
    - 判定覆盖与语句覆盖存在同样的缺点。
      - 不能发现条件语句错误
      - 不能发现逻辑运算错误
      - 不能发现循环次数错误
      - 不能发现循环条件错误

- 条件覆盖
  - 条件覆盖的测试用例
    - 条件覆盖要求设计足够多的测试用例，使得程序中的每一个判定中的每个条件获得所有各种可能结果。
      - 判定覆盖和条件覆盖的区别与联系
        - 判定覆盖逻辑依赖于判定表达式整体取值的“真”、“假”情况，而不考虑判定表达式内部的逻辑结构。
        - 一个包含 n >1 个条件的复合判定表达式具有 2^n 个条件取值组合，判定覆盖不能覆盖全部条件。
        - 条件覆盖逻辑依赖于判定表达式中的条件表达式取值的“真”、“假”情况，而不考虑条件之间的组合，条件覆盖不一定能够覆盖全部判定分支。
        - 当每一个判定表达式都是单条件表达式时，判定覆盖就是条件覆盖。

条件覆盖

条件覆盖的测试用例 (续)

例2： C 语言程序段落如下

```
func (int a, b, x)
{
    if ((a > 1) && (b == 0))
        x = x / a;
    if ((a == 2) || (x > 1))
        x = x + 1;
    else
        x = x - 1;
}
```

- 测试用例 (a, b, x) = (2, 0, 4) 和 (a, b, x) = (1, 1, 1) 满足条件覆盖要求 (注意到在本例中它们同时也满足判定覆盖要求)。

条件覆盖

条件覆盖的测试用例 (续)

例2： C 语言程序段落如下

```
func (int a, b, x)
{
    if ((a > 1) && (b == 0))
        x = x / a;
    if ((a == 2) || (x > 1))
        x = x + 1;
    else
        x = x - 1;
}
```

条件表达式

- 测试用例 (a, b, x) = (2, 0, 4) 和 (a, b, x) = (1, 1, 1) 满足条件覆盖要求 (注意到在本例中它们同时也满足判定覆盖要求)。

条件覆盖

条件覆盖的测试用例 (续)

例2： C 语言程序段落如下

```
func (int a, b, x)
{
    if ((a > 1) && (b == 0))
        x = x / a;
    if ((a == 2) || (x > 1))
        x = x + 1;
    else
        x = x - 1;
}
```

判定表达式

- 测试用例 (a, b, x) = (2, 0, 4) 和 (a, b, x) = (1, 1, 1) 满足条件覆盖要求 (注意到在本例中它们同时也满足判定覆盖要求)。

条件覆盖

条件覆盖的测试用例 (续)

例2： C 语言程序段落如下

```
func (int a, b, x)
{
    if ((a > 1) && (b == 0))
        x = x / a;
    if ((a == 2) || (x > 1))
        x = x + 1;
    else
        x = x - 1;
}
```

- 思考：测试用例 (a, b, x) = (2, 0, 1) 和 (a, b, x) = (1, 1, 2) 虽然满足条件覆盖，但不满足判定覆盖，在上述测试用例下分支 x = x - 1 不能得到执行。

- 条件覆盖
  - 条件覆盖的评价
    - 能够检查所有的条件错误。
    - 不一定能够实现对每个判定分支的检查。
      - 可能需要增加用例数量。

- 判定-条件覆盖
  - 判定-条件覆盖的测试用例
    - 判定-条件覆盖要求设计足够多的测试用例，使得判定中每个条件的所有可能取值至少能够获取一次，而且每个判断的所有可能的判定结果 (即每个分支) 至少执行一次。
      - 用于解决条件覆盖不一定包括判定覆盖、判定覆盖也不一定包括条件覆盖的问题。

判定-条件覆盖

判定-条件覆盖的测试用例

例2：C 语言程序段落如下

```
func (int a, b, x)
{
    if ((a > 1) && (b == 0))
        x = x / a;
    if ((a == 2) || (x > 1))
        x = x + 1;
    else
        x = x - 1;
}
```

- 测试用例 (a, b, x) = (2, 0, 4) 和 (a, b, x) = (1, 1, 1) 既满足条件覆盖要求，也满足判定覆盖要求。

- 判定-条件覆盖
  - 判定-条件覆盖的评价
    - 判定-条件覆盖既考虑了每一个条件，又考虑了每一个分支，发现错误能力强于单独的判定覆盖和条件覆盖。
    - 满足判定-条件覆盖要求的测试用例不一定能覆盖所有路径。
      - 可能需要增加用例数量。

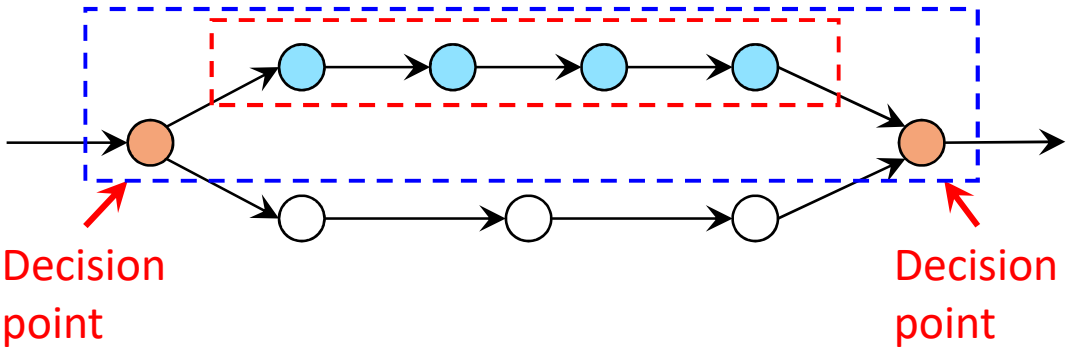
- 条件组合覆盖
  - 条件组合覆盖的测试用例
    - 条件组合覆盖要求设计足够多的测试用例，使得每个判定中条件的各种组合至少出现一次。
      - 满足条件组合覆盖标准的测试用例，也一定满足判定覆盖、条件覆盖和判定-条件覆盖标准。
      - 需要指数级的用例数量。
      - 满足条件组合覆盖要求的测试用例并不一定能覆盖所有路径。

- 路径覆盖
  - 路径覆盖的测试用例
    - 路径覆盖要求设计足够多的测试用例，使得程序中所有的路径都至少执行一次。
  - 路径覆盖的评价
    - 路径覆盖对所有程序路径进行测试，发现错误能力较强。
    - 程序路径数量庞大，难以实现真正的路径覆盖。
    - 用例数量将急剧增加。
    - 仍然无法保证能够发现所有的条件错误。

概述

- 路径测试是根据程序的逻辑控制结构所产生的执行路径进行测试用例设计的方法。
  - 路径测试从程序的入口开始，执行所经历的各个语句，到达程序的出口，是程序一次执行的完整过程；广义上任何有关路径分析的测试都可以被称为路径测试。
- 路径测试的理想情况是设计足够的测试用例，测试程序的所有可执行路径，从而实现路径覆盖。对于高复杂性的程序，要做到路径覆盖存在很大困难。

- DD-路径测试
  - DD-Path
    - DD-path, Decision-to-Decision path, the name refers to a sequence of statements that begins with the “out-way” of a decision statement and ends with the “in-way” of the next decision statement. No internal branches occur in such a sequence.



DD-路径测试

DD-Path

- A chain is defined as a path in which:
  - initial and terminal nodes are distinct, and all interior nodes have indeg = 1 and outdeg = 1.
- The length of a chain is the number of edges it contains.
- A maximal chain is a chain that is not a subpath of another chain.
- A DD-path is a set of nodes in a program's Control Flow Graph such that one of the following holds:
  (1) It consists of a single node with indeg = 0 (initial node).
  (2) It consists of a single node with outdeg = 0 (terminal node).
  (3) It consists of a single node with indeg ≥ 2 (merge points) or outdeg ≥ 2 (decision points).
  (4) It consists of a single node with indeg = 1 and outdeg = 1, but not consisted in a chain.
  (5) It is a maximal chain of length ≥ 1.
- A DD-path is a node in the corresponding DD-graph.

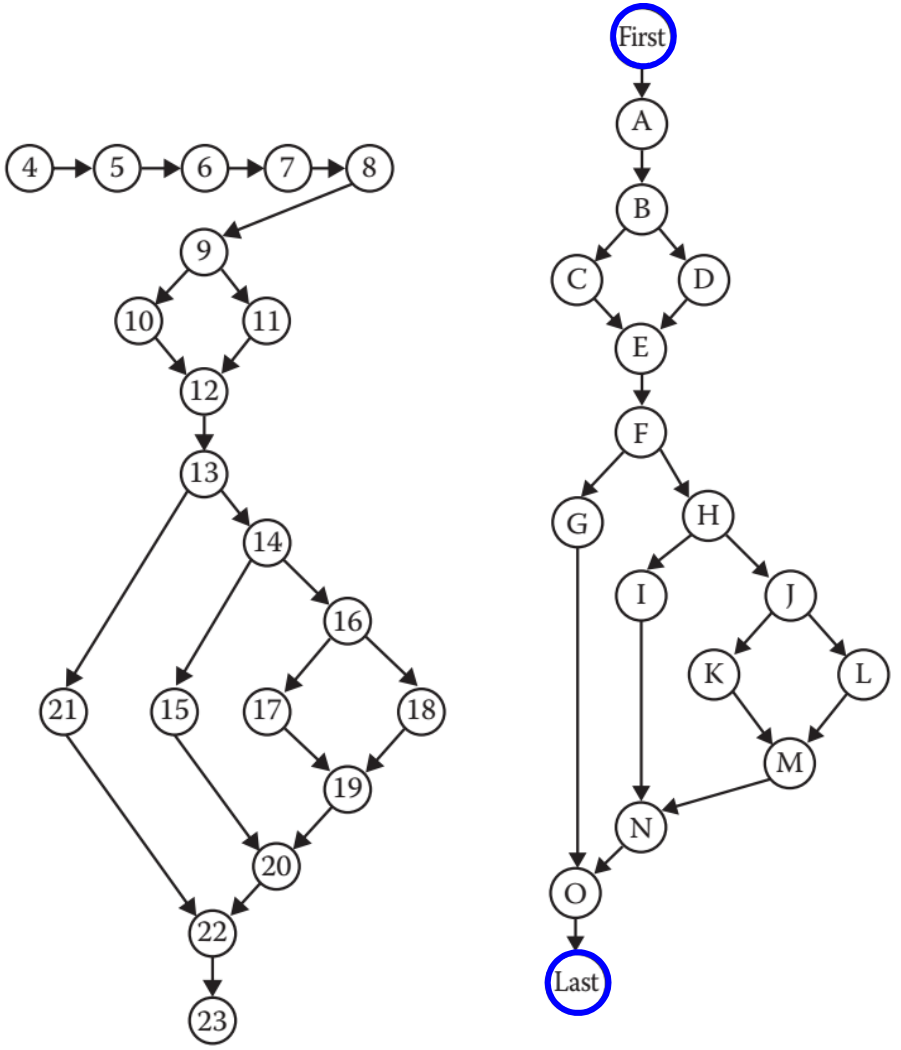
DD-路径测试

- 程序控制流图 (CFG) 到 DD-图 (压缩图) 的转换：
  (1) 取得 CFG；
  (2) 在 CFG 上应用规则 (1)-(3)，确定程序起始点、终结点和各个决策点的 DD-路径；
  (3) 在 CFG 余下的结点上应用规则 (4)-(5)，确定其他的 DD-路径；
  (4) 每一条 DD-路径对应于 DD-图上的一个结点；
  (5) DD-路径之间的连接关系在 DD-图上表示为对应结点的邻接关系。

DD-路径测试

例3：

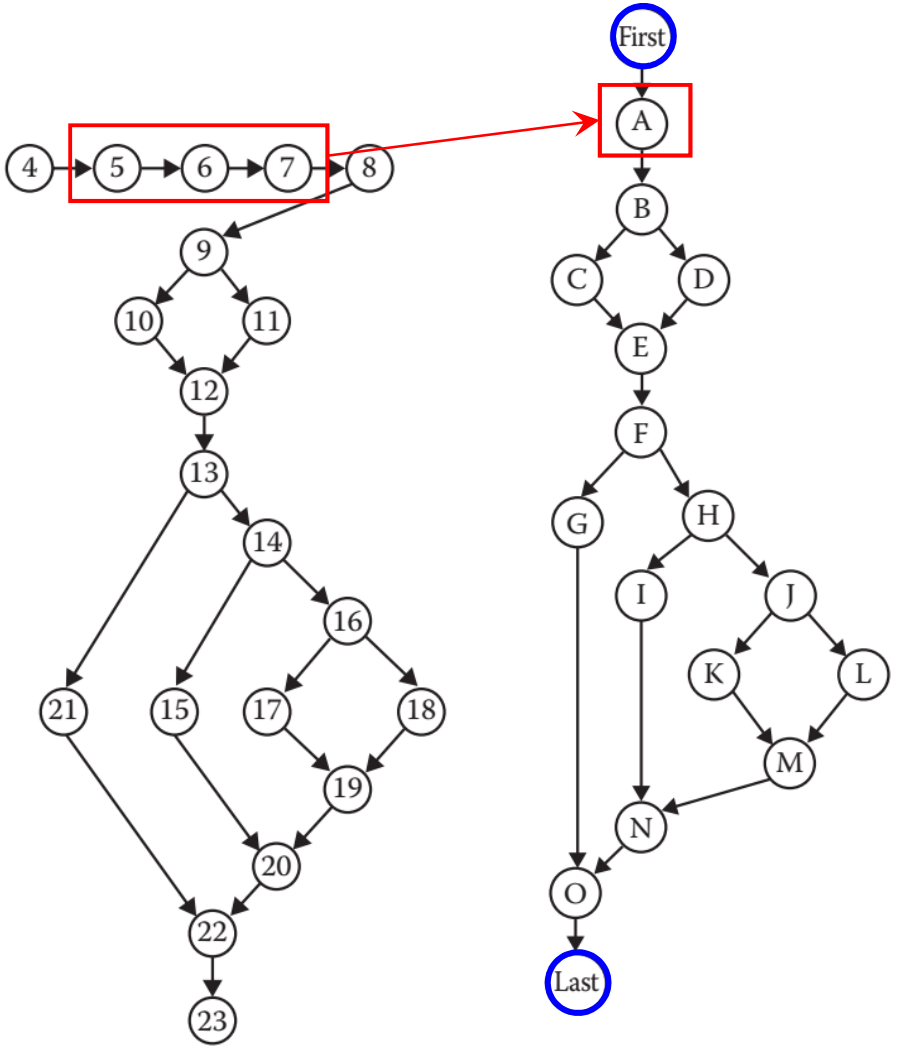
Nodes	DD-Path	Case of definition
4	First	1
5-8	A	5
9	B	3
10	C	4
11	D	4
12	E	3
13	F	3
14	G	3
15	H	4
16	I	3
17	J	4
18	K	4
19	L	3
20	M	3
21	N	4
22	O	3
23	Last	2



DD-路径测试

例3：

Nodes	DD-Path	Case of definition
4	First	1
5-8	A	5
9	B	3
10	C	4
11	D	4
12	E	3
13	F	3
14	G	3
15	H	4
16	I	3
17	J	4
18	K	4
19	L	3
20	M	3
21	N	4
22	O	3
23	Last	2



- DD-路径测试
  - 测试覆盖率
    - 测试覆盖率考虑对命令式语言 (Imperative Language) 程序流程图中各个分支情况的测试覆盖程度，因此可以对流程图中线性串行的部分进行压缩，在 DD-路径的基础上进行测试用例设计，用测试覆盖指标考察测试效果。压缩图清晰描述了程序执行的分支情况，便于进行覆盖率分析。
    - 很多质量评测机构把 DD-路径覆盖作为测试覆盖的最低可接受级别。
  - DD-路径的作用
    - 使用一组满足 DD-路径覆盖要求的测试用例，可以发现约85%的代码缺陷。
    - 如果 DD-路径图中的每条边得到遍历，则程序的每个判断分支都得到了执行。

基本路径测试

- 概述
  - 实际应用中一个不太复杂的程序的路径都是一个庞大的数字。
  - 在测试中覆盖所有路径是不现实的。在不能做到所有路径覆盖的情况下，如果被测程序的每一条独立路径都被测试过，则可认为程序中的每个语句都已经得到检验或覆盖。
- 基本路径测试是 *McCabe* 提出的一种白盒测试方法。
  - 根据过程设计画出程序控制流图 (CFG)；
  - 计算程序控制流图的 *McCabe* 环路复杂度；
  - 确定一个线性独立路径 (数量由 *McCabe* 环路复杂度确定) 的基本集合；
  - 为上述每条独立路径设计可强制执行该路径的测试用例；
  - 测试用例总体保证了语句覆盖和 (单条件判定情况下的) 条件覆盖。

循环路径测试

- 循环路径的测试用例需要检查下列情况：
  - 0 次循环：检查跳过循环条件；
  - 1 次循环：检查循环初始值；
  - 2 次循环：检查多次循环；
  - m 次循环：检查某次循环；
  - 最大次数、比最大次数多一次、比最大次数少一次的循环：检查循环次数边界。
- 循环测试的过程简化
  - 循环使路径数量急剧增长，为此需对循环测试过程进行简化。
  - 无论循环的形式和实际执行循环体的次数多少，只考虑循环1次和0次 (即进入循环体一次和跳过循环体)。

概述

- 数据流测试也称“定义/引用”测试，其目的是发现定义/引用的异常缺陷。
  - 发现被定义后从未引用的变量
  - 发现没有被定义的变量
  - 发现重复定义的变量

数据流测试的重点

- 数据流测试重点关注变量的定义与使用。
  - 调试修改代码错误时，我们可能会在一段代码中搜索某个变量的定义和引用，考察程序运行时该变量的值的变化，据此分析错误产生的原因。程序是一个程序元素对数据访问的过程。
  - 数据流测试将这种方法形式化，便于构造测试算法，实现自动化分析。
  - 数据流测试关注变量定义与引用位置，它是一种结构性测试方法，也可看作是基于路径测试的一种改良方案 (进行“真实性检查”)。
    - 数据流测试用数据流图描述数据的“定义-使用”路径并进行“真实性检查”，发现数据的不正确定义及使用。
    - 一种简单的数据流测试策略要求测试用例覆盖每个数据的“定义-使用”路径一次。

- 信息流分析的重点
  - 信息流测试通过分析输入数据、输出数据和语句之间的关系来检查程序错误。
    - 还可用来分析是否存在无用的语句。
  - 信息流分析的具体作用
    - 能够列出对输入变量的所有可能的引用。
    - 在程序的任何指定点检查某些语句，其执行可能影响某一输出变量值。
    - 为输入输出关系提供一种检查，观察每个输出值是否由相关的输入值导出。

覆盖率分析

- 代码覆盖率
  - 覆盖测试的目标
    - 对程序模块的所有独立的执行路径至少测试一次；
    - 对所有的逻辑判定，取“真”与取“假”的两种情况都至少测试一次；
    - 在循环的边界和运行界限内执行循环体；
    - 测试内部数据结构的有效性；
    - 其它。
  - 代码覆盖率是指进行白盒测试时测试用例对程序内部逻辑的覆盖程度。
    - 最理想的白盒测试是实现程序的语句覆盖、分支覆盖以及路径覆盖，实现难度大，需要采用其它标准来度量覆盖的程度。
    - 覆盖率分析对代码的执行路径覆盖范围进行评估。这些覆盖从不同要求出发，为测试用例的设计提供依据。

覆盖率分析

- 覆盖率分析
  - 逻辑覆盖率计算方法
    - 逻辑覆盖率主要指语句覆盖率、判定覆盖率、条件覆盖率、判定/条件覆盖率、条件组合覆盖率和路径覆盖率。

      覆盖率 = (至少被执行一次的项目数)/项目总数.

      - 项目可以是需求、语句、分支、条件、路径等等；
      - 覆盖率公式对项目的覆盖情况进行计算。
    - 覆盖率是用来度量测试完整性的一个手段，不是测试的目的。
    - 通过覆盖率数据，可以估计测试是否充分，测试弱点在哪些方面，进而指导我们去设计能够增加覆盖率的测试用例。

- ESTCA 准则
  - ESTCA：错误敏感测试用例分析 (Error Sensitive Test Cases Analysis, *K. A. Foster*)。
  - 规则1：对于 A rel B (rel 是关系运算符 <, ==, 或 >， A, B 是变量) 型的判定谓词，适当选择 A 与 B 的值，使得测试执行到该判定语句时，A < B, A == B 和 A > B 的情况分别出现一次。
    - 目的：检测关系运算符 rel 的错误。
  - 规则2：对于 A rel1 C (rel1 是关系运算符 < 或 >，A 是变量，C 是常量) 型的判定谓词，当 rel1 为 < 时，适当选择 A，使 A = C-M (M 是距 C 最小的容许正数，若 A 和 C 均为整型时，M = 1)；同理，当 rel1 为 > 时，适当选择 A，使 A = C + M。
    - 目的：检测“差1”类错误。
    - 例如：如本应是 " if A > 1 " 而错成 " if A > 0 "。

- ESTCA 准则
  - 规则3：对外部输入变量赋值，使其在每一测试用例中均有不同的值与符号，并与同一组测试用例中其它变量的值与符号不一致。
    - 目的：检测程序语句中的错误。
      - 例：如果将引用一个外部输入变量错写成引用一个常量，将导致应用规则3的两次测试结果相同。
  - 上述三项规则适用基于经验的测试用例设计，虽然不是完备的，但规则本身针对程序编写人员容易发生的错误，或是围绕着发生错误的频繁区域，提高了发现错误的命中率，在普通程序的测试中确实有效。

- **LCSAJ 覆盖准则
  - LCSAJ：线性代码序列和跳转 (Linear Code Sequence and Jump Coverage, M. R. Woodward)。
  - 一个 LCSAJ 是一组顺序执行的程序代码。
    - 起始于程序的入口，或者是一个转移语句的入口点，或者是一个控制流可跳达的点；
    - 结束于程序的出口或者是一个可能导致控制流跳转的点。
  - 程序的 LCSAJ 路径
    - 几个 LCSAJ 首尾相接构成一个 LCSAL 串，如果第一个 LCSAJ 起点为程序起点，最后一个 LCSAJ 终点为程序终点，则组成程序的一条 LCSAJ 路径。
  - LCSAJ 路径不同于 DD-Path。DD-Path 由程序流程图决定，一个 DD-Path 是两个判断之间的路径，但其中不再有判断。

覆盖测试准则

- **LCSAJ 覆盖准则
  - LCSAJ 覆盖准则是一个分层的覆盖准则：
    - 第1层：语句覆盖
    - 第2层：分支覆盖
    - 第3层：LCSAJ 覆盖
      - 程序中的每一个 LCSAJ 至少在测试中经历一次。
    - 第4层：两两 LCSAJ 覆盖
      - 程序中每两个首尾相连的 LCSAJ 组合起来在测试中都要经历一次。
    - 第 n+2 层：每 n 个首尾相连的 LCSAJ 组合在测试中都要经历一次。

- **LCSAJ 覆盖准则
  - LCSAJ 覆盖准则的应用
    - 按照 LCSAJ 覆盖准则，越是高层的覆盖越难满足。
    - 在实施测试时，若要实现上述层次 LCSAJ 覆盖，需要产生被测程序的所有 LCSAJ。
    - 尽管 LCSAJ 覆盖要比判定覆盖复杂的多，但是 LCSAJ 的自动化过程相对比较容易实现。
    - 一个模块的微小改动都可能对 LCSAJ 产生重大影响，因此维护 LCSAJ 的测试数据相当困难。
    - 一个大模块包含极其庞大的 LCSAJ，因此要获得100%的覆盖率并不现实。
    - 证据表明，把测试100%的 LCSAJ 作为目标比100%的判定覆盖要有效的多。

Lecture 16. White-Box Testing (1)

End of Lecture