Vrije Universiteit, Department of Computer Science

Examination paper for **Software Testing-SOLUTIONS**

27 March 2014 15:15-18:30

This is a closed book written exam.

No printed material or electronic devices are admitted for use during the exam.

The answers have to be given in English or Dutch.

Both homework and exam are compulsory and graded on an 1 to 10 scale.

The exam grade is calculated as (Q1+Q2+Q3+Q4+Q5 +10)/10.

The final grade is calculated as 0.5*homework + 0.5*exam

A pass is given if both homework and exam components are >= 5.5.

	Q1	**Q2**	**Q3**	**Q4**	**Q5 (code)**	**Σ Qi**	**Maximum credits= (ΣQi+10)/10**
a)	**5**			**5**	**3**		
b)	**5**			**7**	**5**		
c)	**5**				**3**		
d)	**5**				**8**		
e)	**5**				**5**		
f)	**5**						
Total	**30**	**12**	**12**	**12**	**24**	**90**	**10**

Good luck!

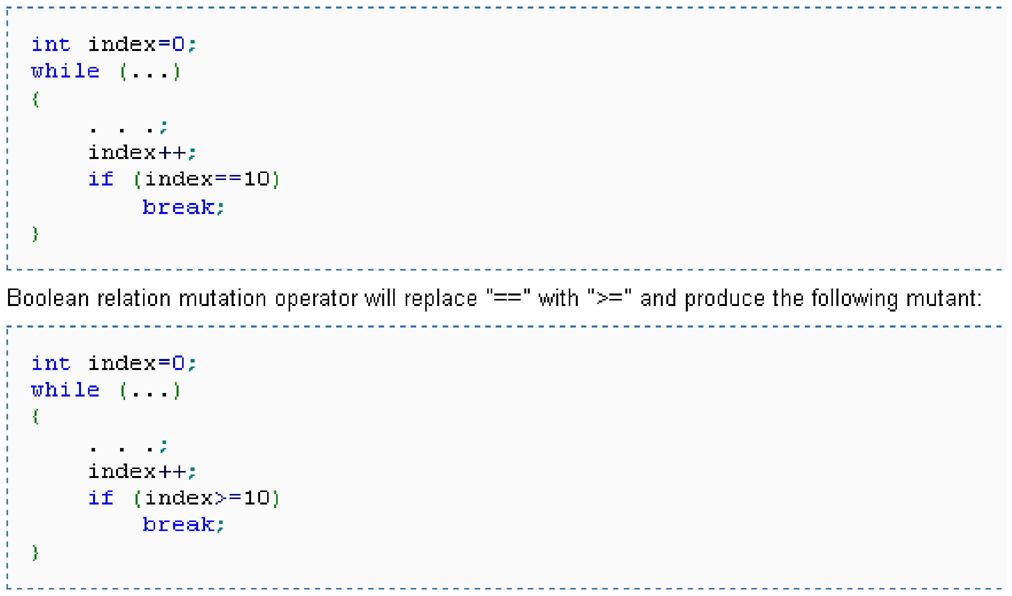
Q1. Concepts [30p]

a. Give an example of a test adequacy criterion and explain its role in software testing. [5p]
b. What is an equivalent mutant? Give an example. [5p]
c. Explain the essence of combinatorial testing: why it can be applied and how ? [5p]
d. How does risk-based testing work? [5p]
e. Compare Agile/scrum testing with traditional waterfall testing. [5p]
f. Safety critical software needs a special kind of testing, regulated by standards. Enumerate a few special requirements imposed on testing by these standards. [5p]

Solutions:

a) A test adequacy criterion can say something about how good a test suite is. For example the decision (or branch) coverage criterion. We can say: A test suite T is good for the program P when it covers 100% of all decision blocks in the control flow graph of P. Its role is twofold: it can guide test case generation by enhancing an existing test suite until the criterion is satisfied and it can help deciding when to stop testing.

b) An equivalent mutant is a slightly modified program that semantically is the same as the original program. As a consequence, an equivalent mutant can never be killed. An example can be:



```
int index=0;
while (...)
{
    . . .;
    index++;
    if (index==10)
        break;
}
```

Boolean relation mutation operator will replace "==" with ">=" and produce the following mutant:

```
int index=0;
while (...)
{
    . . .;
    index++;
    if (index>=10)
        break;
}
```

c) Combinatorial testing is a model-based test generation technique based on input modeling. It tries to reduce the explosive number of combinations of inputs by considering only t-way interactions of inputs where t = 2,3,..6. The justification is based on a research of NIST that applied data mining on all failures of medical devices, networks security vulnerabilities, etc during past ten years and concluded that failures happened only as a result of an interaction between t variables, where t is no more than 6. Very often pairwise testing (t=2) is enough. There are mathematical tools like orthogonal arrays and even automatic tools that given the input variable sand their discrete values can generate all possible t-way combinations that serve as input to test cases. There is no oracle and no expected output. The user has to generate this by hand.

d) Testing based on risk prioritize software modules or functions based on the risk. Risk is calculated as a product between the probability of occurrence and impact. Modules with high risk are tested more thoroughly.
e) Both waterfall and agile processes have quality as a goal. But in agile testing the users are involved earlier , testing is the responsibility of the team, all test levels are tested at once and communication is more important than documentation.
f) Here all kind of regulations stated by standards have to be mentioned. For example some techniques are especially required like: inspections, MC/DC coverage (required by DO 178b for aviation) or 100% requirements, statement or decision coverage, combinatorial testing n-way with n>3, formal verification of the design, peer reviews, fault tree analysis, probabilistic testing, all bugs have to be traceable, programmers that make 3 bugs are dismissed, simulations, compiler certification, verification of safety-behaviour during degraded and failure conditions, logging, avalanche testing, etc.

Testing from requirements Q2-Q4 [36p]

Q2. [12p]

Consider these requirements for a student examination grading module:

If the student scores 0 to less than 50 then assign D Grade, if the student scores between 50 to 69 then assign C Grade, if the student scores between 70 to 84 then assign B Grade, and if the student scores 85 to 100 then assign A Grade.

Generate test cases using equivalence partitioning combined with boundary value analysis for a by-contract testing of this module.

Solution:

The input variable is score.
The equivalence relation that will induce a partition of the input domain is in this case “A and B are in the same class if they generate the same output”. We

identify 4 classes. Because it is required to perform *by-contract* testing we will limit ourselves to only valid inputs.
VC1: 0<=score <=49 , expected output is D
VC2: 50<=score<=69, expected output is C
VC3:70<=score<=84, expected output is B
VC4: 85<=score<=100, expected output is A

BVA will require for each class to test for 5 elements: the boundaries, off by one inside the domain near the boundaries and one element in the middle.
We make the assumption that the precision of the grades is 1. For example for VC1 we will test the following inputs: 0,1, 20, 48 and 49.

Test design specification:

			Assumption: Grades are only between 0 and 100 boundaries included the lowest grade increment is 1
type	Description	Tag	BT (belongs to)
VC	**0<=score <=49**	**1**	
VB	Score = lower boundary	1.1	1
VB	Score = lower boundary+1	1.2	1
VB	Score = middle	1.3	1
VB	Score = high -1	1.4	1
VB	Score = high boundary	1.5	1
VC	**50<=score<=69**	**2**	
VB	Score = lower boundary	2.1	2

Etc

We obtain the following low- level test cases:

Tag	Test case ID	score	Expected output
1.1	TC1	0	D
1.2	TC2	1	D
1.3	TC3	20	D
1.4	TC4	48	D
1.5	TC5	49	D
2.1	TC6	50	C
2.2	TC7	51	C
2.3	TC8	60	C
2.4	TC9	68	C
2.5	TC10	69	C
3.1	TC11	70	B
3.2	TC12	71	B
3.3	TC13	78	B
3.4	TC14	83	B
3.5	TC15	84	B
4.1	TC16	85	A
4.2	TC17	86	A
4.3	TC18	90	A
4.4	TC19	99	A
4.5	TC20	100	A

Q3. [12p]

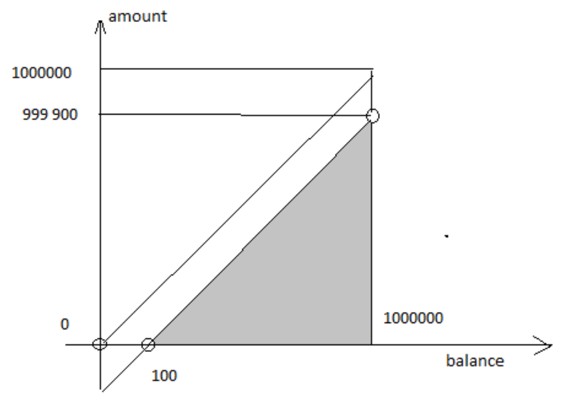
For a banking software, we want to test a small method, called `validate_withdraw`. This method has to decide whether a withdraw amount, required by the user, will be approved or rejected. A withdraw amount will be approved only if the user has an account that is currently open, it has enough money in it and the current balance left after withdrawal is still more than 100 euro. Both balance and amount have to be positive and cannot exceed 1000000 euros. If any of these conditions is not satisfied, the withdrawal will be rejected

Generate test cases for 1x1 domain testing of these requirements.

Solution:

We have the following inputs: account_state (open/not open), balance and amount. The output is rejected or approved.

We draw the domain for amount and balance and we eliminate some boundaries. We apply COO OOI rule to decide the OFF point that has to be tested.



The boundaries of the input domain and their ON and OFF points are :

balance > 100 border open ON point: balance =100, Off point: balance inside domain
balance<=1000000 border closed, ON point : balance = 1000000, OFF point: outside
0<amount border open, ON point: amount = 0, OFF point: amount inside the domain
balance – amount > 100 border open ON point: balance-amount = 100 OFF point balance – amount inside the domain > 100
account is open ON point = true , OFF point = false, typical in point = true

Variable	Condition		TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9	TC10	TC11	TC12
	balance > 100	ON	100	100										
		OFF			101	101								
	balance<=1000000	ON					10000000							
		OFF						10000001						
		Typical in							50000	1000	2000	40000	60000	3000
amount	0<amount	ON							0					
		OFF								1				
		Typical in	5		10									
	amount < balance -100	ON									1900			
		OFF										39 89 9		
		Typical in		-1		0.5	10000	300					300	50
account	Is open	ON											True	
		OFF												False
		Typical in	True	True	True	True	True	True	True	True	True	True		
	Expected output		R	R	R	A	A	A	R	A	R	A	A	R

Q4. [12p]

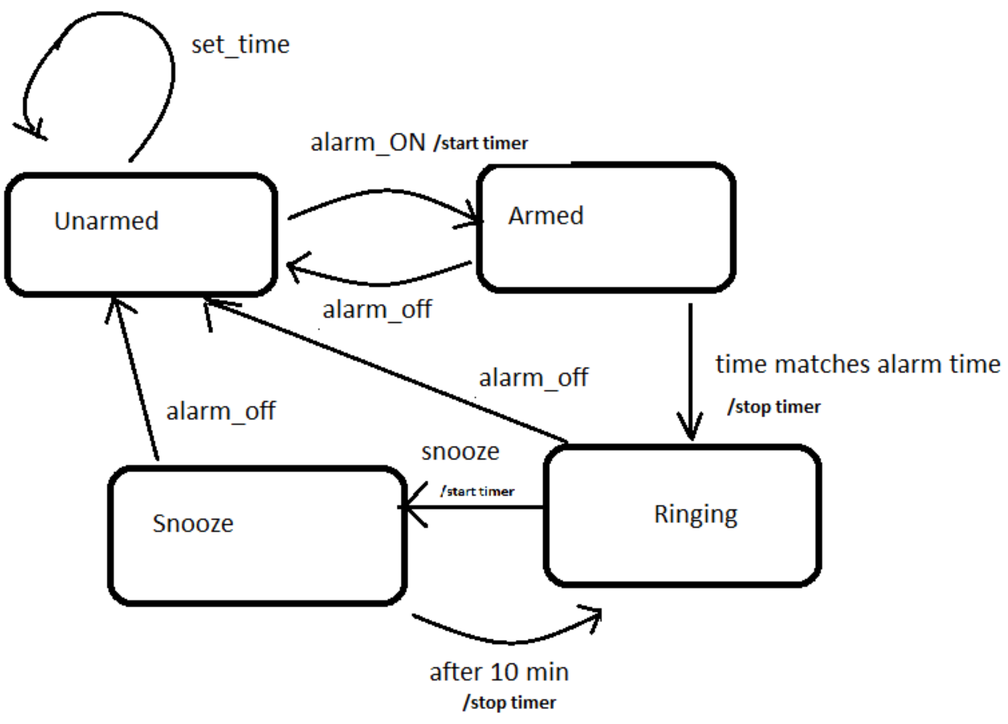
These are the requirements for an application that implements an alarm clock:

The user can set the time for alarm to go off.
The user can turn the alarm on or off.
The user can snooze the alarm.
When the clock arrives at the time set for the alarm to ring, then the alarm will ring.

a) Draw a state transition diagram to model this behavior [5p]
b) Apply model based test generation from this state transition diagram [7p].

Solution:

a) **This the state transityions diagram that models the behaviour**



b) From this model, we generate test cases that satisfy 100% transition coverage.

ID	Current State	Event	Action	Next State	
TC1	Unarmed	Set_time		Unarmed	
TC2	Unarmed	Alarm_ON	Start Timer	Armed	
TC3	Armed	Alarm_off		Unarmed	
TC4	Armed	Time matches	stop timer	Ringing	
TC5	Ringing	Alarm_off		Unarmed	
TC6	Ringing	Snooze	Start timer	Snooze	
TC7	Snooze	After 10min		Ringing	

TC8	Snooze	Alarm_off		Unarmed	

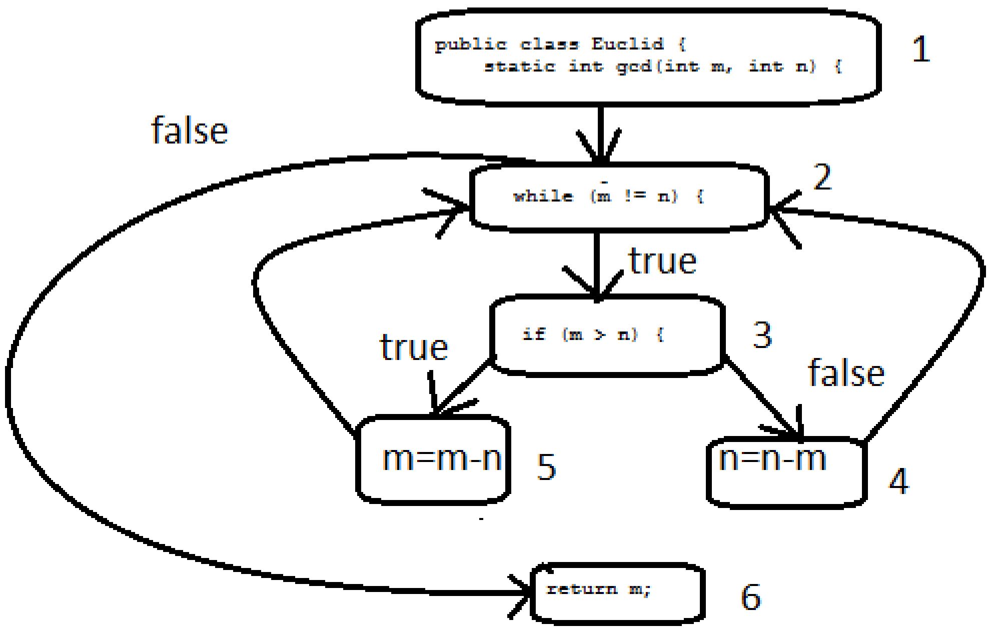
Q5. Code based testing [24p]

Below is a code snippet that computes the greatest common divisor of two natural numbers by Euclid’s algorithm

```
public class Euclid {
    static int gcd(int m, int n) {
        while (m != n) {
            if (m > n) {
                m -= n;
            } else {
                n -= m;
            }
        }

        return m;
    }
```

For this code snippet:
a) Draw the control flow graph. [3p]
b) Generate a test suite that achieves 100% statement coverage. [5p]
c) Enhance your test cases from b) to achieve 100% decision coverage. [3p]
d) Generate a test suite that is adequate with respect to the all-uses criterion [8p]
e) Generate a mutant and show a test case that will kill it. [5p]



a)

b) TC1: m=3, n=3, Expected output : 3
TC2: m=12 n = 9 expected output : 3
These 2 test cases will cover all statements.

c)The test suite from a) covers also all decisions
d)

variable	Defined in	c-use	p-use
m	1	{5,4,6}	{(2,3),(2,6),(3,5),(3,4)}
m	5	{5,4,6}	{(2,3),(2,6),(3,5),(3,4)}}
n	1	{5,4}	{(2,3),(2,6),(3,5),(3,4)}
n	4	{5,4}	{(2,3),(2,6),(3,5),(3,4)}

Paths that cover all c–uses for m:
1-2-3-5
1-2-3-4
1-2-6
5-2-3-5
5-2-3-4
5-2-6

Paths for all p-uses for m:
1-2-3
1-2-6
1-2-3-5

1-2-3-4
5-2-3
5-2-6
5-2-3-5
5-2-3-4

Paths that cover all c-uses for variable n:
1-2-3-5
1-2-3-4
4-2-3-5
4-2-3-4

Paths that cover all p-uses for variable n:
1-2-3
1-2-6
1-2-3-5
1-2-3-4
4-2-3
4-2-6
4-2-3-5
4-2-3-4

We notice that for example 5-2-3-5 is not covered by the test suite from b).we need a test case to cycle two time through the left loop.
TC3: m=12 n =3 expected output: 3
Also 5-2-6 is not covered. We need an extra test case:
TC4: m=12 n=6 expected output : 6
Also 4-2-3-5 Is not covered so we need another extra test case:
TC5: m=15, n=25, expected output: 5

e)A possible mutant is if (m<n) instead of if (m>n). TC2 will kill it. The program will never end and the output will be definitely not the expected one.