MÄLARDALENS HÖGSKOLA
ESKILSTUNA VÄSTERÅS

TENTAMEN
WRITTEN EXAMINATION

Ifylles av dig som är student *To be completed by the student*

Kurskod: DVT414 *Course code*

Tentamensdatum: 9/1 18 *Today's date*

Kursnamn: Software Verification and Validation *Course title*

Kod: *Code*
FDW-FTX

Ort: *City*
Västerås

Antal inlämnade blad: *Numbers of sheets you hand in*
15

Viktiga upplysningar *Important information*

- Skriv kod, kurskod och kursnamn på varje inlämnat blad
  Put your code, course code, and course title on every single sheet
- Börja alltid ny uppgift på nytt blad
  Always begin a new task on a new sheet

Ifylles av dig som är tentamensvakt *To be completed by the examination supervisor*

Skrivningen inlämnad kl: 12.15 Signatur: Y.J.

För akademins anteckningar *This part is for the department only*

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
4	5	5	5	4	4	5	5	5	5	7	10	10	10	10					

21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40

Betyg: G *Grade*

Summa: 94 *Sum*

Lärarsignatur: Samuel Ad *Teachersignature*

MÄLARDALENS HÖGSKOLA
ESKILSTUNA VÄSTERÅS

Kod: Code	Kurskod: Course code	Bladnr: Page nr
FDW-FTX	CDT-414	1

Uppgift nr: Task nr 1

Kursnamn: Course title: Software Verification and Validation

Poäng/notering
Point/note

1.- Difference between executing, debugging and testing

There is a clear difference between the three concepts. While debugging consists on eliminating bugs or deffects in the code (this task is done by the developer and requires an knowledge of the code and its structure), testing tries to identify the bugs or defects and reports them to the developer, but never corrects them (tester make the report but don't have inside knowledge since they didn't develop this). Lastly executing the code is a part of software testing and debugging, but while debugging ~~the~~ doesn't require executing ~~tester~~ to check is bugs have been removed, testing executes the code and checks the output that the code gives (actual output) with the expected value according to the requirements (expected output) to determine if ~~the~~ the test passes or fails

2.- Which of the following statements are correct and which not and why?

a) The goal of software testing is to prove that all defects are identified. ⇒ Incorrect ⇒ One of the main principles in testing is that a tester can't prove that a system is fault-free, so it is impossible to detect or identify all defects. Testing only proves that a bug have been identified but not all of them. This statement is more related to static-analysis

b) The goal of software testing is to detect as many faults as possible so that defects can be identified and corrected ⇒ Correct ⇒ This is one of the main goals of testing: find as many defects as possible to assure software quality. The task of testers is to identify these defects and report them to developers so they can fix them, but finding all can't be assured.

c) The goal of software testing is to prove that any remaining defect will not cause any failure. ⇒ Incorrect ⇒ the goal of software testing is to find as many error as possible and correct them, but if you don't find defects with your test cases, you can't apply counter measures. If you don't know what is causing the failure due to an unmasked error you can't do anything.

3. - What is the oracle in test case and how are "expected", "actual" and veredict corrected.

The oracle is a mechanism that allows a test to determine wether the test is passing or not (this except is the veredict) by executing the code and comparing the value that some function in the code calculates (this value is the actual output of test) with the value that is specified by the requirements (this value is the expected output)

In other words, the veredict is the result of comparing the value that the program should give (expected) with the value the program is actually giving (actual) and concluding if this two values are the same (test passes) or if they are different (test fails)

Test oracle is a desired characteristic for automating test cases.

4.- In which group of testing techniques does TDD belong?

TDD (Test-driven development) is not a testing technique. This is a software development approach that is ~~less~~ focused in the design. In this technique, tests are written before writting the code, so in this way, once the test have been desined, the code flows and adapts to the test, making the test pass (first the test always must fail)

5.- Explain the principle behind MC/DC coverage criteria.

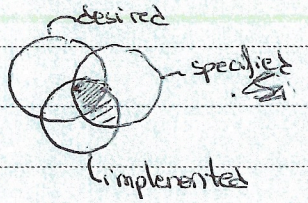
This coverage criterion is a logic-based one. In this criterion the focus is set on the predicate, which contains clauses (which can be evaluated to true and to false) → a condition in the code. The main principle of this coverage criterion is that each clause in a predicate should be set as active clause. With each active clause we have to find the combination of all other clauses that makes this clause directly affect the result with the non active clauses remained unchanged. If the active clause is true the result of the predicate should be true and if it is false, the predicate evaluation should be false.

We have to follow this (set one clause as the active one) for each clause in the predicate. Once we have done this, we remove test cases which are equal and the remaining are the minimum number of test that we need to generate so that our test fulfill 100% MC/DC. With this, we derive test cases that make each clause to be true or false (for each test case derived with this criteria)

6.- Differences and similarities between ~~the~~ specification-based and implementation-based technique?

. Differences ⇒ while specification-based testing (functional) is based on knowing if the program behaves according to the requirements (specifications) the implementation-based (structural) is ~~the~~ interesting in ~~the~~ knowing is the behaviour is the correct according to the structure of the code (what was implemented) The first one (specified) only takes into account inputs and outputs of the program (black-box testing) while the second one (implemented) is focused on the structure (white-box ~~(grey-box testing)~~)

. Similarities



} they both ~~the~~ focus on testing what was desired, specified and implemented and what was specified and implemented but never desired ~~Hemgjorda~~. These two areas are common for both of them

7.- How can we use coverage information for designing test cases?

Coverage gives us information that states that, for a given adequacy criterion, how many of the coverage items have been exercised from the total number of coverage items in the system. This information is given in the form of a percentage. Usually, we try to reach 100% coverage but sometimes the number can be lower. If we know how much coverage we want our test cases to have, and we have performed some kind of flow-graph, we can know the coverage items we want to be exercised and design the test cases so they fulfill this. Depending on the coverage type, it can be statement, branch or path. And if we have a coverage value lower than the intended one, this means that we need to add more tests in order to exercised the unexercised coverage item of our code and raise the coverage value.

8 - What is mutation testing? For what it could be useful?

Mutation testing is a technique that introduces ~~theme~~ deliberate bugs or defects in the code (these are called mutants) and checks if the ~~testcases~~ test cases we have detects these changes. If the error is detected, the mutant is killed but if it isn't discovered the mutant remains alive. Once all of this it's done, the mutation testing derived a ~~re~~report with how many mutants have been killed by our test cases, ~~# the~~ ~~Mutants~~ from the total number of mutants that were introduced. If the number of mutation coverage ~~(%)~~ is not 100% (all mutants were killed) it means that we have to add more tests in order to kill the remaining mutants.

This technique is really useful to check if the set of test cases we have derived is good enough for the code. Sometimes ~~tes~~ all our test cases are passing but ~~some~~ because the test we derived aren't the correct ones (we have a code with bugs but tests that aren't complete enough). In this cases, mutation testing inform us in order to be sure that the test we derived are good enough or if they need more

test are passing but flaws unnoticed

~~to sum up, the recreated case~~

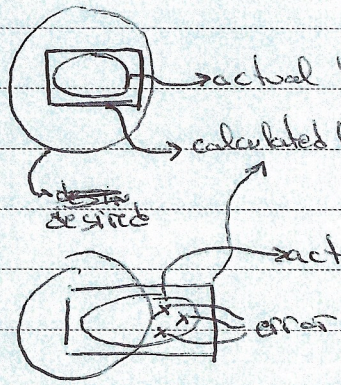
9.- What is a value analysis? Give to examples on what it can be used for?

A value analysis is one of the overall types of static analysis that there are. In this ~~the~~ type, we used a control graph that representes a mathematicall model of the analysis. With the value analysis we can determine the value the variables of a program can have in each statement/assignation of the program. This analysis can be focused, for example, in the value of a variable in the form of an interval of value that this variable can have.
For example it can be useful to show that a ~~var~~ variable has assigned the value 0, and if the next operation ~~of~~ with this variable is to divide something by this one (the 0) it can show us a division by 0 error.

Another example ~~useful~~ that shows that this analysis is useful is when a condition is imposed in the value of a variable (for example $y < 5$). In this case if the variable 5 is assigned a value between $[0, 4]$, the condition is never going to go to the false statement, so an infinite loop is detected.

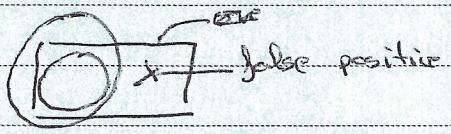
10.- What does it mean that a static analysis is safe? Why it is sometimes important to have a safe analysis?

Static analysis is safe because it is an analysis that overestimates the behaviour of a program/(s)tware product).



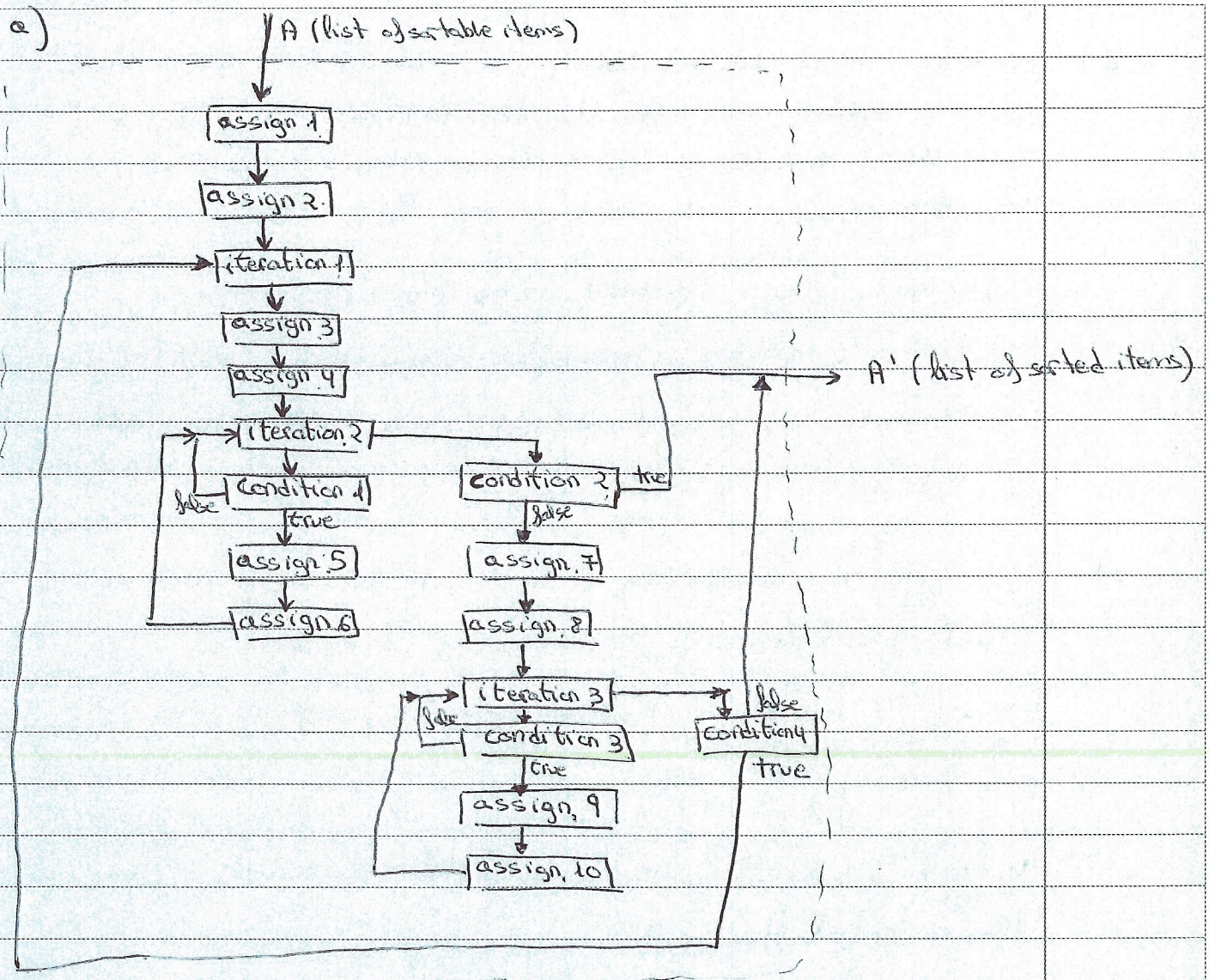
As we can see on the picture on the left, the static analysis ~~tot~~ overestimate the behaviour of the program, so if it states it is safe (it is ~~does~~ doesn't have errors) it is

The main problem with this analysis is that sometimes, the overestimation is high and warn us that there is an error where there really is not. This is called false positive.



Sometimes it is important to perform a static analysis because it ~~reve~~ not only reveal problems with the syntax or dead code. This analysis also discovers security vulnerabilities, so in case we are developing something that uses information of users or it's online, it's interesting to do the static analysis so our security vulnerabilities are as protected as posible.

11.- a)



(*) Additional anotations

- I have considered that the break statement goes out/finish do while loop and, as there is no statement after the do-while loop, this can be one of the two possible exits of the program
- I have also considered that the exit is the while condition, because when swapped == false, this is the exit (in some case the iteration statement is the exit but I think this more for "for iteration" type than in a "do-while" type iteration.
- Some assignations (assign 2 and assign 1; assign 3 and assign 4...) could be combined in one to simplify the graph but I prefered to leave all separated

b) In order to ~~me~~ statement coverage, all statements (or nodes) should be exercised ~~once~~ at least once by the test cases. In order to meet this, the set of coverage items exercised should be the following:

{assign 1, assign 2, iteration 1, assignation 3, assignation 4, iteration 2, condition 1, assignation 5, assignation 6, condition 2, assignation 7, assignation 8, iteration 3, condition 3, assignation 9, assignation 10, condition 4}

c) I'm going to consider that the input for the program is a list of sortable items composed by three integers.

The set of test inputs that when executed, satisfy statement coverage criterion is:

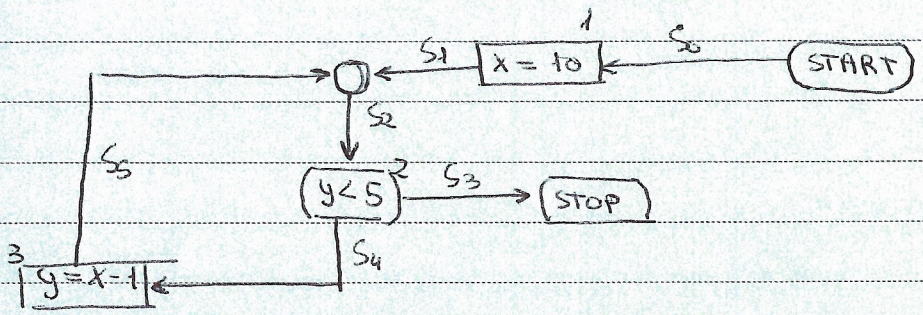
{3, 2, 1}

With this test input, we exercised all the statements of the previous control flow graph

d) To meet the branch coverage criterion means that all the branches existing in the previous control flow graph must be exercised. In order to do so, the set of coverage items are the following ones:

{assign 1, assign 2, iteration 1, assign 3, assign 4, iteration 2, condition 1 evaluates to true, condition 4 evaluates to false, assignation 5, assignation 6, condition 2 evaluates to true, condition 2 evaluates to false, assignation 7, assignation 8, iteration 3, condition 3 evaluates to true, condition 3 evaluates to false, assignation 9, assignation 10, condition 4 evaluates to true, condition ~~for~~ 4 evaluated to false}

12.-



$S_0 = \{(x,?),(y,?)\}$

$S_1 = f_1(S_0)$

$S_2 = S_1 \cup S_5$

$S_3 = f_2(S_2)$

$S_4 = f_2(S_2)$

$S_5 = f_3(S_4)$

$S_{0,0} = \emptyset \Rightarrow S_{0,1} = \{(x,?),(y,?)\}$

$S_{1,0} = \emptyset \Rightarrow S_{1,1} = \{(x,1),(y,?)\}$

$S_{2,0} = \emptyset \Rightarrow S_{2,1} = \{(x,1),(y,?)\} \Rightarrow S_{2,2} = \{(x,1),(y,?),(y,3)\}$

$S_{3,0} = \emptyset \Rightarrow S_{3,1} = \{(x,1),(y,?)\} \Rightarrow S_{3,2} = \{(x,1),(y,?),(y,3)\}$

$S_{4,0} = \emptyset \Rightarrow S_{4,1} = \{(x,1),(y,?)\} \Rightarrow S_{4,2} = \{(x,1),(y,?),(y,3)\}$

$S_{5,0} = \emptyset \Rightarrow S_{5,1} = \{(x,1),(y,3)\} \Rightarrow S_{5,2} = \{(x,1),(y,3)\}$

We reached a fix point, this is the end, equation are done

Set of reaching definitions $\Rightarrow$ $S_0 = \{(x,?),(y,?)\}$

$S_1 = \{(x,1),(y,?)\}$

$S_2 = \{(x,1),(y,?),(y,3)\}$

$S_3 = \{(x,1),(y,?),(y,3)\}$

$S_4 = \{(x,1),(y,?),(y,3)\}$

$S_5 = \{(x,1),(y,3)\}$

The analysis have that there is the risk of an uninitialized variable usage. The assignation 2 $\overset{(y<5)}{\text{test}}$ the value of y, but this value isn't initialized anywhere before. this could be a problem (S_3 and S_4 depend on this variable)

13.- if boarded & (temp < 0) & (take-off) < 30 min)
then spray-plane-with-chem;
else clear-for-takeoff;

(1) To fulfill clause coverage, we have to look at the predicate. It has three clauses: boarded
temp < 0
take-off < 30 min

} To meet this criterion also the three clauses should be evaluated to true and to false at least once.

The minimum number of test cases to do so would be 2

(2) Test case inputs 1 => { boarded = true ; temp = -4 ; take-off = 20 min }
Test case inputs 2 => { boarded = false ; temp = 10 , take-off = 40 min }

(3)

	boarded	temp < 0	take-off < 30 min	p
active clause =>	TRUE	TRUE	TRUE	TRUE
active clause =>	FALSE	TRUE	TRUE	FALSE
	TRUE	TRUE (active clause)	TRUE	TRUE
	TRUE	FALSE	TRUE	FALSE
	TRUE	TRUE	TRUE (active clause)	TRUE
	TRUE	TRUE	FALSE	FALSE

If we remove repeated combinations, we have the minimun number of test cases to fulfill 100% Active Clause coverage is 4

(4) Test case inputs 1 => { boarded = true ; temp = -4 ; take-off = 20 min }
Test case inputs 2 => { boarded = false ; temp = -4 ; take-off = 20 min }
Test case inputs 3 => { boarded = true ; temp = 10 ; take-off = 20 min }
Test case inputs 4 => { boarded = true ; temp = -4 ; take-off = 40 min }

14.- Testing exceptions using JUnit 4.0

In JUnit 4.0 there are three different ways of testing exceptions:

1) Expected parameter ⇒ We can declare, inside the test anotation (@Test) an expected parameter that the test should provide. An example of this configuration would be like the following one:

```
@Test(expected = IndexOutOfBoundsException.class)
public void testEmptyArray(){
    new ArrayList<Object>().get(0);
}
```

2) Try/catch ⇒ the second way of testing exceptions would be using a try/catch structure. In this one, we declare on the try some statement, that should launch an exception, and we use the cath to "catch" this exception. Once the catch has been reached, we compare the expected output with the actual output in this exception. In addition, we can add a fail statement under the try, in case the exception isn't launch due to some error. An example of this structure would be like the following.

```
@Test
public void testEmptyArray(){
  try{
      new ArrayList<Object>().get(0);
      fail("An exception was expected");
  }catch(ArrayIndexOutOfBoundsException e){
      assertEquals(e.getMessage(), "0 value.....");
  }
```

3) Rule => in JUnit there are different rules we can use, but the one that is usefull in this case is the ExpectedException rule. We can declare this expectedException previous the test, and in the test declare the especific exception we want to use. Later, we can use the expect and expectedMessage and write the code that should launch the exception. A example of this structure would be like the following:

```
@Rule ( new ex ExpectedException = ExpectedException.none();
@Test

public void ~~empty~~ testEmptyArray() throws IndexOutOfBoundsException{

    ex.expect('message/value...')
    ex.expectMessage('Message expected')
    new ArrayList<Object>().get(0);
}
```

this two are written this way but they should exist value and message according to the exception

15. - Equivalence partitioning on Triangle problem

- The first step is to identify the inputs, which are three integer values : a, b, c. These three inputs represent the sides of a triangle.
- One we have identified the inputs, we have to make partitions and for each partition, determine block with characteristics. The first partitition is based on the input value. thes will be is the value valid the block will be a valid value, an invalid value and null. Valid value are positive integers greater than zero, invalid value integers lower than zero with negative and null when no value is provided. The partitions and block are the following.

Characteristic	B_1	B_2	B_3
Is a valid value	B_{valid}	~~Invalid~~ Invalid	Null
Is b valid value	Valid	Invalid	Null
Is c valid value	Valid	Invalid	Null

The second partition we will make is based on the output The charcterist will be the number of integers that are equal

Number of equal sides	B_1	B_2	B_3
3	a=b=c		
2	a=b	a=c	b=c
0	a≠b≠c		

- The next step would be to combine the partitions and eliminate impossible combinations

Valid

Number of equal sides			
3	a = b = c a=valid, b=valid, c=valid	a = invalid b = invalid c = invalid	a = null b = null c = null
2	a = b a=valid, b=valid, c=valid	a = c a=valid, c=valid	b = c b=valid c=valid
			a = invalid b = invalid c = invalid
			c = null b = null c = null
0	a ≠ b ≠ c a=valid, b=valid, c=valid	a = invalid b = invalid c = invalid	a = null b = null c = null

After doing this, we should identify significant values. To do so, it would be interesting to do the boundry Analysis the frontier between valid and invalid partitions is 0, so it would be interesting to test this value, since the failures more likely to appear are the ore in the border.

Lastly we derived test cases the test for the valid cases are 5, but the ones related with invalid ore null combinations are ~~25~~ 9 (the invalid and null combinations are the same for the second partition). To reduce the 26 number we could do a pair-wise technique to reduce this or an each choice the second one is the one we are going to use

a	b	c	expected output
1	1	1	~~Scalene~~ Equilateral
1	1	2	Isosceles
1	3	1	Isosceles
2	1	1	Isosceles
1	2	3	Scalene
-	-	-	Incorrect triangle
0	-1	-1	Incorrect triangle

We have 7 test with each choice (ore invalid for all and ore null for all three)