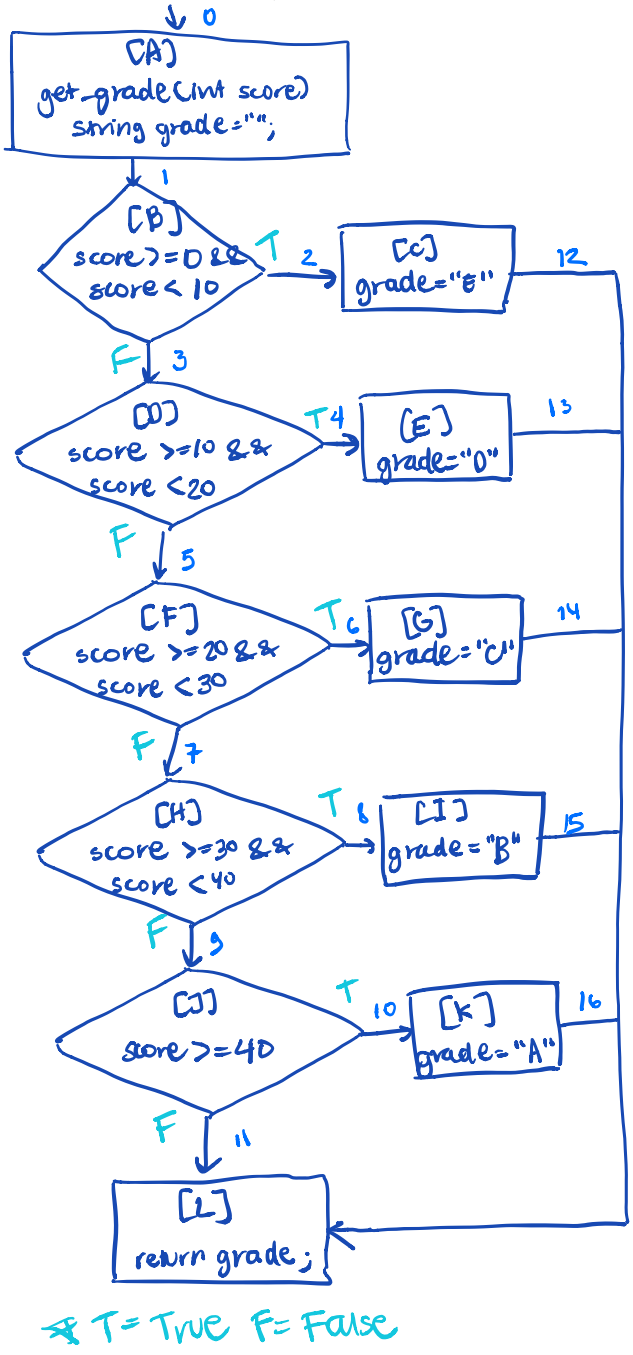
ST 2016 RESIT Past Paper

1. Consider the following method:

```
 public String get_grade(int score)
{
   string grade = "";
   if (score >=0 && score <10) {
      grade = "E";
   }
   else if (score >=10 && score <20) {
      grade = "D";
   }
   else if (score >=20 && score <30) {
      grade = "C";
   }
   else if (score >=30 && score <40) {
      grade = "B";
   }
   else if (score >=40) {
      grade = "A";
   }
   return grade;
}
```

a. Draw CFG, label edges with numbers and blocks with letters.



* T = True F = False

b. Write tests, input, expected output, and sequence of edges.

Test	Input	Expected Output	Seq of edges
a	0	'E'	0, 1, 2, 12
b	10	'D'	0, 1, 3, 4, 13
c	20	'C'	0, 1, 3, 5, 6, 14
d	30	'B'	0, 1, 3, 5, 7, 8, 15
e	40	'A'	0, 1, 3, 5, 7, 9 10, 16
f	-10	''	0, 1, 3, 5, 7, 9, 11

c. Coverage from tests?

coverage = 16/16 = 100%.

No additional tests needed.

d. Typo → || instead of && in if (score >=0 || score <10). Will the tests reveal this fault?

Yes. Test (f) will have an output of 'E' instead of '' as expected.

e. Coverage criteria for complex boolean expressions? Develop tests for ((x1 && y1) || z1). → truth table

MC/DC

x1	y1	z1	((x1 && y1) || z1)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

total of 5 tests needed.

f. find-maximum method: write tests, input, expected output that achieves "All DU pairs" coverage for var c, max, and index. Write out <D, U> pairs for the vars and show coverage achieved.

```
1  int find_maximum(int a[], int n) {
2  int c, max, index;
3
4  max = a[0];
5  index = 0;
6
7  for (c = 1;
8            c < n;
9            c++) {
10    if (a[c] > max) {
11       index = c;
12       max = a[c];
13    }
14 }
15
16  return index;
17 }
```

variables	definitions	uses	<D,U> pairs
a	1	4, 10, 12	<1,4>, <1,10>, <1,12>
n	1	8	<1,8>
c	2, 7, 9	8, 9, 10, 11, 12	<7,8>, <7,10>, <7, 11>, <7, 12>, <7,9>, <9,9>, <9,10>, <9,11>, <9,12>
max	2, 4, 12	10	<4,10>, <12,10>
index	2, 5, 11	16	<5,16>, <11,16>

Test a = [1, 2, 5] and n = 3 expected = 5 excludes these.
Test a = [1] and n = 1 expected = 1 excludes these.

2. Functional testing of a dialog w/ three sets of radio buttons, rep. travel needs within the UK.
First set of radio buttons: destination by country (England, Scotland, Wales, NI)
second set: aisle/window seat
third set: coach choice (business/economy)
Click OK → app takes diff path for each combo.

a. No. of all the different combinations?

$4 \times 2 \times 2 = 16$

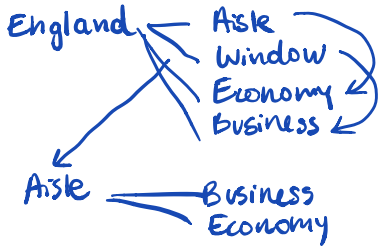
b. How to bring down the no. of tests?

Pairwise Testing - intuitive constraints reduce the no. of combinations to a small amount of test cases.
- generate combinations that efficiently all pairs

Why would this work?
→ Most failures are triggered by single values / combinations of a few values. Covering pairs reduces the no. of test cases, but reveals most faults.

c. Apply this method.

- England, Aisle, Business
- England, Window, Economy
- scotland, Window, Business
- Scotland, Aisle, Economy
- Wales, Window, Economy
- Wales, Aisle, Business
- Northern Ireland, Window, Business
- Northern Ireland, Aisle, Economy



d. Combinations of inputs your test suite will miss. AND WHY?

- England, Aisle, Economy
- England, Window, Business
- Scotland, Window, Economy
- Scotland, Aisle, Business
- Wales, Window, Business
- Wales, Aisle, Economy
- Northern Ireland, Window, Economy
- Northern Ireland, Aisle, Business

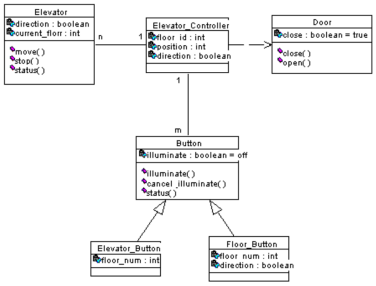
?

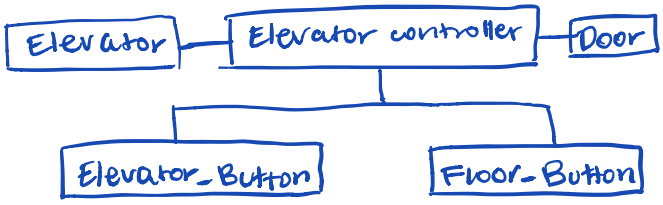
These combinations were missed because pairwise testing covers all two-way, not three-way combinations, and our input is in triples and not pairs.

e. spec: a function that takes as input an integer inp1 and returns half the value of inp1 if inp1 is even, and inp1 otherwise.
Identify representative value classes for inp1 with a brief explanation.

Characteristic	Partition	Value classes	Explanation
inp1 is even or odd	P1	inp1 is even	inp1 % 2 == 0 IS TRUE
	P2	inp2 is odd	inp1 % 2 == 0 IS FALSE

3. a. Draw the use/include relation for this class diagram to represent potential interactions between classes to be verified through the inter-class testing.





b. Grandmother approve if 25 < age < 40 and (rich || smart).

```
ShallowGrandmother.java
1 import java.util.Scanner;
2
3 public class ShallowGrandmother {
4     public static void main( String[] args ) {
5         Scanner keyboard = new Scanner(System.in);
6         int age;
7         double income, smart;
8         boolean allowed;
9
10        System.out.print( "Enter your age: " );
11        age = keyboard.nextInt();
12
13        System.out.print( "Enter your yearly income: " );
14        income = keyboard.nextDouble();
15
16        System.out.print("How smart are you, on a scale from 0.0 to 10.0?");
17        smart = keyboard.nextDouble();
18
19        allowed = (age>25 && age<40 && (income > 50000 || smart >= 8.5));
20
21        System.out.println("Allowed to date my grandchild? " + allowed);
22    }
23 }
```

Inputs are:
'age', 'income', 'How smart are you?'

Output:
'allowed' (boolean).

i. Derive 6 mutations for expression/operands.

Mutation	Line number	Change
M1	11	age = 22
M2	17	smart = 0
M3	19	age >= 25
M4	19	age > 25 || age < 40 || ...
M5	19	(.... && !(income <= 50000 || smart < 8.5))
M6	21	Syso... ("Allowed ..." + ! allowed)

ii. Tests that reveals the mutations

	INPUTS				
Mutation	Age	Income	Smart	Expected output	Actual Output
M1	27	60000	9.5	ALLOWED	NOT ALLOWED
M2	27	40000	9.5	ALLOWED	NOT ALLOWED
M3	25	60000	8.5	NOT ALLOWED	ALLOWED
M4	8	70000	9.0	NOT ALLOWED	ALLOWED
M5	27	60000	8.5	ALLOWED	ALLOWED → equivalent
M6	26	80000	8.5	ALLOWED	NOT ALLOWED

iii. Do the tests achieve condition coverage. If yes explain, if no, add tests.

There is only one condition:
(age > 25 && age < 40 && (income > 50 000 || smart >= 8.5))

Our tests evaluate this condition as both true and false.
Thus, full condition coverage is achieved.

c. You have 1000 regression tests and you can only run 200.
How will you choose which tests to run + discuss.

- Regression test selection
  → from entire test suite, only select subset of test cases whose execution is important / relevant to changes.

- Regression test set minimisation
  → identify redundant test cases and remove them from the test suite to reduce its size.

- Regression test set prioritisation
  → sort test cases in order of increasing cost per additional coverage