白盒测试
White-box testing

1/76

1

提纲

- 白盒测试概述
- 基于数据流的白盒测试
- 基于控制流的白盒测试
- 基本路径测试（McCabe圈覆盖）
- Foster的ESTCA覆盖准则
- xUnit简介

2/76

2

提纲

- 白盒测试概述
- 基于数据流的白盒测试
- 基于控制流的白盒测试
- 基本路径测试（McCabe圈覆盖）
- Foster的ESTCA覆盖准则
- xUnit简介

3/76

3

1. 白盒测试概述

- 白盒测试概念
- 为什么要引入白盒测试
- 白盒测试分类
- 白盒测试的应用范围
- 白盒测试的优缺点
- 白盒测试过程

4/76

4

1.1 白盒测试概念

- 白盒测试(white-box testing)：通过分析组件/系统的内部结构，并设计相应测试用例进行的测试。
- 白盒测试又称结构测试、透明盒测试、逻辑驱动测试、基于代码的测试。白盒测试是一种基于软件内部路径、结构和代码基础上的软件测试策略。

5/76

5

1.2 为什么要引入白盒测试

- 对于行业要求严格的代码，比如用于军事、金融、医学等的代码通常必须进行严格的白盒测试；
- 白盒测试由于能够洞察程序中的一切，因此往往可以进行比较完整的测试。

例：下面是一段登录银行网站的代码

```
void    Login( double  Account, CString
   pwd)
{   /// 1如果登录帐号检查正确
   if( Check_Account( Account, pwd ) )
   {    ……
   ///2 将登录帐号和密码发给指定位置
   Send_To_Me( Account, pwd );///后门
   }
   …
}
```

6/76

6

1.3 白盒测试的分类

- 按照是否运行软件，白盒测试可以分为静态白盒测试和动态白盒测试。
- 白盒测试方法：
  - 控制流分析
  - 数据流分析
  - 覆盖分析
  - 路径分析
  - 符号化测试

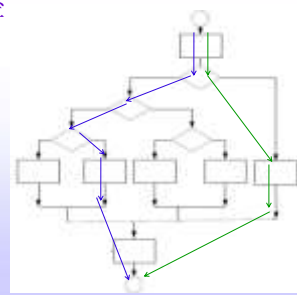
7

1.4 白盒测试的应用范围

- 白盒测试可以用于所有的系统开发阶段，包括单元测试、集成测试和系统测试；
- 白盒测试通常进行路径测试，可以测试单元内部、单元之间、子系统之间以及系统内部的各种执行路径；
- 路径（测试路径）：从开始到结束执行之间运行的语句序列。

8

测试路径



9

1.5 白盒测试的优缺点

- 白盒测试的优点

1. 使程序员注意对于代码的自我审查
2. 可以测试代码中的每条分支路径，对代码的测试比较彻底，可以证明测试工作的完整性
3. 揭示隐藏在代码中的缺陷，保证程序中没有不该存在的代码
4. 根据内部结构进行最优化测试

10

1.5 白盒测试的优缺点

- 白盒测试的缺点

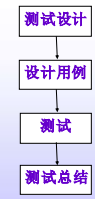
1. 执行路径可能非常多，造成无法进行完全测试
2. 测试员必须具备编程知识
3. 通常而言，由于涉及到代码分析，白盒测试的效率不高，导致测试成本过高

11

1.6 白盒测试过程

- 通常的白盒测试过程：

1. 分析被测软件的内部实现
2. 识别被测软件的工作路径
3. 选择输入，执行被测路径，并确定期望的测试结果
4. 运行测试
5. 比较真实输出和期望输出的异同
6. 判断被测软件正确性



12

提纲

- 白盒测试概述
- 基于数据流的白盒测试
- 基于控制流的白盒测试
- 基本路径测试（McCabe圈覆盖）
- Foster的ESTCA覆盖准则
- xUnit简介

13/76

13

2 基于数据流的白盒测试

- 通过查看代码中变量的定义与引用等情况，可以判定软件可能存在的数据方面的隐患或错误，这被称为基于数据流的白盒测试。

14/76

14

2.1 数据流测试的基本概念

- 变量被定义——如果变量x的值被某条语句修改，则称x被该语句定义；变量被定义通常意味着变量被赋值；
- 变量被引用——如果在某条语句中引用了x的值，则称该语句引用了x；变量被引用意味着该变量存在于赋值语句的右边或在一个不改变其值的表达式中。

举例：

```
double Area, r;      -------- (1) Area和r既没有被定义也没有被引用
Area = 3.14*r*r;     --------- (2) Area被定义，r被引用
if（ r < 1 ）        ---------- (3) r被引用
```

15/76

15

2.2 数据流测试的评定依据

- 判断代码是否存在错误隐患的依据是：
  1. 每一个被引用的变量必须预先被定义；
  2. 被定义的变量一定要在程序中被引用；
- 第1条规则表明：变量必须先定义，后使用，否则有隐患
- 第2条规则表明：被定义的变量需要被使用，否则无意义

16/76

16

2.3 数据流测试举例

- 例 计算n的阶乘

```
int Factorial ()
{  int   i, n, f;
   int   result = 0 ;
   if( n < 0 )
     return(0);
   else {
     f = 1;
     for(i=1; i<=n; i++)
           f = f*i;
     return ( f );
   }
}
```

序号	语句	被定义的变量	被引用的变量
1	int i, n, f;		
2	int result = 0;	result	
3	if(n < 0)		n
4	return (0);		
5	else{		
6	f=1	f	
7	for(i=1; i<=n; i++)	i	i, n
8	f = f * i;	f	f, i
9	return (f) }		f

17/76

17

2.4 数据流测试非常成熟

- 基于数据流的白盒测试技术已经非常成熟，通常可以由编译器自动检查。

18/76

18

提纲

- 白盒测试概述
- 基于数据流的白盒测试
- 基于控制流的白盒测试
- 基本路径测试（McCabe圈覆盖）
- Foster的ESTCA覆盖准则
- xUnit简介

19/76

19

3 基于控制流的白盒测试

- 基于控制流的白盒测试：需要测试程序的状态以及其中的程序流程，必须设法进入和退出每一个模块，执行每一行代码，追踪每一条逻辑和决策分支，这些测试称为控制流测试。

20/76

20

3.1 控制流测试概述

- 控制流测试通过识别程序代码中的执行路径建立覆盖那些路径的测试案例
- 通常情况下，进行彻底地控制流路径测试是不现实的。测试路径的数量太多造成无法在合理的时间内测试完成

举例：路径太多使控制流无法完全测试

```
for ( i=0; i<1000; i++ )
    for ( j=0; j<1000;j++ )
        for ( k=0; k<1000; k++ )
            doSomething (i, j, k );
```

doSomething () 函数将执行10亿次 (1000^3)，相当于10亿条路径

21/76

21

3.2 控制流图

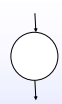
- 尽管控制流测试存在不足，但它仍然是白盒测试的重要工具
- 控制流图是控制流测试的基础，控制流图包含3种成分：
  1. 进程块（Process Block）
  2. 判定点（Decision Point）
  3. 连接点（Junction Point）

22/76

22

3.2.1 控制流图的进程块

- 进程块是从头至尾顺序执行的程序语句序列，只有起始块可以没有进入点，只有结束块可以没有退出点
- 一旦进程块启动，其内部的每一条语句都会被顺序执行
- 进程块使用含有一个输入和一个输出的圆圈表示



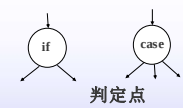
Process block

23/76

23

3.2.2 控制流图的判定点

- 判定点是模块中控制流能够改变方向的点。很多判定点是二值化的（使用if-then-else实现）；多路判定点通常使用case语句实现。
- 判定点使用包含一个输入和多个输出的圆圈表示



判定点

24/76

24

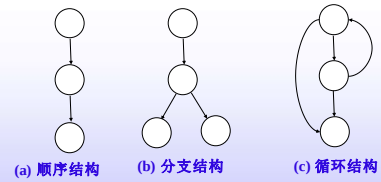
3.2.3 控制流图的连接点

- 连接点是将控制流结合起来的点
- 连接点使用包含多个输入和1个输出的圆圈表示

25/76

25

3.2.4 控制流图的基本结构



控制流图可以表达程序代码的基本结构

26/76

26

3.2.5 控制流图的特点

1. 有唯一的入口节点，即源节点，表示程序段的开始语句
2. 具有唯一的出口节点，即终止节点，表示程序段的结束语句
3. 节点由带有标号的圆圈表示，表示程序语句
4. 控制边由带箭头的直线或弧表示，代表控制流的方向
5. 包含条件的节点称为判定节点，由判定节点发出的边必须终止于一个节点

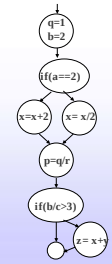
27/76

27

3.2.6 控制流图举例

- 代码的控制流图：

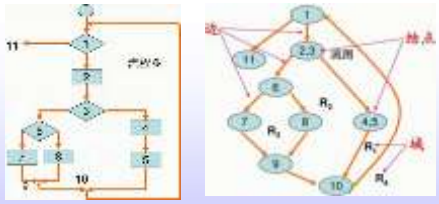
```
q=1;
b=2;
if(a==2)
      { x=x+2; }
else
      { x= x/2; }
p=q/r;
if(b/c>3)
      {z=x+y;}
```



28/76

28

流程图和控制流图



29/76

29

3.3 基于控制流的测试技术

- 在控制流测试中，定义了8种级别测试覆盖（coverage）
- 覆盖是指某个语句或分支被测试执行
- 覆盖（coverage）率是已经被测试的代码/分支的百分比

30/76

30

3.3.1 小于100%语句覆盖

- 0级：
- 测试与调试之间没有区别，除了支持调试，测试本身没有目的
- 缺陷可能会偶尔被发现，但是没有正式的努力去找到它们

31

3.3.2 100%语句覆盖 (Statement Coverage-SC)

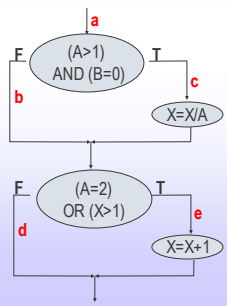
- 1级：100%语句覆盖
- 设计若干测试用例，运行被测程序，使得每一条可执行语句至少执行一次。
- 是最弱的逻辑覆盖准则，效果有限，必须与其它方法结合使用。

32

语句覆盖举例

设计满足语句覆盖的测试用例是:
输入：(A=2, B=0, X=4)
输出：(A=2, B=0, X=3)

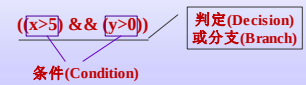
覆盖 ace



33

3.3.3 判定覆盖 (Decision Coverage-DC)

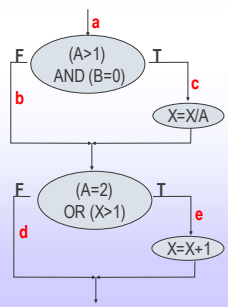
- 2级：100%判定覆盖
- 也叫分支覆盖（Branch Coverage - BC）。
- 判定覆盖就是设计若干个测试用例，运行被测程序，使得程序中每个判定的取真分支和取假分支至少经历一次。



34

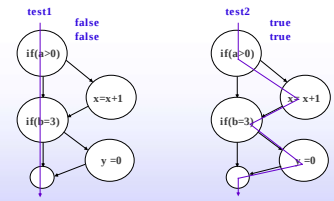
判定覆盖例1

或
设计满足判定覆盖的测试用例是:
输入：(A=2, B=1, X=1)
输出：(A=2, B=1, X=2)
覆盖 abe
输入：(A=3, B=0, X=3)
输出：(A=3, B=1, X=1)
覆盖 acd



35

判定覆盖例2



- 需要使用两个用例：a=-2，b=2和a=4，b=3达到判定覆盖

36

3.3.4 条件覆盖 (Condition Coverage-CC)

- 3级：100%条件覆盖
- 条件覆盖就是设计若干个测试用例，运行被测程序，使得程序中每个判定的每个条件的可能取值至少执行一次。
- 设计测试用例时，可以事先对所有条件的取值加以标记，然后设计相应测试用例。

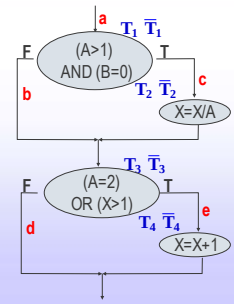
37/76

37

条件覆盖

或:
输入：(A=2, B=1, X=1)
输出：(A=2, B=1, X=2)
覆盖 abe $T_1\overline{T}_2T_3\overline{T}_4$

输入：(A=1, B=0, X=3)
输出：(A=1, B=0, X=4)
覆盖 abe $\overline{T}_1T_2\overline{T}_3T_4$



38/76

38

条件覆盖和判定覆盖的关系

- 条件覆盖通常优于判定覆盖，因为条件覆盖对每个条件进行了测试，而判定覆盖并不需要测试每个条件
- 这两种测试并不互相包含，有什么区别？
- 条件组合不一定会使判定的真假取值都出现一次

如：前面的条件组合$T_1\overline{T}_2T_3\overline{T}_4$ 和$\overline{T}_1T_2\overline{T}_3T_4$ ，使第一个判定始终为假，第二个判定始终为真。

39/76

39

3.3.5 判定－条件覆盖 (Decision Condition Coverage-DCC)

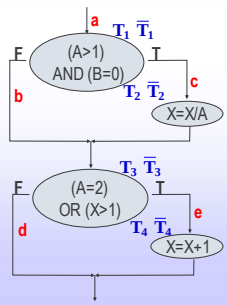
- 4级：100%判定覆盖+ 100%条件覆盖
- 设计足够的测试用例，使得判定中每个条件的所有可能取值至少执行一次，同时每个判定本身的所有可能判断结果至少执行一次。换言之，即是要求各个判定的所有可能的条件取值组合至少执行一次。

40/76

40

判定－条件覆盖

输入：(A=2, B=0, X=4)
输出：(A=2, B=0, X=3)
覆盖 ace[L1] $T_1T_2T_3T_4$
判定1:T 判定2:T
输入：(A=1, B=1, X=1)
输出：(A=1, B=1, X=1)
覆盖 abd[L2] $\overline{T}_1\overline{T}_2\overline{T}_3\overline{T}_4$
判定1: F 判定2:F



41/76

41

3.3.6 条件组合覆盖

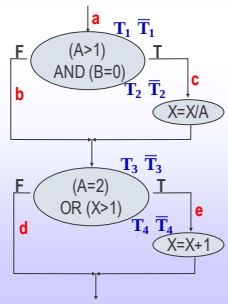
- 5级：设计足够测试用例，使每个判定的所有可能的条件取值组合至少执行一次。
- 也叫多条件覆盖（Multiple Condition Coverage - MCC）

例中，判定1的组合有
①T_1T_2 ②$\overline{T}_1T_2$ ③$T_1\overline{T}_2$ ④$\overline{T}_1\overline{T}_2$
判定2的组合有
⑤T_3T_4 ⑥$\overline{T}_3T_4$ ⑦$T_3\overline{T}_4$ ⑧$\overline{T}_3\overline{T}_4$

42/76

42

In(2, 0, 4), Out(2, 0, 3)
覆盖 ace[L1] $T_1T_2T_3T_4$
覆盖组合：① ⑤
In(2, 1, 1), Out(2, 1, 2)
覆盖 abe[L3] $T_1\overline{T}_2T_3\overline{T}_4$
覆盖组合：② ⑥
In(1, 0, 3), Out(1, 0, 4)
覆盖 abe[L3] $\overline{T}_1T_2\overline{T}_3T_4$
覆盖组合：③ ⑦
In(1, 1, 1), Out(1, 1, 1)
覆盖 abd[L2] $\overline{T}_1\overline{T}_2\overline{T}_3\overline{T}_4$
覆盖组合：④ ⑧



43

3.3.7 循环覆盖

- 6级：
- 当模块中包含循环时，路径将变得无限，需要将循环数量降低到较小的程度
- 一般设计的案例：执行0次循环，执行1次循环，执行典型的n次循环，执行最大数量m次循环，还可以测试m-1和m+1（循环边界测试）次循环

44

3.3.8 100%路径覆盖 (Path Coverage - PC)

- 7级：
- 100%路径覆盖是最高层次的测试
- 无论是程序中有没有出现的分支路径均要做测试，并且需要考虑多点分支和循环的情况
- 在存在循环的情况下由于路径太多造成测试几乎不可完成

45

3.3.9 测试覆盖情况统计

- 通过统计各种覆盖（代码覆盖、分支覆盖、基本块覆盖、……）情况，可以知道测试完整性；
- 一般使用工具进行统计
  - C++ Test可以显示覆盖情况
  - 一些专用的测试覆盖统计工具：Cobertura, EMMA，Jcoco等

46

提纲

- 白盒测试概述
- 基于数据流的白盒测试
- 基于控制流的白盒测试
- 基本路径测试（McCabe圈覆盖）
- Foster的ESTCA覆盖准则
- xUnit简介

47

4 基本路径测试（McCabe圈覆盖）

- Thomas.McCabe提出的一种白盒测试技术，使用圈复杂度（Cyclomatic Complexity），用于度量程序的复杂度
- 通过绘制被测试单元的控制流图来度量其圈复杂度
- 美国联邦航空局（FAA）管理的航空软件开发中，McCabe圈测试覆盖被用来作为单元测试的标准

48

4.1 圈复杂度的意义

- 为程序逻辑复杂性提供定量度量。其值定义了程序模块的**独立路径数量**，并提供了确保所有语句至少执行一次的测试数量的下界。

49

4.2 独立路径的含义

独立路径：从程序入口到出口的多次执行中，每次至少有一个语句集是新的，未被重复的。

路径1：1－11

路径2：

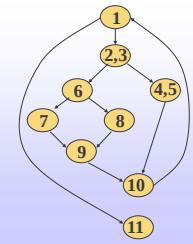
1－2,3－4,5－10－1－11

路径3：

1－2,3－6－8－9－10－1－11

路径4：

1－2,3－6－7－9－10－1－11

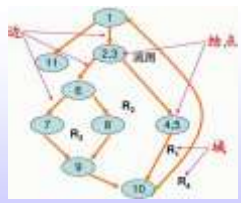


50

4.3 McCabe圈复杂度计算

- 计算圈复杂度的几种方法：

1. 流图中**区域**的数量
2. 流图中的**边数-结点数+2**
3. 流图中的**判定结点数+1**



51

举例

找出程序段的基本路径，并给出测试用例

```
1.  int m, n;
2.  int x=0;
3.  int y=0;
4.  if( m>n )
5.  {
6.      x = m-n;
7.  }
8.      else { x = n-m; }
9.
10. while( n<0 )
11. {
12.     y += n;
13.     n++;
14. }
15.
```

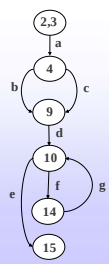
52

举例 找出程序段的基本路径，并给出测试用例

(1) 画出程序简化的控制流图

```
1.  int m, n;
2.  int x=0;
3.  int y=0;
4.  if( m>n )
5.  {
6.      x = m-n;
7.  }
```

```
8.      else { x = n-m; }
9.
10. while( n<0 )
11. {
12.     y += n;
13.     n++;
14. }
15.
```



53

(2) 计算圈复杂度(基本路径数)

图形中有2个封闭区域R1和R2

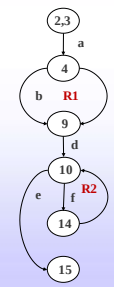
V(G) = 2 + 1 = 3

图形中有7条边(a,b,c,d,e,f,g)和6个节点

V(G) = 7 - 6 + 2 = 3

图形中有2个分支点④和⑩

V(G) = 2 + 1 = 3

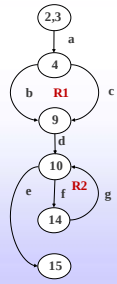


54

(3) 给出基本路径及测试用例

- 共有3条基本路径，只需要3个用例即可
- 基本路径和测试用例表

用例编号	基本路径	输入值	期望结果
1	abde	m=1, n=0	x=1, y=0
2	acde	m=-1, n=1	x=2, y=0
3	abdfge	m=1, n=-2	x=3, y= -3



55

提纲

- 白盒测试概述
- 基于数据流的白盒测试
- 基于控制流的白盒测试
- 基本路径测试（McCabe圈覆盖）
- Foster的ESTCA覆盖准则
- xUnit简介

56

5 Foster的ESTCA覆盖准则

- K.A.Foster发现，对于大小判定型的分支，不仅要判定大小，还需要判定相等，因为一般容易在判定大小的边界出错，即相等处
- 在常规的白盒覆盖测试中，往往将相等条件归于某一分支而忽略它本身，造成程序缺陷

```
/// rel可以是
  <, = 和 >
if( A rel B )
   { ...}
else
   { ...}
```

57

5.1 ESTCA准则的内容

- K.A.Foster提出了一种经验型的测试覆盖准则，称为（Error Sensitive Test Cases Analysis – 缺陷敏感测试用例分析）规则。
- ESTCA规则包括2条：

[规则1] 对于A rel B型的分支，应适当地选择A与B的值，使得A < B, A = B和A > B的情况分别出现一次

这是为了检测逻辑符号写错的情况，如将“A<=B”错写为“A<B”。

58

5.2 ESTCA准则的内容

[规则2] 对于A rel C（rel是 > 或 <，A是变量，C是常量）型的分支，当rel为 < 时，应适当地选择A的值，使：

A = C–M（M是距C最小的机器允许的正数，若A和C均为整型时，M = 1）

同样，当rel为>时，应适当地选择A，使：A = C + M

这是为了检测“差1”之类的错误

（如本应是“if (A > 1)”而错成“if (A > 0）”）

59

提纲

- 白盒测试概述
- 基于数据流的白盒测试
- 基于控制流的白盒测试
- 基本路径测试（McCabe圈覆盖）
- Foster的ESTCA覆盖准则
- xUnit简介

60

6. xUnit简介

- 白盒测试是主要的单元测试方法；
- 一般由开发人员实施白盒单元测试；
- 目前广泛使用的动态白盒测试工具是xUnit测试框架家族。

61/76

61

6. xUnit简介

单元测试工具分为2类：

- 自动化单元测试工具：可以自动产生白盒测试用例，自动执行测试并生成测试报告等。如C++Test、Visual Unit等；
- 单元测试框架：提供了一系列的类库和接口，需要用户自己编写测试代码来完成单元测试工作。如xUnit家族等；

62/76

62

6.1 xUnit简介

xUnit家族成员

- JUnit：测试Java语言代码
- CPPUnit：测试C++语言代码
- PyUnit：测试Python语言代码
- SUnit：测试SmallTalk语言代码
- vbUnit：测试VB语言代码
- utPLSQL：测试Oracle's PL/SQL语言代码
- CUnit：测试C语言代码

63/76

63

一个简单的Junit例子

Note: JUnit 4 syntax

```
public class Calc
{
  static public int add (int a, int b)
  {
    return a + b;
  }
}
```

```
import org.junit.Test;
import static org.junit.Assert.*;

public class CalcTest
{
  @Test public void testAdd()
  {
    assertTrue ("Calc sum incorrect",
      5 == Calc.add (2, 3));
  }
}
```

Test values

Printed if assert fails

Expected output

64/76

64

6.2 单元测试框架的作用

- 单元测试框架可以支持动态白盒单元测试过程中需要的功能；
- 动态白盒单元测试的一般过程：
  - 编写测试代码→执行单元测试→评估测试结果
- 对于"编写测试代码"的支持：提供测试代码基本类库
- 对于"执行单元测试"的支持：提供一个测试运行器(test runner、命令行或者GUI工具)，可以执行一个或所有的单元测试
- 对于"评估测试结果"的支持：执行测试后可以给出测试结果信息：执行了多少测试；多少/哪些测试失败；失败的代码位置；等等

65/76

65

白盒测试中的研究问题

- 如何自动生成高质量的白盒测试用例？
  - 带有反馈机制的随机测试用例生成，Randoop工具
  - 以路径覆盖做为优化目标的用例生成，Evosuite工具 http://www.evosuite.org/
  - 变异驱动的测试用例生成
- 测试的完整性问题，用例质量判断问题
  - 错误注入（变异测试）

66/76

66

The End

Any Question?

67