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.

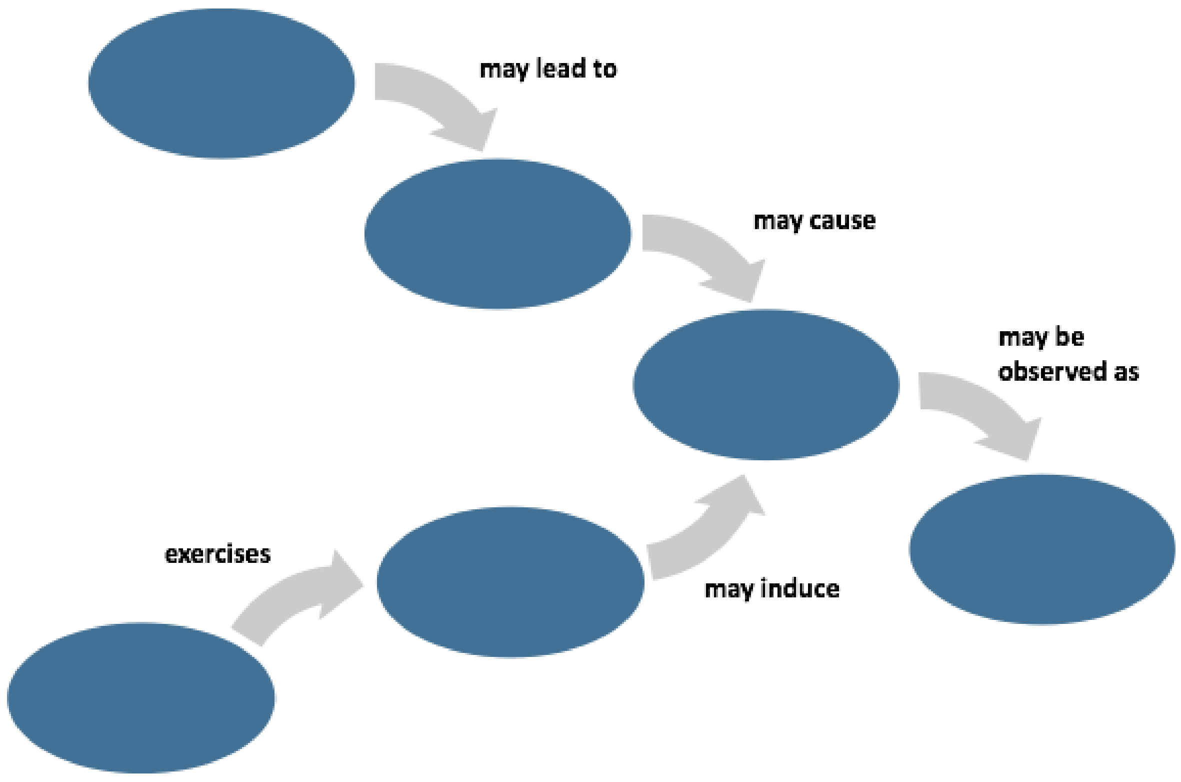
Whenever you make an assumption, stay as true to the original problem as possible.

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.



Errors are human mistakes, that may lead to defects or faults. Defects or faults are the representation of an error, in requirements, design documents or code. Such defects may cause a failure during program execution. A failure during execution may be observed as an incident. As an example, if the requirements of a web application like Dropbox do not specify how to handle files with non-ascii file names, uploading a file with filename “åäö” might cause the program to name the file “aao”, which would result in a later search for that file to come up with an empty set of matches.

Tests exercise test cases that may induce those very same failures. For a more complete description of each term, see the lecture slides from lecture 1.

2. Coverage criteria (8p)

```
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 function `sum` above, and test `testSum`, do we obtain 100% decision coverage? Justify your answer. (2p)

   Solution: A for-loop that we execute at least once will execute both the true and false branches that result from the decision, so yes, we will achieve 100% decision coverage.

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)

   Solution: requirements and models of software are artifacts that may be covered by tests that cover all requirements or all transitions in a state transition graph model.

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)

   Solution: the quality of a test could be defined as the probability that a test will be able to distinguish working software from defect software, in which case the assertions are critical to assess in addition to the coverage. Thus, even with 100% path coverage, we do not require that any assertions are made, or that they are able to distinguish original code from altered code. Using mutation testing, we can use mutation score as an alternative metric for test suite quality, although that assumes that all deviations from the original code are assumed faulty.

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	

Solution:

Fault/Defect	**Attribute**	**Value**
Defect	Asset	sys_mon
Defect	Artifact	Design document
Defect	Effect	functionality
Defect	Mode	missing
Defect	Severity	high

4. True/False(5p)

Answer true or false:

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

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

c) Mutation testing mutates tests to find the best tests for a piece of code. FALSE, it mutates code in the SUT

d) xUnit type test frameworks run isolated test functions and rely on their return values to determine whether tests are successful. FALSE, they rely on exceptions being thrown when assertions are violated.

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

Suggested solution:

As the conditions and actions taken seem to be composite actions, we will model them using more simple rules, so as to devise test cases easier.

C1: password length

C2: letters in password

C3: non-letters in password

C4: uppercase letters in password

C5: lowercase letters in password

We assume that we can prioritize the output constants in the order listed in the assignment.

Using these, we can create a decision table. In R4 and R5, we will have to set C3 to Y, otherwise we could potentially create the same rule as R3. Also, in R4, C5 has to be Y as C4 is set to N. "-" is indterpreted as "don't care".

	R1	**R2**	**R3**	**R4**	**R5**	**R6**
C1	< 8	>= 8	>= 8	>= 8	>= 8	>= 8
C2	-	N	Y	Y	Y	Y
C3	-	-	N	Y	Y	Y
C4	-	-	-	N	Y	Y
C5	-	-	-	Y	N	Y
Action	INSUFFICIENT_ LENGTH	LETTERS_ AND_ NON_ LETTERS_ REQUIRED	LETTERS_ AND_ NON_ LETTERS_ REQUIRED	UPPERCASE_ AND_ LOWERCASE_ REQUIRED	UPPERCASE_ AND_ LOWERCASE_ REQUIRED	OK

Based on this decision table, we should select appropriate test cases for each of the rules. For the rules R1 and R2, there are several interesting values for string length to consider: the default value of a string, the empty string, and a string with length 7, 8 and 9 characters. The default (null) value of a string type object is essential to test to ensure robustness. We *make the assumption* that a null input should result in the output INSUFFICIENT_LENGTH. Uppercase characters may also appear anywhere in the password, not only as leading characters.

	T1	T2	T3	T4	T5	T6	T7	T8	T9

Input	null	“”	“abcd123”	“12345678”	“abcdefghi”	“p4ssw0rd”	“P4SSW0RD”	“P4ssw0rd”	“p4ssW0rd”
Output	INSUFFICIENT_LENGTH	INSUFFICIENT_LENGTH	INSUFFICIENT_LENGTH	LETTERS_AND_NON_LETTERS_REQUIRED	LETTERS_AND_NON_LETTERS_REQUIRED	UPPERCASE_AND_LOWERCASE_REQUIRED	UPPERCASE_AND_LOWERCASE_REQUIRED	OK	OK
Comment	No error expected, R1	R1	Length = 7, R1	Length=8, R2	Length=9, R3	Length=8, R4	Length=8, R5	Length=8, leading uppercase, R6	Length=8, non-leading uppercase, R6

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, due to the number of bytes the characters require. Characters in the ASCII set (English alphabet among other characters) require one byte, and non-ASCII character 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 below, 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)

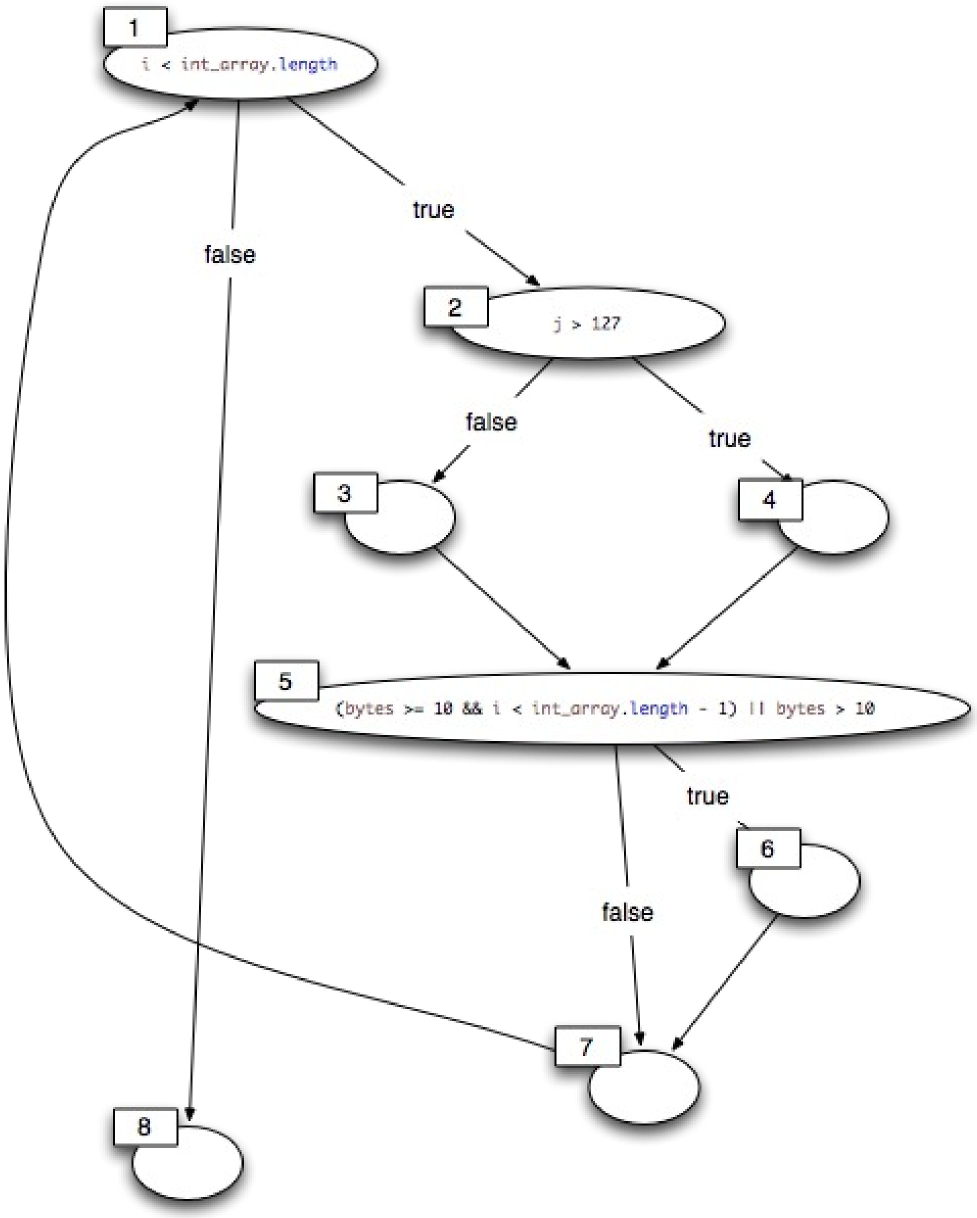
Note that some of the paths will require several iterations of the loop, meaning several executions of decisions. A path corresponding in this case must ensure that the decisions required are taken *at some point during the execution*.

```java
/**
 * 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;
```

```
    }
```

Suggested solution

We have three decisions points, which translate into four basis paths (cyclomatic complexity = 4). Given the CFG like this:



We might start with 1-2-4-5-6-7-1-8 as path A and alter each decision point on this path. Based on A, we alter decision 1 so we obtain path B which is 1-8. Then, we create a new path C by altering the decision at node 2, and receive 1-2-3-5-6-7-1-8. Path D is obtained by altering decision 5: 1-2-3-5-7-1-8.

We describe our final set of paths by the outcomes of decisions taken in those paths

Path	`i < int_array.length`	`j > 127`	`bytes >= 10`
A	True	true	true
B	false	-	-

C	True	False	True
D	True	True	False

Each path constrains the input, so we can create test cases based on the path.

Test Case	Constraint
A	Input length longer than 10 bytes, including at least one non-ASCII character [å,ä,ö]
B	Empty input string
C	No non-ASCII characters in input, longer than 10 bytes
D	Input length between 1 and 9 bytes, inclusive

Test cases for 100% decision coverage:

Test Case	Input	Result
A	“asdf åäö”	2
B	“”	1
C	“qwerty asdf”	2
D	“asd”	1

When we want 100% modified condition/decision coverage, we need to make sure that all conditions in the composite condition `bytes >= 10 && i < int_array.length - 1) || bytes > 10` are demonstrated to independently affect the outcome of that decision. So, we need the cases below. However, A.3 is clearly impossible as `bytes` cannot be smaller than and greater than 10 at the same time.

Test case	`bytes >= 10`	`i < int_array.length - 1`	`bytes > 10`	`(bytes >= 10 && i < int_array.length - 1) || bytes > 10`
A.1	True	true	false	true

A.2	true	false	True	true
A.3	False	True	True	True
D.1	True	false	False	false
D.2	False	True	False	False

This would give us test cases, based on the original ones that we extend:

Test Case	Input	Result
A.1	“asdf åäö”	2
A.2	“åäöåasdf”	2
A.3	---	---
B	“”	1
C	“qwerty asdf”	2
D.1	“asdfqwerty”	1
D.2	“asdf”	1

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)

Solution: We wish to use *drivers* to test low-level functionality, and *stubs* to test high-level functionality. Using either stubs of drivers, we will be able to locate faults better, but faults that result from the interaction of live components may not be properly detected.

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

Solution: A proper test plan, which takes into account dependencies and timing constraints, and communication among team members.

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

Solution: Mock objects are useful as stubs, and provide canned responses to test high-level functionality. See lab 1.

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)

Solution: A charter is a short description of the aim of testing, as in testing performance of a web application. A session describes a set of interactions performed with the application with the charter as the aim.

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)

Solution: Exploratory testing costs more during execution, but automatic tests cost more during setup and maintenance. Exploratory testing may cover more high-level features, and detect more types of defects than scripted tests. Faults may be more difficult to locate precisely with exploratory testing than with scripted, automated tests.

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:

```java
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});
    }
```

Solution: One may do *Test Case Selection*, as in, selecting test cases to run based on modifications made. In that case, `testSum3` might be the only test run for the change introduced. *Test Case Prioritization* requires us to prioritize test cases, in which case `testSum3` should be run first, followed by `testSum` and `testSum2` in some order. Using *Test Suite Reduction*, we may see that `testSum2` does not exercise new paths in the code, and may be a candidate for permanent removal.

10. Model-based testing (4p)

1. Explain when model-based testing might fail to distinguish between two versions of an application. (2p)

   Solution: MBT models only certain aspects of an application. If we have used preconditions/effects to model the behavior of a search algorithm, it will not be able to distinguish between a linear and a logarithmic search as long as the preconditions and effects are kept intact.

2. Explain when this could be a good thing. (2p)

   Solution: If we wish to reuse our test cases for different implementations of the same search algorithm, a single model is obviously advantageous to check that invariants such as pre- and postconditions hold even with a new implementation.