Ch2 代码单元测试

一、判断是非并改正错误

1. 单元测试的最终目的是满足期望的覆盖准则。（ ）
2. 数据驱动测试脚本将测试数据与测试逻辑分离。（ ）
3. 参数化测试是一种数据驱动的测试技术。（ ）
4. 单元测试用例的执行速度快。（ ）
5. 需要使用外部资源且执行时间过长的测试可以认定为不是一个单元测试。（ ）
6. 符合判定-条件测试思想的测试用例集合一定可以发现布尔操作符使用错误的缺陷。（ ）
7. 若覆盖准则 C1 包含（subsume）覆盖准则 C2，则说明满足 C2 的测试集合也满足 C1。（ ）
8. 变异体(Mutant)被杀死(kill)是指至少存在一个测试用例，使变异体产生与原代码不相同的运行结果。（）
9. 分支/条件覆盖的揭错能力高于分支覆盖。（）
10. 单元测试是验证单元的功能是否满足用户需求测试过程。（）

二、单项选择（每题有且仅有一个答案）

1. 代码如**错误!未找到引用源。**所示，对于测试集合{（“8613012345678”,true）}，其判定覆盖度为（ ）

```
public boolean q2(String phone){
    if (phone == null)
        return false;
    if ((phone.length() == 13) && (phone.startsWith("86")))
        return true;
    return false;
}
```

图 1

A．0%； B. 50%； C. 80%； D. 100%；

2. 若图 2 是**错误!未找到引用源。**的一个一阶变异，则测试输入集合{N=1，N=2}的变异得分为（ ）

```java
public void sc4(int N) {
    if (N >= 2 || N <= 20) {
        for (int i = 1; i <= 10; i++) {
            System.out.println(N + " x " + i + " = " + N * i);
        }
    }
}
```

变异体，将&& 改为 ||

图 2

A．0； B. 0.25； C. 0.5； D. 1；

3. 两个存在交互关系的单元 caller 和 callee 的数据流图如图 3 所示，其中，caller 的节点 7 为调用点 callsite，若已知（m, a），(n, b)为耦合变量，下列说法正确的是（ ）

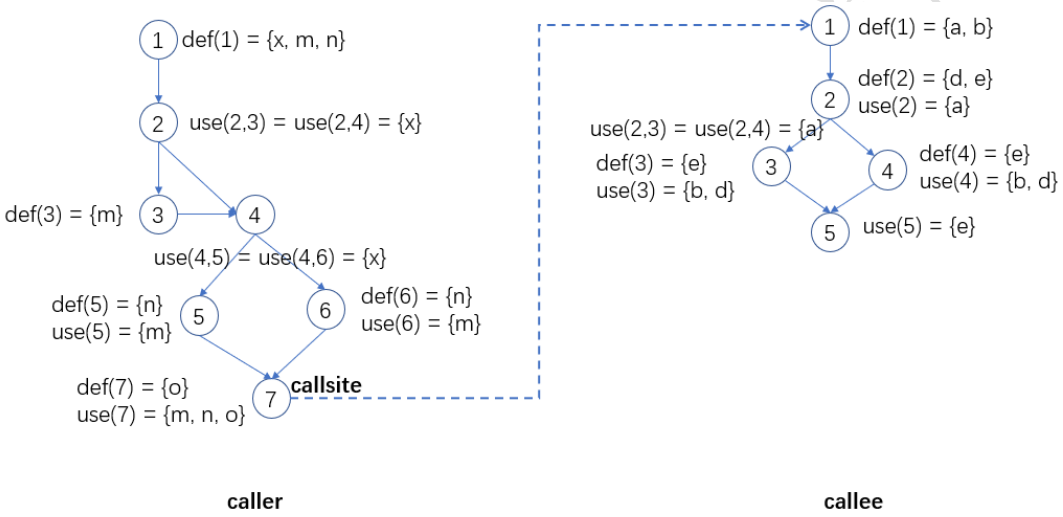


图 3

A. 耦合变量（m,a）中 m 的最后定义为 caller 的节点 3；

B. 耦合变量（m,a）中 a 的首次使用为 callee 的节点 2；

C. 耦合变量（n,b）中 n 的最后定义为 caller 的节点 5；

D. 耦合变量（n,b）中 b 的首次使用为 callee 的节点 3；

4. 判定覆盖不可以揭示（ ）

A．表达式取反缺陷； B. 简单条件插入缺陷； C.与引用缺陷； D. 或引用缺陷；

5. 某待测单元代码如图 4 所示，对于变量 stock 而言，其定义节点有（ ）个

A．0； B. 1； C. 2； D. 3；

```
public void buy(Stock stock){

    List<Stock> stocks = stockMap.get(stock.getSymbol());

    if(stocks != null){
        stocks.add(stock);
    }else {
        stocks = new ArrayList<Stock>();
        stocks.add(stock);
        stockMap.put(stock.getSymbol(),stocks);
    }
}
```

图 4

6. 代码如**错误!未找到引用源。**5 所示，对于测试输入集合{（8,“08:10:20PM”），（8,“08:10:20AM”）}而言，其判定覆盖度为（ ）

```
3     public void singleChoice1(int hour, String s){
4
5         if (hour < 12 && s.charAt(8) == 'P')
6             hour += 12;
7         if (hour == 12 && s.charAt(8) == 'A')
8             hour = 0;
9         System.out.println("The hour of 24 hour format is : " + hour);
10    }
```

图 5

A．0%； B. 50%； C. 80%； D. 100%；

7. 下列覆盖准则中，覆盖度最高的是（ ）

A．全定义覆盖； B. 全使用覆盖； C. 全定义-使用覆盖； D. 基路径覆盖；

8. 若图 6 是图 5 的一阶变异，则下列测试输入中（ ）可以杀死该变异体

```
12    public void singleChoice1Mutant(int hour, String s){
13
14        if (hour < 12 && s.charAt(8) == 'P')
15            hour += 12;
16        //Mutated if (hour == 12 && s.charAt(8) == 'A')
17        if (hour == 12 && s.charAt(8) != 'A')
18            hour = 0;
19        System.out.println("The hour of 24 hour format is : " + hour);
20    }
```

图 6

A．(8,“08:10:20PM”)； B. (11，“11:10:20AM”)；

C. (12,“12:10:20PM”)； D.（13，“13:10:20AM”)；

9. 在其它情况相同的条件下，揭错能力最高的覆盖准则是（ ）

A．语句覆盖； B．判断覆盖； C．判定/条件覆盖； D．多条件覆盖；

10. Junit 中的参数化测试采用的是（ ）技术

A．数据驱动脚本；B．线性脚本； C．关键字驱动脚本；D．结构化脚本；

11. 代码如图 7 所示，若满足修正的判定条件覆盖（Modified Decision-Condition Coverage，MC/DC），则测试用例个数最少为（ ）

```
if (path.getName().toLowerCase().endsWith(".pdf")
 || path.getName().toLowerCase().endsWith(".doc")
 || path.getName().toLowerCase().endsWith(".html"))
```

图 7

A．1； B. 2； C. 4； D. 8；

12. 代码如图 8 所示，其中变量 result 的定义节点有（ ）个

A．1； B. 2； C. 3； D. 4；

```
public int sum(int [] x){
    int result = 0;
    for (int i = 0; i < x.length; i++)
        result = result + x[i];
    return result;
}
```

图 8

13. 将测试数据与测试脚本分开的脚本技术是（ ）

A．数据驱动脚本；B. 线性脚本；C. 关键字驱动脚本；D. 结构化脚本；

14. 在基于数据流的覆盖准则中揭错能力最高的覆盖准则是（ ）

A．所有定义； B. 所有使用； C. 所有定义-引用； D. 以上答案都不正确；

15. 代码如图 9 所示，正确的是（ ）

A、balance 的使用节点是 3， 8； B、balance 的定义节点是 4；

C、interest 的使用节点 3； D、interest 的定义节点 4；

```
1 public void acitonPerformed(ActionEvent event)
2 {
3   double interest = balance * rate/100;
4   balance +=interest;
5
6   NumberFormat formatter
7     = NumberFormat.getCurrencyInstance();
8   System.out.println("balance="+formatter.format(balance));
9 }
```

图 9

16. 模拟调用被测单元的测试脚本被称为（）

A、驱动； B、桩； C、测试模块； D、测试框架；

三、多项选择（每题有 2 个或 2 个以上答案）

1. 判定 B = （c_1 ||c_2 ） && c_3，其中，c_1, c_2, c_3分别表示条件,若期望满足修正的判定/条件覆盖，则测试输入应包括（ ）

   A. [c1=true, c2=false, c3=true]； B. [c1=false, c2=false, c3=true]

   C . [c1=false, c2=true, c3=true]; D. [c1=true, c2=false, c3=false]}

2. 下列关于基于变异的覆盖准则说法准确的是（ ）

   A.满足弱变异覆盖的测试输入应具备可达性和可传播性;

   B.若测试用例满足强变异覆盖，则其一定也满足弱变异覆盖;

   C.若测试用例使得系统产生失效，则一定说明其满足强变异覆盖;

   D.若系统实际执行结果和预期结果一致，则说明测试用例满足弱变异覆盖;

3. 下列关于变异测试说法正确的是（ ）

   A . 变异测试可用于评估测试集合的有效性;

   B. 变异得分高的测试集合，揭错能力一定强;

   C. 弱变异要求测试输入满足可达性和可感染性;

   D. 满足强变异覆盖的测试用例的揭错能力比满足弱变异覆盖的测试集合揭错能力强;

4. 判定B = （y % 400 == 0）|| ((y % 4 != 0) && (y % 100 == 0))，其满足判定/条件覆盖的测试输入集合为（ ）

   A. y = 1900; B. y = 2000; C.y=2002; D. y = 2004;

5. 判定B = c_1&&（c_2 || c_3)，其中，c_1, c_2, c_3分别表示条件,若期望c_1独立影响 B 的结果，则测试输入值可以为（ ）

   A. {[c1 = true, c2=true, c3=true], [c1 = false, c2=true, c3=true]}

   B. {[c1 = true, c2=true, c3=false], [c1 = false, c2=true, c3=false]}

   C. {[c1 = true, c2=false, c3=true], [c1 = false, c2=false, c3=true]}

   D. {[c1 = true, c2=false, c3=false], [c1 = false, c2=false, c3=false]}

6. 代码如右图所示，（ ）是变量 response 的使用节点

A. cin>>response;

B. response = toupper(response);

C. if(response == 'L');

D. else if(response != 'Y');

```
while(!done)
{
    char response;
    int guess;

    bool found = false;
    int lo=1,hi=100;
    while(!found && (lo <= hi))
    {
        guess = ( lo + hi )/2;
        count << "Is it" << guess <<"(L/H/Y):"<<endl;
        cin >> response;
        response = toupper(response);
        if(response == 'L')
            lo = guess + 1;
        else if (response == 'H')
            hi = guess - 1;
        else if (response !='Y')
            cout << "What?Try again..."<<endl;
        else
            found = true;
    }
    .....
}
```

7. 为了满足修正的分支条件覆盖，对于（!found && (lo<=hi)）应选择（ ）测试用例

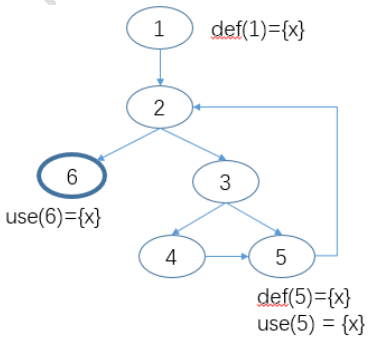
A. found = T, (lo<=hi)=T;　　B. found = T, (lo<=hi)=F;

C. found = F, (lo<=hi)=T;　　D. found = F, (lo<=hi)=F;

8. 有数据流图如右图所示，对于变量 x 而言，路径集合（ ）满足 x 的全使用覆盖

A. [1, 2, 6]　B. [1, 2, 3, 4, 5]

C. [5, 2, 6]　D. [5, 2, 3, 4, 5]



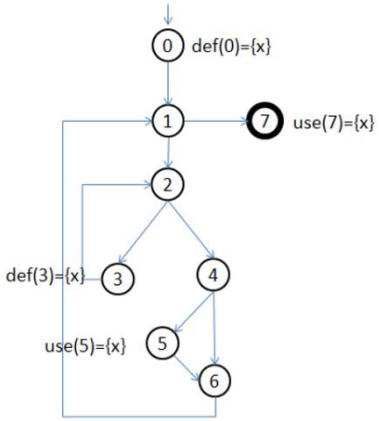
9. 某数据流图如**错误!未找到引用源。**所示，其中标注了有关变量 x 的定义和使用情况，若满足变量 x 的全使用覆盖，则需要包含的路径有（）

A. [0, 1, 7]

B. [0, 1, 2, 4, 5]

C. [3, 2, 4, 5]

D. [3, 2, 4, 6, 1, 7]



10. 有 Junit 测试代码如**错误!未找到引用源。**所示，其中（AD）标注使用不正确

```
public class TestLifeCycle {

    @BeforeAll
    public void initAll() { System.out.println("Before all tests"); }

    @BeforeEach
    void init() { System.out.println("Before each test"); }

    @Test
    void testDemoMethod1() { System.out.println("The 1st test"); }

    @Test
    void testDemoMethod2() { System.out.println("The 2nd test"); }

    @AfterEach
    void tearDown() { System.out.println("After each test"); }

    @AfterAll
    void tearAll() { System.out.println("After all tests"); }
}
```

A .@BeforeAll; B. @BeforeEach;

C . @AfterEach; D. @AfterAll;

四、问答题

1、给定一个长度为 5 且不为 null 的正整数数组，其中每个正整数的取值范围为[1,1000000000]，miniMaxSum(int[] array)方法计算数组中任意 4 个元素之和的最小值和最大值，并以字符串的形式返回以空格分割的最小值和最大值。例如，输入：arr = [3,2,1,5,4]，返回：10 14。因在数组[3,2,1,5,4]任意 4 个元素有(3,1,5,4)，(3,2,5,4)，(3,2,1,4)，(3,2,1,5)和(2,1,5,4)，其中，和最小的是(3,2,1,4)相加的结果为：3+2+1+4=10，和最大的是(3,2,5,4)相加的结果为：3+2+5+4=14。代码如图 10 所示。

1) 代码中存在 1 处缺陷，请指出该处缺陷。

2) 构造 miniMaxSum 的控制流图（Control Flow Graph，CFG）

3) 根据 CFG 计算相应的基路径（Prime Path）集合。

4) 是否存在测试输入集合满足基路径覆盖并简要说明理由。

```java
public String miniMaxSum(int[] array) {

    int sum = 0;
    int min = 1000000000;
    int max = 0;
    String ret = "err_unknown";

    for (int i = 0; i < array.length; i++) {
        int n = array[i];
        sum = sum + n;
        if (min > n) {
            min = n;
        }
        if (max < n) {
            max = n;
        }
    }
    ret = (sum - max) + " " + (sum - min);
    return ret;
}
```

图 10

2、类 Quadratic 计算一元二次方程 $ax^2+bx+c=0$ 的根，代码如图 11 所示，图 12 是方法 solve 的一个变异体，请回答下列问题

1) 构造 solve 的控制流图（Control Flow Graph，CFG）(5 分)；

2) 根据 solve 的 CFG 计算相应的基路径（Prime Path）集合（5 分）。

3) 如果存在，设计测试用例集合 ts1 使其满足基路径覆盖（5 分）。

4) 构造 solve 和 calroots 的数据流图（10 分）

5) 设计满足全耦合定义使用覆盖的测试集合 ts2（10 分）

6) 计算 ts2 的变异得分.（5 分）

```java
public class Quadratic {

    private float root1, root2;

    public void solve(String[] inputs) {
        int a, b, c;
        boolean hasRoots;
        if (("-c").equals(inputs[0])) {
            a = Integer.parseInt(inputs[1]);
            b = Integer.parseInt(inputs[2]);
            c = Integer.parseInt(inputs[3]);
        } else {
            a = 10;
            b = 9;
            c = 12;
        }
        hasRoots = calRoots(a, b, c);
        if (hasRoots)
            System.out.println("Quadratic:" +
                    " root1 = " + root1 + " , root2 = " + root2);
        else
            System.out.println("Quadratic: no solution");
    }

    private boolean calRoots(int coA, int coB, int coC) {
        boolean result = false;
        int expr = coB * coB - 4 * coA * coC;
        if (expr >= 0) {
            root1 = (float) ((-coB + Math.sqrt(expr)) / (2.0 * coA));
            root2 = (float) ((-coB - Math.sqrt(expr)) / (2.0 * coA));
            result = true;
        }
        return result;
    }
}
```

图 11

```java
public void solve(String[] inputs) {
    int a, b, c;
    boolean hasRoots;
    if (("-c").equals(inputs[0])) {
        a = Integer.parseInt(inputs[1]);
        b = Integer.parseInt(inputs[2]);
        c = Integer.parseInt(inputs[3]);
    } else {
        a = 10;
        b = 9;
        c = 12;
    }
    hasRoots = calRoots(a, c, b);//mutant
    if (hasRoots)
        System.out.println("Quadratic:" +
                " root1 = " + root1 + " , root2 = " + root2);
    else
        System.out.println("Quadratic: no solution");
}
```

图 12

3、numOfMaximum 方法输出给定正整数数组中最大值的个数。例如，输入[3,2,4,4]，numOfMaximum 输出 2，其代码如图 13 **错误!未找到引用源。**所示。

1) 构造 numOfMaximum 的控制流图（Control Flow Graph，CFG）

2) 根据 CFG，计算相应的基路径（Prime Path）集合。

3) 构造 numOfMaximum 的数据流图（Data Flow Graph，DFG）

```java
public int numOfMaximum(int[] arr) {
    int maximum = 1;
    int countMax = 0;
    for(int i = 0; i < arr.length; i++) {
        int tmp = arr[i];
        if(tmp > maximum) {
            maximum = tmp;
            countMax = 1;
        }
       else if(tmp == maximum) {
            countMax++;
        }
    }
    return countMax;
}
```

图 13

4、对于第 3 题的 numOfMaximum，测试用例（arr = [1,2,1]，expect result = 1）是否满

足判定覆盖？针对下表给出的 3 个变异体，其变异得分是多少？（10 分）

Mutant ID	Mutated Position	Mutated code (original code→mutated code)
M_1	Line 5	countMax = 0 → countMax = 1
M_2	Line 8	tmp > maximum → tmp >= maximum
M_3	Line 8	tmp > maximum → tmp < maximum

5、下图是 findVal() 和其一个变异体代码如图所示，请回答:

1) 如果可能，给出一个测试输入强杀死 m

2) 如果可能，给出一个测试输入弱杀死 m

```
1  //Effects: If numbers null throw NullPointerException
2  // else return LAST occurrence of val in numbers[]
3  // If val not in numbers[] return -1
4   public static int findVal (int numbers[], int val)
5  {
6    int findVal = -1;
7    for (int i=0; i<numbers.length; i++)
8    if (numbers [i] == val)
9      findVal = i;
10   return (findVal);
11 }
```

```
1  //Effects: If numbers null throw NullPointerException
2  // else return LAST occurrence of val in numbers[]
3  // If val not in numbers[] return -1
4   public static int findVal (int numbers[], int val)
5  {
6    int findVal = -1;
7    for (int i=(0+1); i<numbers.length; i++) //(mutant)
8    if (numbers [i] == val)
9      findVal = i;
10   return (findVal);
11 }
```

图 14

6、给出一个 32 位的有符号整数，reverse(int x)将这个整数中每位上的数字进行反转，例如输入 123，输出 321； 输入 120，输出 21；输入 -123，输出-321；reverse(int x)的代码如图所示，请回答:

1) 构造 reverse 方法的控制流图（Control Flow Graph，CFG）

2) 根据 CFG，计算相应的基路径集合（Prime Path）

```
53  public int reverse(int x) {
54      int y = 0;
55      while (x != 0) {
56          if (y > 214748364 || y < -214748364) {
57              return 0;
58          }
59          y = y * 10 + x % 10;
60          x = x / 10;
61      }
62      return y;
63  }
```

图 15

7、图中的 if 语句是某点火系统采用二级发射方案的控制条件，其中 done 是布尔变量表示点火准备就绪，(x,y,z)代表火箭的笛卡尔系坐标为整型变量，current={East，West，South，North}是枚举变量代表方向，请计算其满足修正的判断条件覆盖（MC/DC）的测试集合并给出相应的过程（10 分）

```
if (done && (((x < y) && (z * z <= y)) || (current == Direction.South)))
```

8、代码如**错误!未找到引用源。**所示，请回答:

1） 构造 Sort 方法的控制流图（Control Flow Graph，CFG）；

2） 根据 CFG，计算相应的基路径（Prime Path）集合

```
1 void Sort(int iRecordNum,int Type)
2 {
3   int x = 0;
4   int y = 0;
5
6   while (iRecordNum--)
7   {
8     if(0 == iType)
9       x = y+2;
10    else
11      if ( 1 == iType )
12       x = y+10;
13      else
14       x = y+20;
15    }
16 }
```

9、根据提示补全针对 reverse(int number)(图 15)的 Junit 测试脚本，32 位整型数的取值范围为[-2147483648，2147483647]

```
//① 定义一个参数化测试，测试的显示名称需要包括测试序号，相应的测试输入和预期结果
//② 采用逗号分割的测试数据源，提供的实参包括三组: (123,321), (210,21) 和(-123,-321)
void should_reverse_number(int number, int expectedResult) {
    FinalExam finalExam = new FinalExam();
    int actual = finalExam.reverse(number);
    assertEquals(expectedResult, actual);
}
 //③ 可以正常执行反转的number的下边界值
@Test
void should_reverse_at_minimum_number() {
    FinalExam finalExam = new FinalExam();
    int actual = finalExam.reverse(      ③      );
    assertEquals(          ③          , actual);
}
//④ 可以正常执行反转的number的上边界值
@Test
void should_reverse_at_maximum_number() {
    FinalExam finalExam = new FinalExam();
    int actual = finalExam.reverse(      ④      );
    assertEquals(          ④          , actual);
}
```