Information page for written examinations at Linköping University

Examination date	2016-10-26
Room (2)	TER1 TER2
Time	14-18
Course code	TDDD04
Exam code	TEN1
Course name **Exam name**	Software Testing (Programvarutestning) Written examination (Skriftlig tentamen)
Department	IDA
Number of questions in the examination	10
Teacher responsible/contact person during the exam time	Kursansvarig: Ola Leifler Jour: Sam Le
Contact number during the exam time	Ola: 070-1739387, Sam: 0767-13 57 71
Visit to the examination room approximately	15:30
Name and contact details to the course administrator (name + phone nr + mail)	Anna Grabska Eklund, Email: anna.grabska.eklund@liu.se Phone: +46 13 282362
Equipment permitted	Dictionary (printed, NOT electronic)
Other important information	
Number of exams in the bag	

LiTH, Linköpings tekniska högskola
IDA, Institutionen för datavetenskap
Ola Leifler

Written exam

TDDD04 Software Testing

2016-10-26

Permissible aids

Dictionary (printed, NOT electronic)

Teacher on duty

Ola Leifler, tel. 070-1739387

Instructions and grading

You may answer in Swedish or English.

Your grade will depend on the total points you score on the exam. This is the grading scale:

Grade	3	4	5
Points required	50%	67%	83%

Important information: how your answers are assessed

Many questions indicate how your answers will be assessed. This is to provide some guidance on how to answer each question. Regardless of this it is important that you answer each question completely and correctly.

Several questions ask you to define test cases. In some cases you are asked to provide a minimal set of test cases. This means that you can't remove a single test case from the ones you list and still meet the requirements of the question. Points will be deducted if your set of test cases is not minimal. (Note that "minimal" is not the same as "smallest number"; even when it would be possible to satisfy requirements with a single test case, a set of two or three could still be minimal.)

You may find it necessary to make assumptions in order to solve some problems. In fact, your ability to recognize and adequately handle situations where assumptions are necessary (e.g. requirements are incomplete or unclear) will be assessed as part of the exam. If you make assumptions, ensure that you satisfy the following requirements:

- You have documented your assumptions clearly.
- You have explained (briefly) why it was necessary to make the assumption.

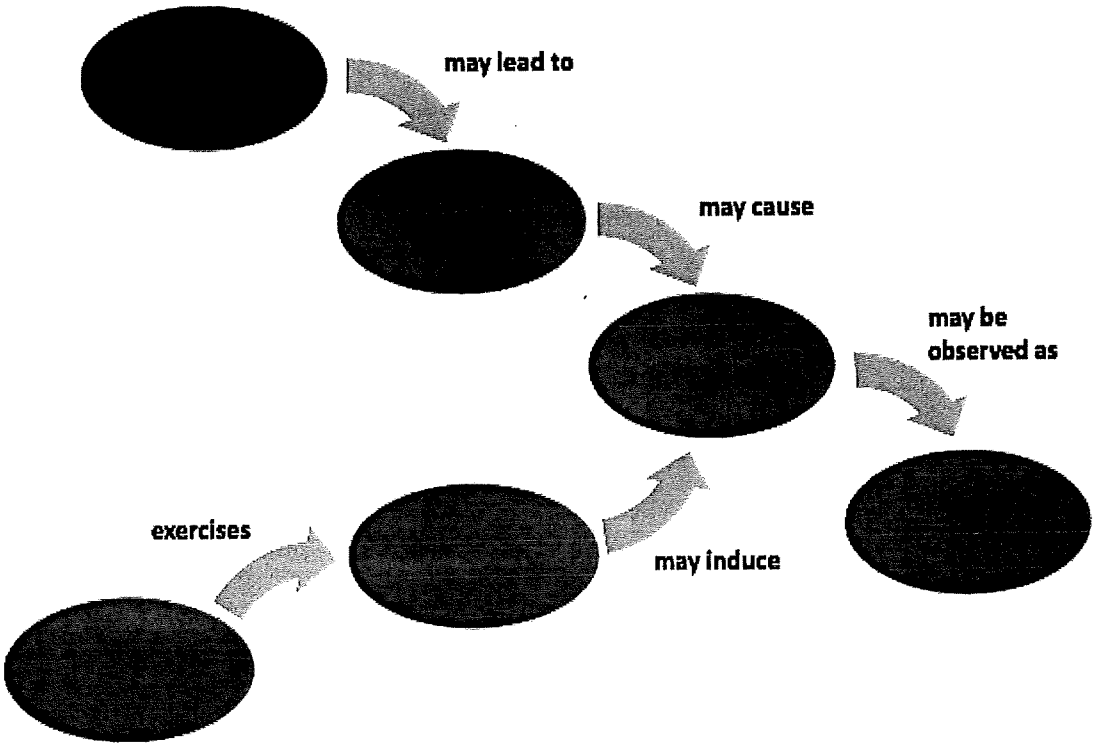
Whenever you make an assumption, stay as true to the original problem as possible.

You don't need to be verbose to get full points. A compact answer that hits all the important points is just as good – or better – than one that is long and wordy. Compact answers also happen to be quicker to write (and grade) than long ones.

Please double-check that you answer the entire question. In particular, if you don't give a justification or example when asked for one, a significant number of points will always be deducted.

1. Terminology (6p)

Fill in the proper (IEEE) software testing terms used in the following diagram. Give a short description of each term.



2. Coverage criteria (8p)

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

@Test
public void testSum(){
        Assert.assertEquals(3, sum(new int[]{3}));
}
```

a) Given the method sum above, and test testSum, do we obtain 100% decision coverage of the method sum? Justify your answer. (2p)

b) Coverage criteria can be applied to other software artifacts than code. Give examples of two such artifacts, and coverage criteria applied to them. (2p)

c) Even when applying the most demanding coverage criterion for software tests, we would need additional information to determine the quality of a test suite. Explain why, by providing a definition of test suite quality. (4p)

3. Defect classification (6p)

You are given the following description of a software defect:

"Company A's battery backup ran out of power because there was no low-power warning. The design of version 1 of the monitoring component *sys_mon* in the battery backup product *UPSMaster* did not include a warning for low battery power, despite the fact that this feature was specified in the requirements for *sys_mon*."

Fill in appropriate values for the columns in the table below (copy to a separate paper first) when you classify the defect above.

Fault/Defect	Attribute	Value
	Asset	
	Artifact	
	Effect	
	Mode	
	Severity	

4. True/False(6p)

Answer true or false:

a) 100% statement coverage is unnecessary for asserting the behavior of all code in a system under test.

b) 100% decision coverage is insufficient as a quality indicator of a test suite.

c) Mutation testing mutates tests to find the best tests for a piece of code.

d) xUnit type test frameworks run isolated test functions and rely on their return values to determine whether tests are successful.

e) According to recent scientific studies, structural (white-box) testing is more *effective* than inspections at finding design defects.

You get 1p for correct answer, 0p for no answer, and -1p for incorrect answer. However, you can not get negative points for this question.

5. Black-box testing (16p)

At your company SecurePass, you have created a password strength meter that shall indicate the strength of a password. The password strength meter checks that passwords have at least 8 characters, contain both letters and non-letter characters, and contain both upper-case and lower-case letters.

The function you are to test is called `check_password` and accepts a string parameter. It shall return one of the following constants, corresponding to the test outcomes:

A) INSUFFICIENT_LENGTH

B) LETTERS_AND_NON_LETTERS_REQUIRED

C) UPPERCASE_AND_LOWERCASE_REQUIRED

D) OK

Only one message can be returned.

Follow a suitable test case design method in order to test the given method. Justify your choice of test case design method, and any assumptions you make about the system. The system must be robust against different types of String values as input, and part of testing should assess that it does not crash.

You can receive up to 8 points for an appropriate representation of the problem, and up to 8 points for an appropriate translation of your representation into test cases. An excessive amount of test cases (based on the choice of test case design method) will lead to a deduction of points.

6. White-box testing (16p)

In this task, you are to test a method for segmenting text messages. Text messages such as SMS messages are limited to a number of characters, based on the number of bytes the characters require. English letters and common non-word characters (" ", ".", ",", etcetera) require one byte, and non-ASCII characters such as Swedish letters "å", "ä", and "ö" require two bytes.

Given the code example below,

1. create a set of basis paths for the code, and justify why your set is *sufficient*, (6p)
2. create test cases for 100% *decision coverage* based on the basis paths (6p), and
3. extend the test cases from step 2 to achieve maximal *Modified Condition/Decision Coverage* of the code. Explain MCDC as part of your answer. If you fail to achieve 100% MCDC, justify why. (4p)

Note that some of the paths will require several iterations of the loop, meaning several executions of decisions. A path corresponding in this case must ensure that the decisions required are taken *at some point during the execution*.

```
/**
 * Determine the number of chunks a text message has to be sent in, if each
 * chunk can contain 10 bytes, including a special character to denote
 * multi-part messages.
 * @param message
 * @return the number of chunks
 */
public static int getChunks(String message) {
        IntStream chars = message.chars();
        int bytes = 0;
        int chunks = 1;
        int[] int_array = chars.toArray();
        for (int i = 0; i < int_array.length; i++) {
                int j = int_array[i];
                if (j > 127) {
                        bytes += 2;
                } else {
                        bytes++;
                }
                // If we have filled
                // the current 10 byte "chunk"
                if (bytes >= 10 && i < int_array.length - 1) || bytes > 10) {
                        bytes++; // One byte required to denote a multi-chunk
                                                // message
                        chunks++;
                        bytes -= 10;
                }
        }
        return chunks;
```

}

7. Integration testing (6p)

a) Name the types of additional scaffolding code that may be needed for integration testing. Explain which tradeoffs are made with respect to the ability to *detect* faults, and the ability to *locate* faults when using scaffolding code. (3p)

b) Name and explain two critical factors for successful integration testing. (2p)

c) How can mock objects be used in integration testing? Give a small example. (1p)

8. Exploratory testing (4p)

a) In terms of exploratory testing, what is a *charter* and a *session*? What is the difference between the two? (2p)

b) Explain the advantages and disadvantages of automated test scripts (checking) versus exploratory testing (exploring) with respect to costs, and how defects are detected and located. (2p)

9. Regression testing (3p)

Three main techniques have been proposed in the literature for the purpose of optimizing regression testing. Describe these main techniques, and explain how they can be applied to optimize the test cases in the example below. In the example, we have extended the code given in task 2 with functionality to check overflow, along with new some test cases:

```
public static int sum(int[] x) {
        int sum = 0;
        for (int i = 0; i < x.length; i++) {
                int tmp = sum + x[i];
                // Check overflow
                if (tmp < sum) {
                        throw new RuntimeException("overflow");
                } else {
                        sum += x[i];
                }
        }
        return sum;
}

@Test
public void testSum(){
        Assert.assertEquals(3, sum(new int[]{3}));
}

@Test
public void testSum2(){
        Assert.assertEquals(0, sum(new int[]{}));
}

@Test(expected=RuntimeException.class)
public void testSum3(){
        sum(new int[]{Integer.MAX_VALUE, Integer.MAX_VALUE});
}
```

10. Model-based testing (4p)

1. Explain when model-based testing might fail to distinguish between two versions of an application. (2p)
2. Explain when this could be a good thing. (2p)