第1篇 单元测试与集成测试实验

单元测试是软件开发和系统测试的基础，只有每个单元模块得到了充分的测试，系统测试才能相对轻松地完成，否则系统测试变得没有止境，缺陷永远找不完。作为开发人员，需要对自己所写的代码负责，也需要做单元测试。单元测试一般和编程同步进行，写完一段代码，就要进行单元测试。

相对来说，软件的单元规模很小，可以精确、有效、完整地进行测试，也比较容易进行测试覆盖率的分析，通过不断改进测试，最终可以达到所需的测试覆盖率。从测试充分性看，单元测试可以更好地帮助我们保证软件产品质量。

单元测试，除了人工的代码评审，其他的测试(代码静态分析、动态测试等)都属于自动化测试范畴，通过工具和脚本自动完成。单元测试的实验，从代码行覆盖/判定覆盖开始，逐步深入到条件覆盖、条件/判定覆盖、组合覆盖和 MC/DC 覆盖、基本路径覆盖等，并结合 PMD、JUnit、CppUnit 等单元测试工具，完成实际代码的测试，其中包括测试覆盖率的度量和分析，并把 TDD/ATDD/BDD、类、包的测试和 Mock 技术的运用等更复杂的单元测试留给大家练习与思考。

本篇主要开展单元测试实验。通过这些实验，提高同学们的单元测试能力，并巩固结构化测试方法的应用。

◇ 实验 1：语句和判定覆盖测试设计

◇ 实验 2：条件覆盖和条件组合覆盖测试设计

◇ 实验 3：修正条件/判定覆盖测试设计

◇ 实验 4：基于 JUnit 的单元测试

◇ 实验 5：基于 CppUnit 的单元测试

◇ 实验 6：基于 JavaScript 的单元测试

◇ 实验 7：基于 PMD 的静态测试

◇ 实验 8：基于 Jenkins 的集成测试

实验1 语句和判定覆盖测试设计

1.1 实验目的

(1) 巩固所学的语句覆盖和判定覆盖测试方法；
(2) 提高运用语句覆盖和判定覆盖测试方法的能力。

1.2 实验前提

(1) 掌握语句覆盖和判定覆盖的基本方法、概念；
(2) 熟悉程序语言的逻辑结构与基础知识；
(3) 选择一段程序语言。

1.3 实验内容

以保险产品投保为实例，针对保险产品投保业务逻辑代码进行分析，运用语句覆盖和判定覆盖法进行测试用例设计。

某个人税收优惠型保险产品 A/B1/B2/C 款承保规则：

(1) 凡16周岁以上且投保时未满法定退休年龄的(男性为59周岁、女性为54周岁，后续将随国家相关法规做相应调整)，适用商业健康保险税收优惠政策的纳税人，可作为本合同的被保险人。保险公司根据被保险人是否参加公费医疗或基本医疗保险确定适用条款。

(2) 被保人为健康体，或者参加医疗保险的，可选择 A 款、B1 款或 B2 款。

(3) 未参加公费医疗的非健康体(有既往症)只能选择 C 款。

以下为个人税收优惠型保险产品承保的部分伪代码实现：

```
If (性别 = '男' and 16 <年龄< 59 or 性别 = '女' and 16 < age < 54)
{
    If (被保人健康属性为正常 or 有医疗保险)
    {可选择险种类型为 A 或 B1 或 B2 的险种、份数为 1}
Else
    {可选择险种类型为 C 的险种、份数为 1}
    EndIf
}
```

```
Else
{提示"不能承保"}
EndIf
```

1.4 实验环境

(1) 首先要让学生了解保险产品投保业务场景,能够模拟操作保险产品的承保流程;
(2) 能够将业务场景与代码逻辑关系对应;
(3) 根据代码画出程序流程图,并分析各判定节点;
(4) 根据代码流程图分析出判定条件与真假取值。

1.5 实验过程简述

(1) 明确被测试对象使用的测试方法;
(2) 小组讨论业务场景并进行分析;
(3) 测试实施工作安排;
(4) 评审程序流程图和测试用例;
(5) 执行测试,根据测试用例代入各条件测试数据,给出测试结果。

1.6 测试过程实施

1. 测试分析

(1) 根据保险产品的承保业务描述,分析产品承保流程,包括主流程、分支流程以及正常流程、异常流程。

(2) 模拟保险产品承保场景:触发允许产品承保的条件,不同条件是否走不同的承保流程。

(3) 数据项检查:数据项的计算规则,数据项后台判断逻辑。

2. 测试设计

根据产品承保代码,设计出程序流程图,并对程序流程图做节点标记,分析图 1-1 所示的两个判定:

```
判定 A:(性别 = "男" AND 16 <年龄< 59)OR(性别 = "女" AND 16 <年龄< 54)
判定 B:健康体 OR 有医疗保险
```

3. 测试设计

根据业务场景与流程逻辑判定,运用语句覆盖法进行用例设计。

语句覆盖是一个比较弱的逻辑覆盖标准,通过选择足够多的测试用例,使得被测试程序中的每个语句至少被执行一次。根据如图 1-1 所示的流程图,为使程序中的每个语句至少执行一次,只需设计两个测试用例,覆盖语句 A、B、C、E,即覆盖判定 A“成立”、判定 B“成立”或“不成立”各被覆盖一次,如表 1-1 所示。

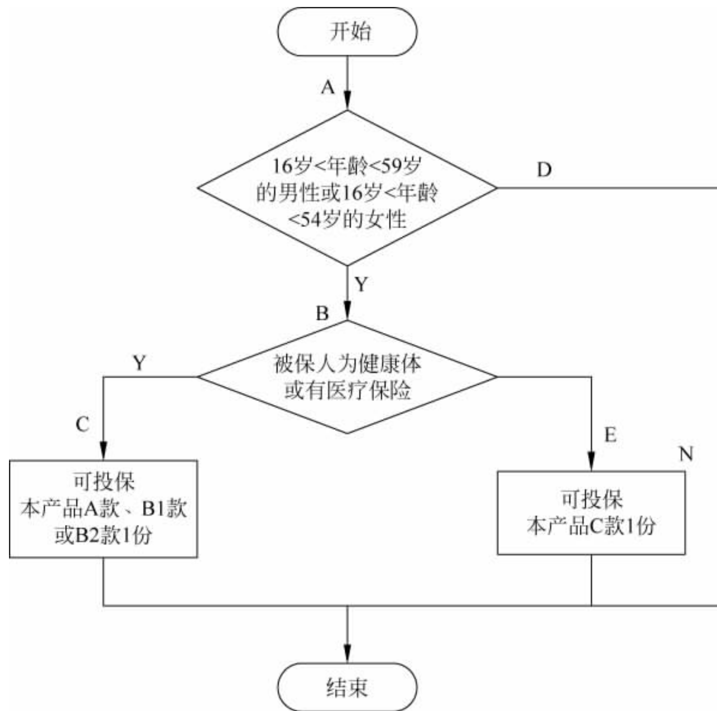

图 1-1　流程图

表 1-1　语句覆盖测试用例设计

测试用例名称	测试用例描述	测试路径
CASE1	投保成功：年龄 20，男性，健康体、有医疗保险	ABC
CASE2	投保成功：年龄 20，男性，非健康体且没有医疗保险	ABE

接下来我们运用判定覆盖法来进行用例设计。判定覆盖又称为分支覆盖，判定覆盖语句覆盖的标准稍强一些，它是指通过设计足够多的测试用例，使得被测试程序中的每个判定(即上述判定 A、判定 B)都获得一次"真""假"值，如表 1-2 所示。

表 1-2　判定覆盖测试用例设计

测试用例名称	测试用例描述	覆盖判定
CASE3	投保成功：年龄 20，男性，健康体，有医疗保险	判定 A ="真"
		判定 B="真"
CASE4	投保不成功：年龄 15，男性	判定 A ="假"
CASE5	投保成功：年龄 20，男性，健康体，没有医疗保险	判定 A ="真"
		判定 B="假"

4. 测试结果分析

从本实验可看出，语句覆盖实际上是很弱的，CASE1、CASE2 可以满足语句覆盖，但如果第 2 个条件语句中 OR 写成了 AND，CASE1、CASE2 都不能发现它。

"判定覆盖"比"语句覆盖"严格，因为如果每个分支都执行过了，则每个语句也就执行过

了。但是,“判定覆盖”还是不够的,例如,CASE3～CASE5 未能检查 AB 分支中女性被保人的承保情况。

1.7 实例练习

(1) 程序实例,计算个人所得税。

```
#include <stdio.h>
int main ()
{
    double dSalary,dTax = 0,dNetIncome = 0;
    double dValue;
    printf("请输入您本月的收入总额(元): ");
    scanf("%lf", &dSalary);
      dValue = dSalary - 3500;        //在起征点基础上考虑纳税
    if(dValue > 0.0)
    {
        if(dValue <= 1500)
            dTax = dValue * 0.03 - 0.0;
        else if(dValue <= 4500)
            dTax = dValue * 0.10 - 105.0;
        else if(dValue <= 9000)
            dTax = dValue * 0.20 - 555.0;
        else if(dValue <= 35000)
            dTax = dValue * 0.25 - 1005.0;
        else if(dValue <= 55000)
            dTax = dValue * 0.30 - 2755.0;
        else if(dValue <= 80000)
            dTax = dValue * 0.35 - 5505.0;
        else
            dTax = dValue * 0.45 - 13505.0;
    }
    dNetIncome = dSalary - dTax;
    printf("您本月应缴个人所得税 %.2lf 元,税后收入是 %.2lf 元.\n", dTax, dNetIncome);
    return 0;
}
```

(2) 请根据程序实例(表 1-3),设计语句和判定覆盖的测试案例。

表 1-3　条件分析

	条　　件	下一步	产品A款	产品B1款	产品B2款	产品C款	不承保
A	(性别="男"并且 16<年龄<59)or (性别="女"并且 16<年龄<54)	B					
	年龄小于 16 的男性	F					●
	年龄小于 16 的女性	F					●
	年龄大于 59 的男性	F					●
	年龄大于 54 的女性	F					●

续表

	条　件	下一步	产品 A 款	产品 B1 款	产品 B2 款	产品 C 款	不承保
B	是健康体或者有医疗保险公费	D	1 份				
				1 份			
					1 份		
	有既往症且医疗保险没有	H				1 份	

实验 2　条件覆盖和条件组合覆盖测试设计

2.1　实 验 目 的

（1）巩固所学的条件覆盖、条件组合覆盖测试方法；

（2）提高运用条件覆盖、条件组合覆盖法的能力。

2.2　实 验 前 提

（1）掌握逻辑覆盖的基本方法、概念；

（2）熟悉程序语言的逻辑结构与基础知识；

（3）选择一段程序语言。

2.3　实 验 内 容

以银行内部转账为实例，针对内部转账业务逻辑代码进行分析，运用条件覆盖进行测试用例设计。

内部转账用于处理发起户口号和接收户口号都是内部账户的系统内资金转账业务，主要用于财务资金的划拨、未实现自动清算业务的清算资金的划拨。

（1）内部转账发起是指：发起行发出内部资金交易，并换人复核，满足条件时需会计主管授权。

（2）内部转账接收是指：内部资金交易接收方根据接收方确认方式，对交易进行接收经办，满足条件的需复核或授权。

确定接收方的入账流程，“确认方式”分为以下三种：

（1）不需接收方确认，即发起方发起后自动记发起方和接收方的一套账务，接收方无须再做接收动作。

（2）需接收方确认，即接收方接收时不能更改接收信息，只能依据发起方输入的信息入账或退发起方。以目前的处理方式，接收经办→入账（金额小于 100 万元），大于 100 万元时为接收经办＋接收授权→入账。

（3）需接收方经办，即接收方接收时可以更改接收信息，执行入账或退发起行。以目前的处理方式，接收经办＋接收复核→入账（金额小于 100 万元），大于 100 万元时为接收经办＋接收复核＋接收授权→入账。

内部转账权限控制如表 2-1 所示。

表 2-1　内部转账权限控制

操　　作	条　　件	经办	复核	授权
内部转账发起	100 万元以下	√	√	
	100 万元以上	√	√	√
内部转账接收	“确认方式”为“2”,100 万元以下	√		
	“确认方式”为“2”,100 万元以上	√		√
	“确认方式”为“3”,100 万元以下	√	√	
	“确认方式”为“3”,100 万元以上	√	√	√

以下为银行内部转账控制的部分伪代码实现：

```
If( 判定 1: 转账金额 > 100W) {
    调用"内部转账发起复核";
    调用"内部转账发起授权";
    If( 判定 3: "确认方式" == 1) {
        抛出异常"确认方式不符合业务流程"
    }
    Else If( 判定 3: "确认方式" == 2) {
        调用"内部转账接收经办";
        调用"内部转账接收授权";
        接收确认
    }
    Else If( 判定 3: "确认方式" == 3) {
        调用"内部转账接收经办";
        调用"内部转账接收复核";
        调用"内部转账接收授权";
        接收确认
    }
    Else {
        抛出异常"确认方式不符合业务流程"
    }
    End If
}
Else If (判定 1: 0 < 转账金额 <= 100W) {
  If( 判定 2: "确认方式" == 1) {
    调用"内部转账接收确认";
    接收确认
    }
    Else If( 判定 2: "确认方式" == 2) {
        调用"内部转账接收经办";
        调用"内部转账接收确认";
        接收确认
    }
    Else If( 判定 2: "确认方式" == 3) {
        调用"内部转账接收经办";
        调用"内部转账接收复核";
        调用"内部转账接收确认";
```

```
            接收确认
        }
        Else {
            抛出异常"确认方式不符合业务流程"
        }
        End If
}
Else If (判定 1: 转账金额 <= 0) {
        抛出异常"输入金额有误,请重新输入"
}
End if
```

2.4 实验环境

（1）首先要让学生了解银行内部转账业务，能够模拟操作转账流程；

（2）能够将业务场景与代码逻辑关系对应；

（3）根据代码画出程序流程图，并分析各判定节点；

（4）根据代码流程图分析出条件覆盖、条件组合覆盖。

2.5 实验过程简述

（1）明确被测试对象使用的测试方法；

（2）小组讨论业务场景并进行分析；

（3）测试实施工作安排；

（4）评审程序流程图和测试用例；

（5）执行测试，根据测试用例代入各条件测试数据，给出测试结果。

2.6 实验过程实施

1. 测试分析

（1）根据银行内部转账业务描述，分析内部转账流程，包括主流程、分支流程以及正常流程、异常流程。

（2）模拟内部转账场景：触发内部转账的条件，不同条件是否走不同的转账流程。

（3）数据项检查：数据项的计算规则，数据项后台判断逻辑。

2. 测试设计

根据内部转账业务需求，设计出程序流程图，如图 2-1 所示，并对程序流程图做节点标记，分析流程图的判定条件与结果。

3. 测试执行

根据业务场景与流程逻辑判定，运用条件覆盖法进行用例设计。

条件覆盖即设计足够多的测试用例，运行被测程序，使得每一判定语句中每个逻辑条件的可能取值至少满足一次。条件覆盖率的公式是：条件覆盖率＝被评价到的条件取值的数

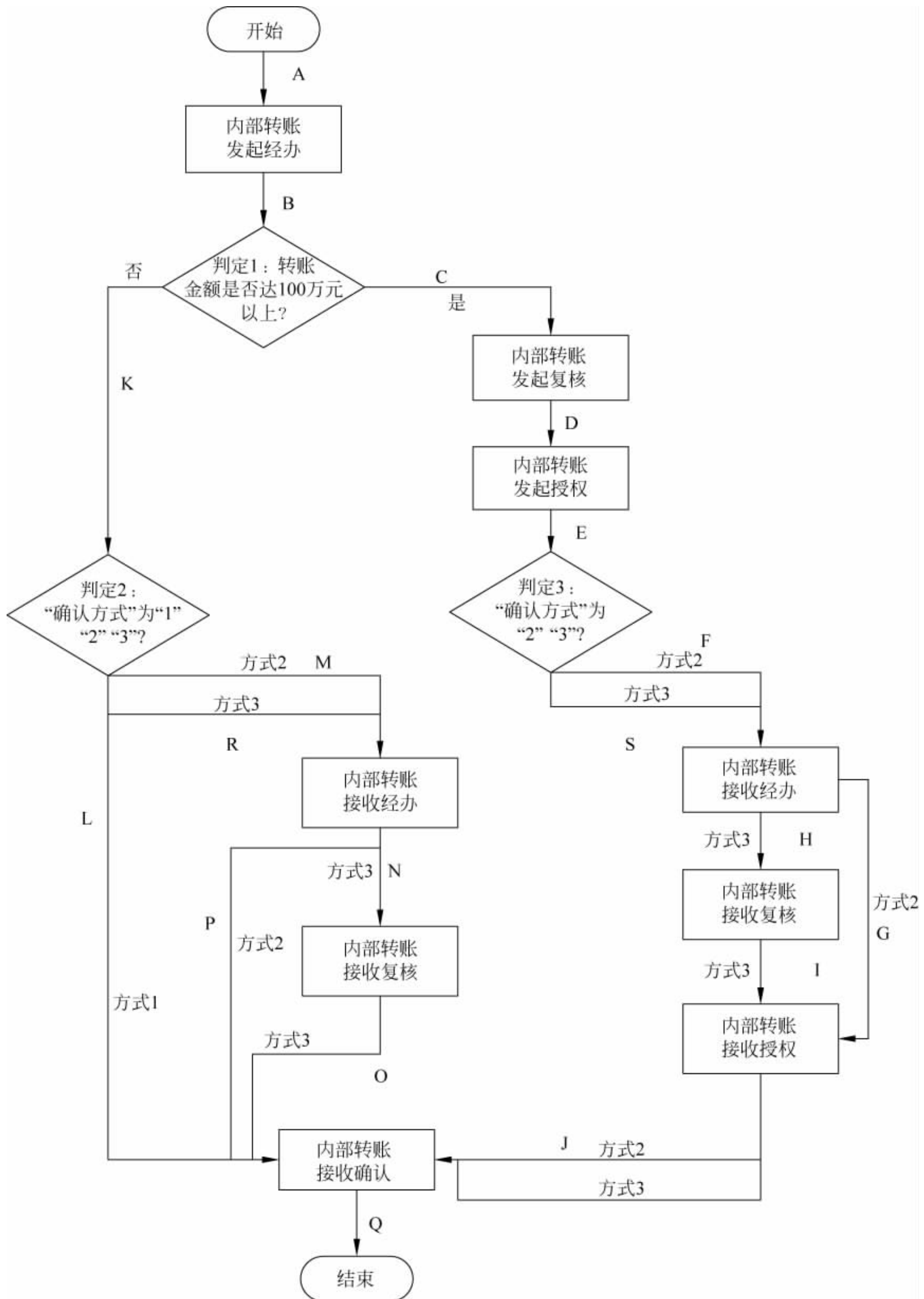

图 2-1　程序流程图

A～Q 为测试路径编号，在下面的测试用例分析中将根据测试路径编号确定测试用例的业务流向。

量/条件取值的总数×100％。具体地说，就是在各种条件中，不考虑条件组合的因素，对每一个条件变量分别只取真假值一次，使得被测试程序中的每个条件取值至少被覆盖一次。

条件组合覆盖是通过设计足够多的测试用例，使得被测试程序中每个判断的所有可能条件取值的组合至少出现一次。

注意：

（1）条件组合只针对同一个判断语句内存在多个条件的情况，让这些条件的取值进行笛卡儿乘积组合。

（2）不同的判断语句内的条件取值之间无须组合。

（3）对于单条件的判断语句，只需要满足自己的所有取值即可。

测试的依据是需求与设计文档，根据程序流程图实现。

（1）条件覆盖

银行内部转账流程在不考虑判定、仅考虑条件分支的情况下，条件分支数为5，即T1～T5。在条件覆盖中只考虑每个判定语句中的每个表达式，没有考虑各个条件分支。

根据图2-1所示的流程图，标记出节点。根据条件覆盖方法来进行分析，得到如表2-2所示的符合条件覆盖标准的测试用例。

表2-2　符合条件覆盖标准的测试用例

测试用例名称	测试用例描述
CASE 1	覆盖条件：转账金额＞ 100W
CASE 2	覆盖条件：转账金额＜＝ 100W
CASE 3	覆盖条件："确认方式"＝＝ 1
CASE 4	覆盖条件："确认方式"＝＝ 2
CASE 5	覆盖条件："确认方式"＝＝ 3
CASE 6	覆盖条件："确认方式"＜＞ 1、2、3

S(2）条件组合覆盖

对于判定1：

① 条件 转账金额＞ 100W　取真为T1

② 条件 转账金额＜＝ 100W　取假为F1

对于判定2：

① 条件"确认方式"＝＝ 1　取真为T2

② 条件"确认方式"＝＝ 2　取真为T3

③ 条件"确认方式"＝＝ 3　取真为T4

④ 条件T2、T3和T4都不成立　取假为F2

对于判定3：

① 条件"确认方式"＝＝ 2　取真为T5

② 条件"确认方式"＝＝ 3　取真为T6

③ 条件T5和T6都不成立　取假为F3

通过设计足够多的测试用例，使得被测试程序中的每个判断的所有可能条件取值的组合至少出现一次。在这个银行内部转账流程上，判定1的条件和判定2、3中的条件分别构

成组合。由于业务特定的逻辑，其组合简化为7个，而不是14个。

① 判定1的条件T1和判定3中的各个条件构成组合，即3个组合，而不是2×3=6个组合；

② 判定1的条件F1和判定2中的各个条件构成组合，即4个组合，而不是2×4=8个组合。

因此根据条件组合覆盖，总共有7个测试用例完成组合覆盖，如表2-3所示。这里不考虑异常情况，如转账金额<=0的情况。遇到这种情况会直接异常退出，也无法进入下一个判定2或判定3，和组合也没关系。

表2-3　符合条件组合覆盖度量标准的测试用例

测试用例名称	测试用例描述
CASE 1	覆盖T1+T5：转账金额>100W &"确认方式"==2
CASE 2	覆盖T1+T6：转账金额>100W &"确认方式"==3
CASE 3	覆盖T1+F3：转账金额>100W &"确认方式"!=2 or 3
CASE 4	覆盖F1+T2：0<转账金额<=100W &"确认方式"==1
CASE 5	覆盖F1+T3：0<转账金额<=100W &"确认方式"==2
CASE 6	覆盖F1+T4：0<转账金额<=100W &"确认方式"==3
CASE 7	覆盖F1+F2：0<转账金额<=100W &"确认方式"!=1 or 2 or 3

4. 测试结果分析

从实验2项目案例中可以看出，条件覆盖仅考虑单个条件取真或取假一次，覆盖度相对较弱。如果想增强覆盖度，可以将本实验的条件覆盖和实验1的判定覆盖结合起来，构成更强的覆盖，即条件-判定覆盖。如果还想达到更高质量的要求，可以设计足够的测试用例达到组合覆盖测试。但条件组合的测试有些冗余，效率偏低。在这种情况下就要考虑到修正条件/判定覆盖来设计测试用例。

2.7　实例练习

(1) 程序实例：企业发放的奖金根据利润提成。

```
#include <stdio.h>

main()
{
long int i;
int bonus1,bonus2,bonus4,bonus6,bonus10,bonus;
scanf("%ld",&i);
bonus1 = 100000 * 0.1;bonus2 = bonus1 + 100000 * 0.75;
bonus4 = bonus2 + 200000 * 0.5;
bonus6 = bonus4 + 200000 * 0.3;
bonus10 = bonus6 + 400000 * 0.15;

  if(i <= 100000)
  bonus = i * 0.1;
```

```
  else if(i<=200000)
      bonus=bonus1+(i-100000)*0.075;
    else if(i<=400000)
        bonus=bonus2+(i-200000)*0.05;
      else if(i<=600000)
          bonus=bonus4+(i-400000)*0.03;
        else if(i<=1000000)
            bonus=bonus6+(i-600000)*0.015;
          else
                bonus=bonus10+(i-1000000)*0.01;
printf("bonus=%d",bonus);
}
```

（2）请根据以上程序设计条件、判定条件、条件组合判定覆盖方法测试用例。

实验 3　修正条件/判定覆盖测试设计

3.1　实 验 目 的

（1）巩固所学的修正条件/判定覆盖测试；
（2）提高运用修正条件/判定覆盖测试的能力。

3.2　实 验 前 提

（1）掌握逻辑覆盖的基本方法、概念；
（2）熟悉程序语言的逻辑结构与基础知识；
（3）选择一种程序语言。

3.3　实 验 内 容

以信用卡还款为实例，见图 3-1，针对信用卡还款业务逻辑代码进行分析，运用修正条件/判定覆盖法进行测试用例设计。信用卡还款是网上银行系统和第三方支付平台的常见功能。登录第三方支付平台，选择信用卡还款模块，进入信用卡还款页面。在信用卡还款页面的第二步操作页面，验证储蓄卡是否有效并进行还款。信用卡还款业务流程描述如下。

（1）在“填写还款信息”页面，输入信用卡卡号、持卡人姓名，单击“确定付款”按钮，进入“使用储蓄卡付款”页面；

（2）在“使用储蓄卡还款”页面，输入储蓄卡卡号、持卡人姓名、单击“下一步”按钮，进入“还款详细”页面；

（3）在“还款详细”页面，在“还款类型”下拉框中选择“全部还款”或“分期还款”，单击“确定还款”按钮完成还款。

以下为通过第三方支付平台进行信用卡还款的部分伪代码实现。

```
If (银行卡号 is 有效 AND 姓名 is 有效 AND 余额> 0) {
    If (全额还款 OR 分期还款) {
        If (还款金额 ≥ 指定金额) then
            打印"还款成功"
        else
```

```
                打印"余额不足"
            }
        else
              打印"返回"
        }
    else
        打印"卡号错误或卡号姓名不一致或余额≤0"
    endIf
```

图 3-1　信用卡还款界面

3.4　实 验 环 境

(1) 首先要让学生了解信用卡还款业务场景,能够模拟操作信用卡还款流程;
(2) 能够将业务场景与代码逻辑关系对应;
(3) 根据代码画出程序流程图,并分析各判定节点;
(4) 根据代码流程图分析出判定条件与真假取值。

3.5　实验过程简述

(1) 明确被测试对象使用的测试方法;
(2) 小组讨论业务场景并进行分析;
(3) 测试实施工作安排;
(4) 评审程序流程图和测试用例;
(5) 执行测试,根据测试用例带入各条件测试数据,给出测试结果。

3.6 测试过程实施

1. 测试分析

（1）根据信用卡业务描述，分析信用卡还款流程，包括主流程、分支流程以及正常流程、异常流程。

（2）模拟信用卡还款场景：触发信用卡还款的条件，不同条件是否走不同的还款流程。

（3）信用卡还款数据项检查：数据项的计算规则；数据项后台判断逻辑。

2. 测试设计

根据信用卡还款代码，设计出程序流程图（图 3-2），并对程序流程图做节点标记，分析流程图的判定条件与结果。

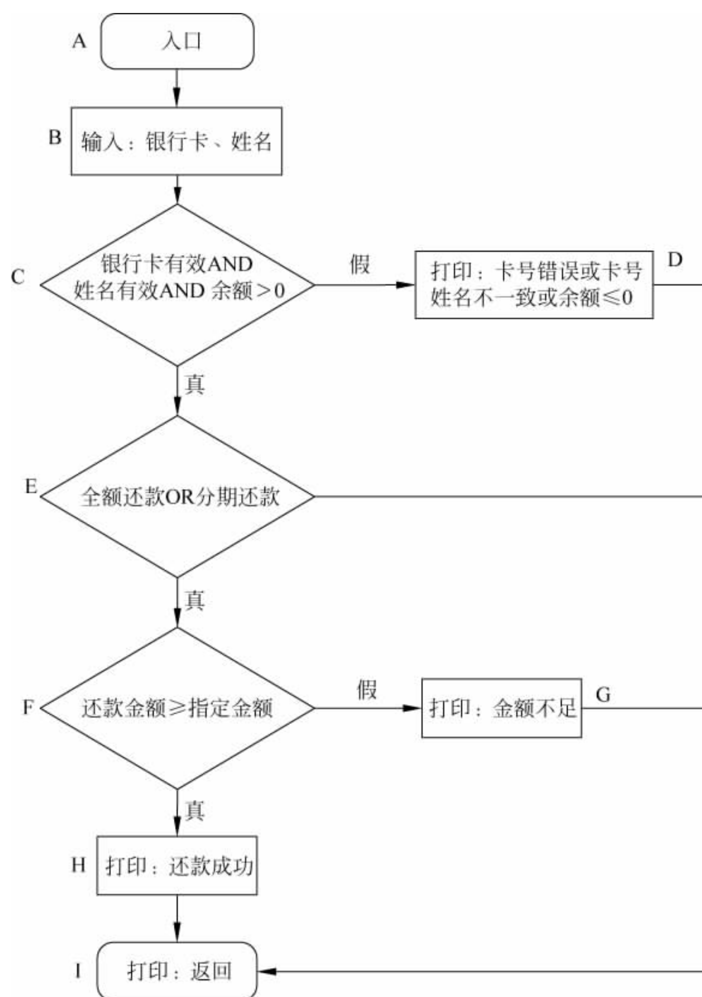

图 3-2　程序流程图

3. 测试执行

根据业务场景与流程逻辑判定，运用修正条件/判定覆盖法进行用例设计。修正条件/

判定覆盖法是为了实现条件/判定覆盖中尚未考虑到的各种条件组合情况覆盖，减少条件组合覆盖中产生的过多、无价值的测试用例。具体地说，修正条件/判定覆盖满足以下条件：

(1) 每个判定的所有可能结果至少能取值一次(达到判定覆盖)。

(2) 判定中的每个条件的所有可能结果至少取值一次(达到条件覆盖)。

(3) 一个判定中的每个条件独立地对判定的结果产生影响(在条件组合中固定一个变量或条件，改变另一个变量或条件，如果对结果有影响，就需要测试，如果对结果没有影响就不需要测试)。

(4) 每个入口和出口至少执行一次，覆盖不同入口或出口的路径。

根据修正条件/判定覆盖方法(MC/DC)进行分析，得到如表3-1所示的符合MC/DC质量标准的测试用例。

表3-1　符合MC/DC质量标准的测试用例

测试用例名称	测试用例描述	测试路径
CASE 1	还款成功：全额还款	ABCEFHI
CASE 2	还款成功：分期还款	ABCDFHI
CASE 3	还款失败：不选择全额还款、分期还款	ABCEI
CASE 4	还款失败：银行卡有效、姓名无效、余额>0	ABCDI
CASE 5	还款失败：银行卡无效、姓名有效、余额>0	ABCDI
CASE 6	还款失败：银行卡有效、姓名有效、余额≤0	ABCDI
CASE 7	还款失败：全部还款	ABCEFGI
CASE 8	还款失败：分期还款	ABCEFGI

4. 测试结果分析

从实验3可以看出，修正条件/判定覆盖是逻辑覆盖方法中相对较强的，超过判定覆盖、条件覆盖和条件/判定覆盖。

3.7 实例练习

根据以下程序(根据销售额计算奖金)设计修正条件/判定覆盖的测试用例：

```
#include <stdio.h>
int main(void)
{
float sales,prize;

printf("请输入月销售额\n:");
scanf("%f", &sales);
if (sales <= 10000)
  {
    prize = sales * 0.2;
    printf ("干得不错!\n");
    printf("奖金是 %f\n", prize);
  }
else if ((sales > 10000) && (sales <= 20000))
  {
```

```
    prize = 2000 + (sales - 10000) * 0.15;
    printf ("干得不错!\n");
    printf("奖金是 %f\n", prize);
  }
else if ((sales > 20000) && (sales <= 50000))
  {
    prize = 3500 + (sales - 20000) * 0.08;
    printf ("干得不错!\n");
    printf("奖金是 %f\n", prize);
  }
else if ((sales > 50000) && (sales <= 100000))
  {
    prize = 5500 + (sales - 20000) * 0.08;
    printf ("干得不错!\n");
    printf("奖金是 %f\n", prize);
  }
else if (sales > 100000)
  {
    prize = 7900 + (sales - 20000) * 0.05;
    printf ("非常优秀!\n");
    printf("奖金是 %f\n", prize);
  }
else
  {
    printf("需要努力!\n")
  }
return 0;
}
```

实验 4　基于 JUnit 的单元测试

（共 2 学时）

4.1　实 验 目 的

（1）通过动手实际操作，巩固所学的单元测试相关知识；

（2）初步了解 JUnit 工具的使用方法，加深对单元测试的认识。

4.2　实 验 前 提

（1）学习单元测试基本知识；

（2）熟悉 Eclipse 工具的基本操作；

（3）掌握基于 Eclipse 工具的 Java 编程；

（4）选择一个被测试的 Web 应用系统，能够正常编译部署（本实验中选择开源 Web 框架 Jeesite）作为单元测试对象。

4.3　实 验 内 容

针对被测试的 Web 应用系统（本实验中为开源 Web 框架 Jeesite）中的某个类进行单元测试，并使用 JaCoCo 对测试覆盖率进行分析。

4.4　实 验 环 境

（1）2～3 个学生一组；

（2）基础硬件清单：1 台 Windows 操作系统的客户端（进行单元测试）；

（3）Jeesite 框架：Jeesite 网站源码需要转换成 Eclipse 工程，若需要部署网站还要安装 MySQL 数据库，可自行拓展，具体可按照源码 doc 目录中提供的帮助文档进行操作。本次实验直接从网盘的 jeesite-master 文件目录中下载，在本地安装，重命名为 jeesite；

（4）Java 环境：在客户端上需要安装 Java 运行环境和 Eclipse。Eclipse 安装路径需要记录，如本实验中使用的路径是 C:\eclipse-jee-juno-win32\eclipse。具体安装步骤参考附录 A。