实验5

面向逻辑覆盖的白盒测试设计

覆盖率分析是评估测试充分性的重要方法之一，在白盒测试中广泛应用语句覆盖、分支覆盖等逻辑覆盖的覆盖率指标来评估测试过程对程序实体检查的充分性，引导构建更全面的测试集，以全面检查待测软件的质量。本实验尝试为指定程序设计满足覆盖要求的测试输入，通过工具来检查覆盖率，根据检查结果补充测试用例，直至覆盖提升。

一、实验目标

掌握白盒逻辑覆盖的概念，能够根据覆盖要求设计测试用例。体会各种逻辑覆盖准则之间的联系和差异，能够对用以实现某种覆盖的测试集进行优化。了解覆盖率收集工具的使用方法，能够在工具指引下提升测试覆盖。目标知识与能力如表 5-1 所示。

表 5-1　目标知识与能力

知　识	能　力
(1) 逻辑覆盖的概念 (2) 面向逻辑覆盖的测试生成方法	(1) 问题分析：能够将白盒测试问题抽象为图论和命题逻辑问题，进而分析覆盖要求，并优化覆盖设计 (2) 设计/开发解决方案：白盒测试用例设计 (3) 使用现代工具：使用覆盖分析工具并分析其局限性

二、实验内容与要求

使用逻辑覆盖测试方法测试如程序 5-1 所示的程序单元。其中，程序段每行开头的数字(1～7) 是对每条语句的编号。

程序 5-1　待测案例程序

```
    void doWork(int x, int y, int z) {
1       int k = 0, j = 0;
2       if((x > 3) && (z < 10)) {
3           k = x * y - 1;
4           j = sqrt(k);
        }
5       if((x == 4) || (y > 5))
6           j = x * y + 10;
7       j = j %3;
    }
```

要求：

（1）画出待测程序的程序流图。

（2）编写 Java 主函数，在其中单次或多次调用 doWork()方法，实现对该方法的测试，并尝试使对 doWork()方法的测试满足语句覆盖准则。

（3）使用覆盖率采集工具（如 Eclipse 自带的 EclEmma）获取测试的实际覆盖情况，并分析语句、分支覆盖（判定覆盖）的不足。

（4）根据前一步骤测试覆盖的不足，补充测试用例，实现语句和分支覆盖，并在覆盖率分析工具中确认。

（5）设计测试用例，使测试满足条件覆盖、判定/条件覆盖和 MC/DC 覆盖。

（6）对于前述针对各个覆盖准则的测试用例，试分析是否有更优化的测试用例组合，能够用更少的测试用例数量，实现同等程度的覆盖。

（7）思考覆盖采集工具对于白盒覆盖测试有哪些不足，并尝试给出改进建议。

三、实验环境

（1）JDK 11。

（2）Eclipse 2020 开发环境。

（3）Eclipse Java 自带的 EclEmma 覆盖信息收集插件。

四、评价要素

评价要素如表 5-2 所示。

表 5-2　评价要素

要　素	实 验 要 求
程序流程图	在既往教学中，发现许多同学不能针对 if-then、if-else、if-if 等多种情况绘制正确的流程图，因此此处仍需要仔细检查
覆盖提升流程	本实验的待测程序相对简单，重点是体验一个测试覆盖逐步提升的过程，了解在既有基础上，如何分析不足，提升覆盖
复杂覆盖准则	是否能够设计用例，实现 MC/DC 等复杂覆盖准则
测试优化	能够正确评判测试集是否可以优化，在不可优化时给出理由，在可以优化时给出优化方案
分析不足	能够开展批判性分析，认识到测试工具的不足；能够创新性地思考工具有哪些改进方向

问 题 分 析

1. 逻辑覆盖的概念

逻辑覆盖测试要求在测试集的执行过程中能够对程序中的逻辑控制实体及其取值形成某种程度的覆盖。不同覆盖准则对测试集执行的要求如表 5-3 所示。

表 5-3　逻辑覆盖的要求

覆盖准则	对测试用例集的要求
语句覆盖	保证程序中的每条语句都执行一遍
判定覆盖	保证每个判定取 True 和 False 至少一次
条件覆盖	保证每个判定中的每个条件取 True、False 至少一次
判定/条件覆盖	覆盖每个条件和由条件组成的判定的真假取值
条件组合覆盖	保证每个条件的取值组合至少出现一次
修正条件/判定覆盖(MC/DC)	保证每个条件取到其所有可能值各一次，保证每个条件独立影响判定结果至少一次
路径覆盖	覆盖程序中所有可能路径

有几种典型的思路可以用来构造测试用例，以实现某种覆盖。一种思路是试探法，采用随机测试等手段去逐个构造测试用例，执行所构造出的测试用例，考察它能够新覆盖哪些实体或其取值，检查加入该用例后测试集是否能够满足覆盖准则，如此不断尝试直到获得满足要求的测试集。该过程也可以吸收测试人员的经验、执行反馈等以提升实现覆盖的速度。

另一种思路是首先根据当前的测试覆盖情况，推测需要多执行哪一条程序路径，路径上各个条件表达式又需要满足怎样的真假要求，即可使扩增后的测试集满足覆盖准则。然后，将所有该路径上的条件表达式取值要求都逆推追溯到程序入口，将条件表达式叠加，构成“路径条件”，也即使得该路径可以得到执行的约束条件。求解路径条件，即可获得一个测试输入，补充该测试输入，可以比较直接地对测试集进行面向覆盖的针对性补充。该种测试生成对应一种经典的测试方法，称为基于符号执行[1]的测试。

2. 实验问题的解决思路与注意事项

本实验可以采用第二种测试构造思路来获得测试用例。也可以采用第一种方法来构造初始测试用例，再用第二种方法进行用例补充。

虽然 if-else 是最基本的程序结构，但是在实验中却发现有许多同学不能正确画出其程序流图。对于判断三角形是一般三角形、等边三角形、等腰三角形这样以 if-else 处理为主的问题，也发现仅有少部分学生能够一次性写对程序。主线路径往往没有问题，但是对于各类意外、边界情况，却总是难以一次性处理妥当。if-else 看似简单，实则极容易引发缺陷，在测试过程中需多加注意。

请在实验中特别注意思考如表 5-4 所示程序结构的不同。

表 5-4　不同的 if-else 语句结构

if 带 else	if 不带 else	双 if 串联	else if 结构	if-return

if(C1) {
　S1;
}
else{
　S2;
}
	if(C1) {
	　S1;
	}
	
	S2;
		if(C1) {
		　S1;
		}
		if(C2) {
		　S2;
		}
			if(C1) {
			　S1;
			}
			else if(C2) {
			　S2;
			}
				if(C1) {
				　S1;
				　return;
				}
				S2;

3. 难点与挑战

（1）测试用例构造：对于简单程序而言，构造测试用例去实现覆盖不难。但是如何在一个系统性的方法指引下以最高的效率去构造测试用例，是实验中亟待思考的问题。

（2）最优测试集：如何评判一个测试集是否已经最优，如果不是最优，更优秀的解是什么，这些是需要结合数学原理去探索的问题。

实验方案

1. 绘制程序流图

对于逻辑覆盖问题，在不够熟悉相关分析方法的情况下，可以先绘制代码的程序流图，理清程序的跳转关系。绘制程序流图的过程中，特别需要注意 if、while 等语句的真假分支方向，break、continue、return 语句的跳转目标，甚至 try-catch 语句的执行顺序等。

覆盖分析可以在源代码、字节码、中间代码等层面展开。本实验主要在源代码层实现覆盖，对于实验内容中指定的程序，根据语句间的控制转移关系，画出的程序流图如图 5-1 所示。

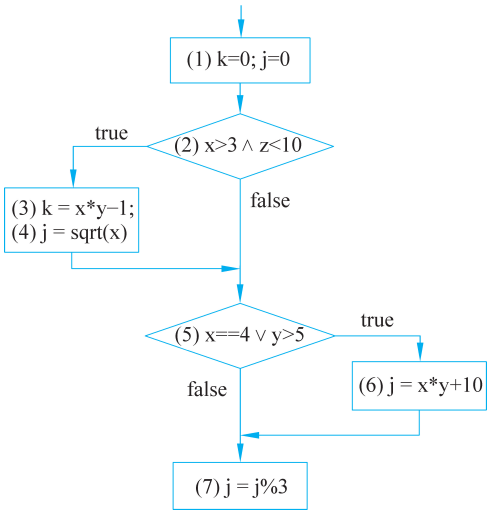

图 5-1　程序流图

2. 测试并实现语句覆盖

若要实现语句覆盖，比较经济的方式是使程序执行路径“1-2-3-4-5-6-7”，如此，只用一次执行，即可实现测试的语句覆盖。找到 if 语句 2 和 5，在该路径下，要求两处的判定 x＞3 && z＜10 和 x＝＝4 || y＞5 取值为真。从这些判定出发，结合 x、y、z 在路径“1-2-3-4-5-6-7”上的计算过程，可以逆推在程序入口处对于变量 x、y、z 取值的要求：（x＞3 && z＜10）&&（x＝＝4 || y＞5），由此可构造一个测试输入（x＝4，y＝6，z＝9），具体测试输入和各个判定与条件的真假情况如表 5-5 所示。在该输入下，语句 2 和语句 5 处的判定取值均为真，if 语句执行真分支。在语句 5 的判定中，条件 x＝＝4 为真，条件 y＞5 因为短路表达式计算规则，而未得到实际执行，未实现关于该条件的任何覆盖。

表 5-5　实现语句覆盖的测试输入

逻辑覆盖准则	测试输入	执行路径	x>3	z<10	x>3 && z<10	x==4	y>5	x==4 || y>5
语句覆盖	x=4,y=6,z=9	1 2 3 4 5 6 7	T	T	T	T	—	T

程序 5-2 展示了实现语句覆盖的测试程序 Test.java，其中以表 5-5 的输入在程序入口 main()中对被测方法 doWork()进行了调用。

程序 5-2　实现语句覆盖的测试程序 Test.java

```
package test;

public class Test {
    public static void main(String[] args) {
        DoWork d = new DoWork();
        d.doWork(4, 6, 9);
    }
}
```

3. 用 Eclipse 收集测试覆盖信息

在 Eclipse 中选中测试程序 Test.java，以 Coverage As→Java Application 的方式运行，可以收集到被测程序的覆盖率信息。切换到 Coverage 视图，单击视图右上角的“查看方式”按钮，以 Line Counters 方式展示覆盖率数据，可以看到对于 DoWork.java 程序，已经实现了 100%的语句覆盖，如图 5-2 所示。Eclipse 中会以代码高亮方式展示测试结果，有颜色高亮表明语句已经被测试过程全部或部分覆盖，部分覆盖的语句左侧有方块标记，如图 5-2 右侧所示的第 8 和 13 行，将鼠标悬停于部分覆盖的语句所在行，可以看到提示信息，告知尽管语句已被覆盖，但是仍有部分 if 分支未得到执行。

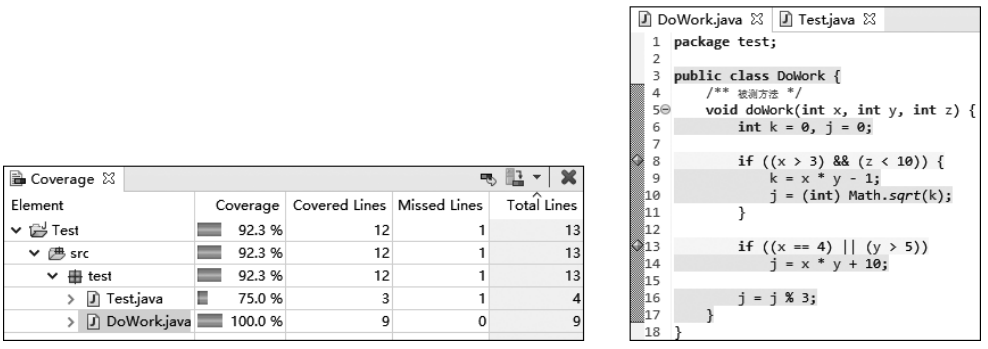

图 5-2　语句覆盖的信息展示

4. 补充测试输入，实现语句和分支的覆盖

在仅执行路径“1-2-3-4-5-6-7”的情况下，测试过程未覆盖到 if 语句的假分支。如果再执行一个路径“1-2-5-7”，从源代码角度来看，可以实现对程序中两个 if 语句假分支的覆盖。在该路径下，要求语句 2、5 两处的判定 x>3 && z<10 和 x==4 || y>5 取值为假。从这些判定取值出发，可以逆推在程序入口处对于变量 x、y、z 取值的要求：!(x>3 && z<10)

&&!(x==4 || y>5)，即(x<=3 || z>=10) && (x!=4 && y<=5)，由此构造一个测试输入(x=2,y=4,z=9)，如表 5-6 所示。

表 5-6 实现分支覆盖的测试输入

逻辑覆盖准则	测试输入	执行路径	x>3	z<10	x>3 && z<10	x==4	y>5	x==4 || y>5
分支覆盖	x=4,y=6,z=9	1 2 3 4 5 6 7	T	T	T	T	—	T
	x=2,y=4,z=9	1 2 5 7	F	—	F	F	F	F

在测试程序中增加形如 d.doWork(2,4,9)的调用，并用 Eclipse 执行程序，收集覆盖信息。切换到 Coverage 视图，以 Branch Counters 方式查看覆盖率数据，如图 5-3 所示。很遗憾，可以看到，Eclipse 中给出的覆盖率数据表明，对于程序 5-1，实际的 Covered Branches 仅达到 75.0%。Eclipse 的 EclEmma 插件认为当前并未实现分支覆盖。

是我们对分支覆盖的概念理解有误？还是 EclEmma 插件的覆盖率收集功能有缺陷？对于该问题进一步分析，可以发现，EclEmma 插件认为 doWork()方法中有 8 个跳转方向，而查看图 5-1 中的程序流图，发现源代码中仅有两个 if 语句，4 个跳转分支。是何种原因造成了这种差异？查阅 EclEmma 插件官方文档，可以发现，该工具基于 JaCoCo 工具来收集测试覆盖。再进一步查阅 JaCoCo 工具的官方文档，发现在其文档的 Coverage Counters 一节中有“All these counters are derived from information contained in Java class files which basically are Java byte code instructions…”。也就是说，JaCoCo 本质上是在 Java 字节码层面收集测试覆盖信息。

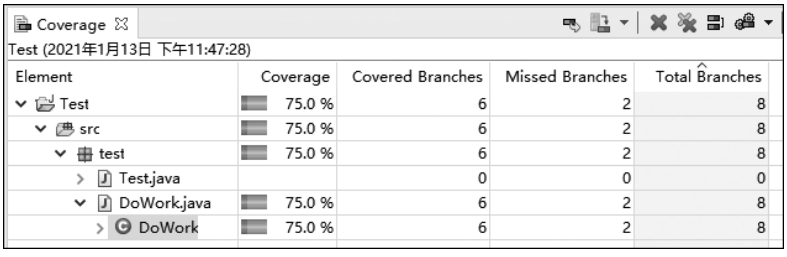

图 5-3 表 5-6 输入下 EclEmma 插件收集的分支覆盖率信息

用 Eclipse 打开被测程序 DoWork 的字节码文件 DoWork.class，可以看到其中 doWork()方法的字节码如图 5-4 所示。字节码中确实存在 4 个 if 指令，表明 EclEmma 插件的判定并没有问题。根据编译原理相关知识，在编译后的指令中，每个 if 跳转一般最多只检查一个条件。由此，从 doWork()方法的源代码看，大约每个条件检查会编译为一个 if 跳转。如表 5-6 所示，共有 4 个条件，对应 8 个字节码层面的跳转方向，测试用例覆盖了其中 6 个，z<10 的假和 y>5 的真未覆盖到，字节码层的覆盖率确实是 75.0%。

以上分析表明，表 5-6 中的测试输入，在源代码层可以实现 100%分支覆盖，而在字节码层可以实现 75%覆盖，两个结论均正确。

5. 设计测试用例，使测试满足条件覆盖、判定/条件覆盖和 MC/DC 覆盖

由上述分析可见，字节码层的分支覆盖基本对应源代码层的条件覆盖。表 5-6 中的测

```
void doWork(int x, int y, int z);
   0   iconst_0
   1   istore 4 [k]
   3   iconst_0
   4   istore 5 [j]
   6   iload_1 [x]
   7   iconst_3
   8   if_icmple 33
  11   iload_3 [z]
  12   bipush 10
  14   if_icmpge 33
  17   iload_1 [x]
  18   iload_2 [y]
  19   imul
  20   iconst_1
  21   isub
  22   istore 4 [k]
  24   iload 4 [k]
  26   i2d
  27   invokestatic java.lang.Math
       .sqrt(double) : double [16]
  30   d2i
  31   istore 5 [j]
  33   iload_1 [x]
  34   iconst_4
  35   if_icmpeq 43
  38   iload_2 [y]
  39   iconst_5
  40   if_icmple 51
  43   iload_1 [x]
  44   iload_2 [y]
  45   imul
  46   bipush 10
  48   iadd
  49   istore 5 [j]
  51   iload 5 [j]
  53   iconst_3
  54   irem
  55   istore 5 [j]
  57   return
```

图 5-4　doWork()方法的 Java 字节码

试输入显然并未实现源代码层的条件覆盖。

为实现条件覆盖，需要构造测试输入，使得 z<10 的假和 y>5 的真得到覆盖。在短路机制下，欲使条件 z<10、y>5 得到检查，x>3 必须为真，x==4 必须为假，可以据此构造一个测试输入(x=5，y=6，z=10)，结合输入(x=4，y=6，z=9)和(x=2，y=4，z=9)，可以实现源代码层的条件覆盖。具体测试输入组合如表 5-7 所示，该输入组合从表 5-6 扩增而来，显然同时也满足判定/条件覆盖准则。

表 5-7　实现条件覆盖的测试输入

逻辑覆盖准则	测试输入	执行路径	x>3	z<10	x>3 && z<10	x==4	y>5	x==4 || y>5
条件覆盖	x=4，y=6，z=9	1 2 3 4 5 6 7	T	T	T	T	—	T
	x=2，y=4，z=11	1 2 5 7	F	—	F	F	F	F
	x=5，y=6，z=10	1 2 5 6 7	T	F	F	F	T	T

MC/DC 覆盖准则要求覆盖程序的入口和出口，并且每个条件的取值可以单独影响判定的结果，使其发生真假变化。对于表 5-7 的测试输入，显然已经覆盖了被测方法 doWork()的唯一入口和出口(语句 1 前是入口、语句 7 后是出口)。但条件 x>3 未在条件 z<10 取稳定值的情况下，通过自身变化影响判定 x>3 && z<10 的结果。条件 x==4 未在条件 y>5取稳定值的情况下，通过自身变化影响判定 x==4 || y>5 的结果。为此，需要补充 z<10为真情况下，x>3 为假的测试输入，并补充 y>5 为假的情况下，x==4 为真的测试输入。以上两个补充，一个要求!(x>3)，也即 x<=3，一个要求 x==4，两者不可能同时满足，因此需要补充两个新测试输入。具体补充结果如表 5-8 所示，由于补充的输入下 z<10 和 y>5 并未实际执行，因此表中用标记“—”表示，这里用形式“(F)”表示假如被执行情况

下条件的真假。

表 5-8 实现 MC/DC 覆盖的测试输入

逻辑覆盖准则	测试输入	执行路径	x>3	z<10	x>3 && z<10	x==4	y>5	x==4 || y>5
MC/DC 覆盖	x=4,y=6,z=9	1 2 3 4 5 6 7	T	T	T	T	-(T)	T
	x=2,y=4,z=11	1 2 5 7	F	-(F)	F	F	F	F
	x=5,y=6,z=10	1 2 5 6 7	T	F	F	F	T	T
	x=3,y=6,z=9	1 2 5 6 7	F	-(T)	F	F	T	T
	x=4,y=4,z=9	1 2 3 4 5 6 7	T	T	T	T	-(F)	T

6. 分析测试集是否可优化

分析前述满足相应覆盖准则的测试输入组合，发现对于语句覆盖，表 5-5 中一个输入组合已经是最少，故而无法再优化。

即使只有一个分支语句，无论如何也要两个输入组合才能实现分支覆盖，故而表 5-6 的测试输入数量也无法减少。

对于条件覆盖，在存在短路机制的情况下，对于类似 A && B 的运算，至少要 3 个真假组合才能覆盖到其真假情况，并确保条件得到实际的检查。因此表 5-7 中条件覆盖和判定/条件覆盖的测试输入数量已经达到最少。

MC/DC 覆盖相对判定/条件覆盖增加了一个条件独立影响判定结果的要求，尽管如此，对于两个判定中的 4 个条件而言，覆盖条件取值的真假组合有很多种，同一变量上的条件还有所关联，不能保证表 5-8 面向 MC/DC 覆盖的 5 个测试输入组合已经达到最优。先考虑单独的判定 x>3 && z<10，只需 3 个输入组合即可实现其 MC/DC 覆盖，如表 5-9 所示。同样，对于判定 x==4 || y>5，最少也只需 3 个输入组合即可实现其 MC/DC 覆盖。将这两个判定的相关条件取值恰当地组合在一起，可以降低实现整个 doWork() 方法 MC/DC覆盖所需的测试输入数量。表 5-9 给出了优化后的、能够满足 MC/DC 覆盖准则的测试输入组合。该表仅用 3 个输入组合即实现了 MC/DC 覆盖。

表 5-9 实现 MC/DC 覆盖的优化测试输入组合

逻辑覆盖准则	测试输入	执行路径	x>3	z<10	x>3 && z<10	x==4	y>5	x==4 || y>5
MC/DC 覆盖	x=5,y=6,z=9	1 2 3 4 5 6 7	T	T	T	F	T	T
	x=2,y=4,z=9	1 2 5 7	F	-(T)	F	F	F	F
	x=4,y=4,z=10	1 2 5 6 7	T	F	F	T	-(F)	T

当然，本实验并未证明目前所给出的方案已经是最优测试输入组合，最优测试输入组合也可能存在很多组。未来可以考虑对此开展更细致的研究。

7. 局限分析

在使用白盒覆盖工具辅助开展测试设计的过程中，发现存在以下一些不足，给测试设计带来一些困扰，未能进一步提高测试效率。

（1）未能在源代码层面收集测试覆盖，很多收集到的测试覆盖信息与直观理解不是太吻合，给测试设计带来困扰。测试者不清楚是不是确实如工具所汇报的存在测试不充分问题。

（2）覆盖信息是在静态的代码层展示，有些语句会执行多次，从多个途径经多个程序路径被执行到。当程序比较复杂时，测试者难以理清为什么一个语句被覆盖到，另一个语句又没有被覆盖。在补充生成测试输入的过程中，需要花费比较大的精力来理清程序执行，针对性地设计新输入。

（3）覆盖信息展示时，没有能够展示相关数据取值，不便于带入取值来分析条件、判定的执行情况。

（4）不支持分析测试是不是存在冗余，例如，对于 doWork()方法，其入口参数的取值组合数量对于实现某种覆盖准则是不是过多？

（5）无法针对代码自动构造实现某种覆盖的输入数据组合，而从实验中人工设计测试输入的经验来看，一些输入的构造似乎可以通过算法推理实现。

若要改进覆盖分析工具，一个建议是调研广大用户最迫切的功能需求是什么，分析哪些是相对容易实现的，如此可能以较小代价迅速改善工具的使用体验。

附件资源

（1）被测程序代码。

（2）测试用例代码。

参考文献

[1] Baldoni R, Coppa E, D'elia D C, et al. A Survey of Symbolic Execution Techniques [J]. ACM Computing Surveys, 2018, 51(3), Article No. 50: 1-39.