Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2015-06-03
Sal	U1,U3,U4,U6
Tid	8-12
Kurskod	TDDD04
Provkod	TEN1
Kursnamn/benämning	Programvarutestning
Institution	IDA
Antal uppgifter som ingår i tentamen	12
Antal sidor på tentamen (inkl. försättsbladet)	7
Jour/Kursansvarig	Ola Leifler
Telefon under skrivtid	070-1739387
Besöker salen ca kl.	10:00
Kursadministratör (namn + tfnnr + mailadress)	Anna Grabska Eklund
Tillåtna hjälpmedel	Dictionary (printed, NOT electronic)

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

Written exam

TDDD04 Software Testing

2015-06-03

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. The maximum number of points is 86. This is the grading scale:

Grade	**3**	**4**	**5**
Points required	42	55	70

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 (4p)

Explain what “white-box testing” is. Explain one test case design methodology that can be used for white-box testing. (4p)

Testing the implementation, not the specification. Control-flow-based testing to achieve full statement coverage is one test design methodology.

2. Coverage criteria (8p)

a) Order the following coverage criteria with respect to their requirements on test cases in ascending order:

1. Path Coverage
2. Statement Coverage
3. Branch Coverage

(2,3,1)

(2p)

b) Explain when *multiple condition/decision coverage* equals *statement coverage* (2p)

When there are no composite conditions in a decision (conditions combined with boolean operators), and all decision outcomes add statements to the control-flow graph (CFG).

c) Explain why some coverage criteria cannot be quantified in the same way as e.g. statement coverage, and give an example to justify. (4p)

Path coverage, for example, may not be determined if the unit under test contains loop constructions that we may not analyze fully how many iterations they will require during the execution of the code. Then, determining the ratio of covered paths to the total number of paths becomes impossible.

3. Test automation (6p)

Explain advantages that a test automation framework such as CPPUNIT/JUnit have over each of the options below.

Name one advantage and one disadvantage of creating tests for use of a test automation framework instead of

a) performing exploratory testing (2p)

xUnit test frameworks are more economical to use for checking that known faults do not reappear, whereas exploratory testing can highlight new areas with potential problems

b) generating test cases through symbolic execution (2p)

The two techniques are really complementary, in that generated test cases can be executed with a unit test framework. However, generated test cases can have no information about expected results, and assertions about correct behavior have to be made in order for the test cases to be useful.

c) stepping through a debugger (2p)

A debugger can provide very precise information about the state of a program when a crash occurs, but it requires manual intervention during execution, which is not very economical for large sets of tests, where information from failed assertions may be a more efficient way to probe an application for faults.

4. True/False(6p)

Answer true or false:

a) One goal of software testing is to verify that the system under test (SUT) contains no errors. FALSE

b) MM-Paths can be used for both unit testing and integration testing. FALSE

c) A bug is the observable effect of executing a fault. FALSE

d) Define-use-kill data-flow patterns may only used for static program analysis. FALSE

e) Decision-table testing subsumes Model-based testing. FALSE

f) You can automate exploratory testing. FALSE

(It's not worth guessing: you get 1p for correct answer, 0p for no answer, and -1p for incorrect answer; you can get negative points on this question.)

5. Black-box testing (16p)

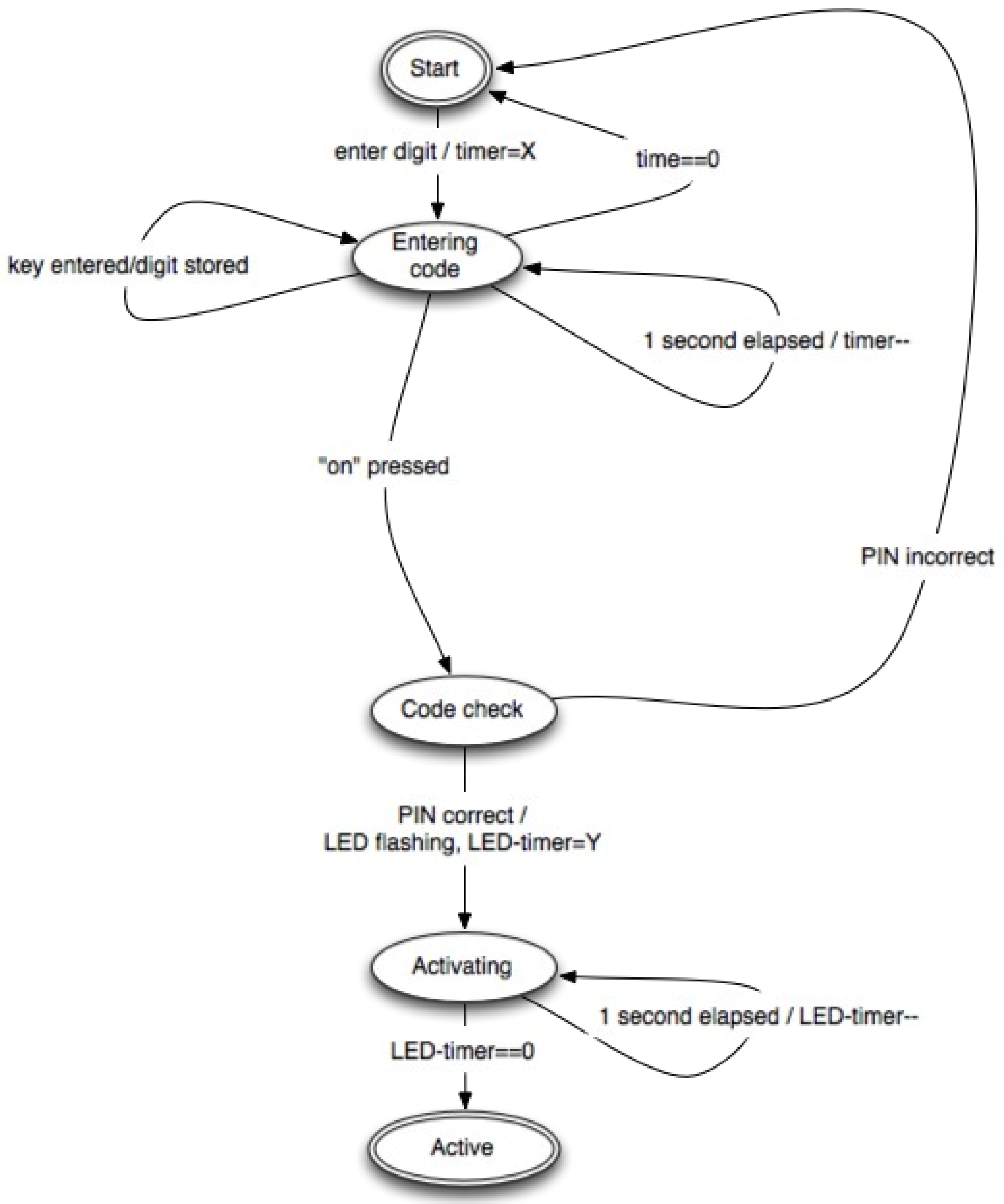
You are to test a home alarm system with the following description:

The alarm is set by entering a four-digit code and pressing the "on" button. When activating the alarm, the code has to be entered in full within five seconds or else the system is reset. If the correct code is entered, then, after a given time period, the alarm is activated. Until the alarm is activated, a red LED will flash to indicate that the alarm is about to be activated.

Use a suitable representation to describe the behaviour of this system so that it can be tested using a black-box testing technique. Derive a set of test cases based on the representation, and use a suitable method and metric to determine the character and minimal number of test cases.

We draw a state chart to describe the system.

Such a state chart can be used to devise test cases for coverage of either the nodes, edges or paths in the graph.



6. Symbolic execution and white-box testing (10p)

Explain how symbolic execution works and how it can be used to generate test cases in the following example. Also, provide a set of test cases to obtain 100% branch coverage.

For full points, explain how symbolic execution works in general, how it applies specifically to test case generation, and whether generated test cases are complete or need additional information to provide useful information. Also, describe how branch coverage can be obtained, and provide a clear description of how your test cases achieve branch coverage.

```
public static double calculateBill(int usage) {
        double bill = 0;
        if (usage > 0) {
                bill = 40;
        }
        if (usage > 100) {
                if (usage <= 200) {
                        bill += (usage - 100) * 0.5;
                } else {
                        bill += 50 + (usage - 200) * 0.1;
                        if (bill >= 100) {
                                bill *= 0.9;
                        }
                }
        }
        return bill;
}
```

7. Model-based testing (6p)

Explain the workflow involved in model-based testing with state chart diagrams compared to script-based testing. Describe what may and may not be automated with respect to test design and execution.

Model-based testing uses a model, possibly a state chart as in assignment 5, to guide the test execution. Based on the model, a programmer creates the concrete steps necessary to achieve the effects specified by the nodes and edges in the model, and the model-based test execution proceeds to perform all transitions in the model according to a heuristic for best coverage of nodes, edges or paths (if the model is based on finite state charts)

8. Integration testing (6p)

a) State one advantage thread-based integration testing has over other methods. (2p)

It provides more realistic test cases that integrate the components necessary for a certain feature or use case (execution thread)

b) In top-down integration testing, how many *drivers* are needed at most? (2p)

0 :)

c) In bottom-up integration testing, how many *test sessions* are needed at most? (2p)

Nodes-leaves+edges

9. Exploratory testing (6p)

a) Explain the difference between exploratory testing and ad-hoc testing. (2p)

Exploratory testing follows a charter, and is guided by uses cases and domain knowledge.

b) How can exploratory testing be justified as a test method? (2p)

It provides new information about program behavior that is difficult to obtain otherwise.

c) In terms of exploratory testing, what is a *tour*? (2p)

The type of tour defines the heuristic used to guide a tester in selecting features or behaviors to test, and may be based on the manual or difficult corner cases.

10. Modified condition/decision coverage (10p)

Specify a minimal set of test cases for the following function that result in 100% *modified condition/decision coverage*.

```
int rules(int a, int b, int c) {
  if (a < 3 || b > 0) {
    if (b < 1 && c > 2) {
      return b+c;
    }
    return a+c;
  } else if (c > 3 || a > 2) {
    return a-c;
  }
  return a;
}
```

To determine MCDC coverage, we will need to establish how each condition value affects the values of a, b & c. Orange background: unnecessary value according to MCDC, green: expression true, gray: expression false, dark gray: impossible.

Conditional expression and evaluation	T T (1)	T F (2)	F T (3)	F F (4)
X: (a < 3 || b > 0)	a < 3, b > 0	a < 3, b <= 0	a >= 3, b > 0	a >= 3, b <= 0
Y: (b < 1 && c > 2)	b < 1, c > 2	b < 1, c <= 2	b >= 1, c > 2	b >= 1, c <= 2
Z: (c > 3 || a > 2)	c > 3, a > 2	c > 3, a <= 2	c <= 3, a > 2	c <= 3, a <= 2

Also, we need to establish how X, Y and Z relate to one another:

- X(4) is incompatible with Y(1,2,3) as X will have to evaluate to TRUE in order for the program to evaluate Y
- X(4) is incompatible with Z(3)
- X(4) and Z(2) are incompatible, so X(4) implies Z(3).
- X(2) and Y(3) are incompatible
- X(3) and Y(2) are incompatible

With this information, we can deduce that the following test cases are necessary to achieve maximum MCDC coverage (All green and gray cells chosen at least once):

Test case	Cells chosen	A	B	C	Expected result
1	X2, Y1	2	0	3	3
2	X2, Y2	2	0	2	4
3	X3, Y3	3	1	3	6
4	X4, Z3	3	0	3	0

11. Testing case selection (4p)

Explain how to determine the quality of a test suite, by reasoning about how to evaluate different kinds of qualities of different kinds of software products. For full points, you need to reason about how coverage metrics may or may not be used to determine test suite quality.

No code coverage metric determines that the correct assertions have been made about program behavior, but only that code has been executed. In the general case, to the extent each test in a test suite provides new, valuable information about the state of an application, it can be considered useful. Mutation testing is an alternative way of asserting that program code is being tested correctly by automated tests, and makes sure that test results differ when a program is altered. To the extent a test can differentiate a program that is believed to be correct, and all other variants of a program, it can be considered a good test. A mutation score may be used to determine the ratio of differentiated programs (failed tests) to the total number of mutations (executed tests), much as coverage metrics are used to determine the ratio of executed code to total code.

Also, assessing software properties such as security, usability or performance typically requires domain-specific heuristics for determining how well a test suite covers security evaluation for instance.

12. Test automation: True/False (4p)

Answer the following questions true or false:

a) The most commonly automated steps of a software testing workflow are test execution and visualization. TRUE

b) Continuous Integration reveals *more types of errors* during integration testing. FALSE

c) Continuous Integration reveals *errors earlier* during integration testing. TRUE

d) A failing automated test equals an error in the program under test. FALSE

(It's not worth guessing: you get 1p for correct answer, 0p for no answer, and -1p for incorrect answer; you can get negative points on this question.)