TECHNISCHE UNIVERSITEIT DELFT
Faculteit Elektrotechniek, Wiskunde en Informatica

TU Delft

Exam IN3205
Software Quality and Testing

Version 1.1, April 8, 2010

- *This is trial exam, aimed at helping you to master the material. The best way to use it is by first trying the questions yourself, using the answers in a later stage.*
- There are 8 questions worth of 33 points in total.
- Total number of pages: 11.
- Use of books and readers is not allowed
- Write clearly and avoid verbose explanations: Points may be deducted for unclear or sloppy answers
- You can answer in English or in Dutch
- Please list your answers in the right order
- The tentative grading scheme is:

Question:	1	2	3	4	5	6	7	8	Total
Points:	3	6	6	5	3	2	2	6	33
Score:									

- If p is the number of points you score, the exam grade E will most likely be determined by

$$E = 1 + 9 * p/33$$

- Your final grade F is determined by the average of your labwork L and the exam E:

$$F = (E + L)/2$$

  Note that you can only pass if both $E \geq 5.8$ and $L \geq 5.8$.

GOOD LUCK!

1. (3 points) While working for software company *C*, you created a test and integration plan for software system *X*. Key steps in your plan include ensuring (as close as you can get to) 100% statement coverage for the components used or developed, and a period of 4 weeks intensive integration testing.

   Your manager *M* studies the plans, and proposes to reduce the period of integration testing to just a single week, based on the argument that with 100% statement coverage there is no need for so much integration testing.

   Write a memo of at most 150 words to your manager, in which you include at least three good reasons in defense of your original plan.

   Solution: Dear M.,

   Recently you proposed to reduce the time available for integration testing by 75%, based on the argument that the underlying units are tested with test suites that achieve close to 100% statement coverage.

   While statement coverage is a minimal requirement for good test suites, there are many faults that will not be found. In practice, most faults are related to integration problems, caused by erroneous interface design, different or unexpected usage profiles, or incorrect use of interfaces.

   If you need to reduce the time to market, a better alternative would be to move one or two features to the next release, thus reducing development and testing effort.

   In short, reducing integration test effort is introduces a high risk on product failure; in my professional opinion the best way to avoid this risk is to reduce the feature set of the next release.

   Yours sincerely,
   me

   Note *Take care of the proper structure (introduction, analysis, alternative solution, summary; write clearly and convincingly; use the right arguments; and come up with a good alternative.*

```
/**
  * Determine if the other cell is an immediate
  * neighbour of the current cell.
  * @return true iff the other cell is immediately adjacent.
  */
public boolean adjacent(Cell otherCell) {
   boolean result;

   boolean xdiffersone = Math.abs(getX()-otherCell.getX()) == 1;
   boolean ydiffersone = Math.abs(getY()-otherCell.getY()) == 1;

   if ((xdiffersone && !ydiffersone) || (!xdiffersone && ydiffersone)) {
       result = true;
   } else {
       result = false;
   }
   return result;
}
```

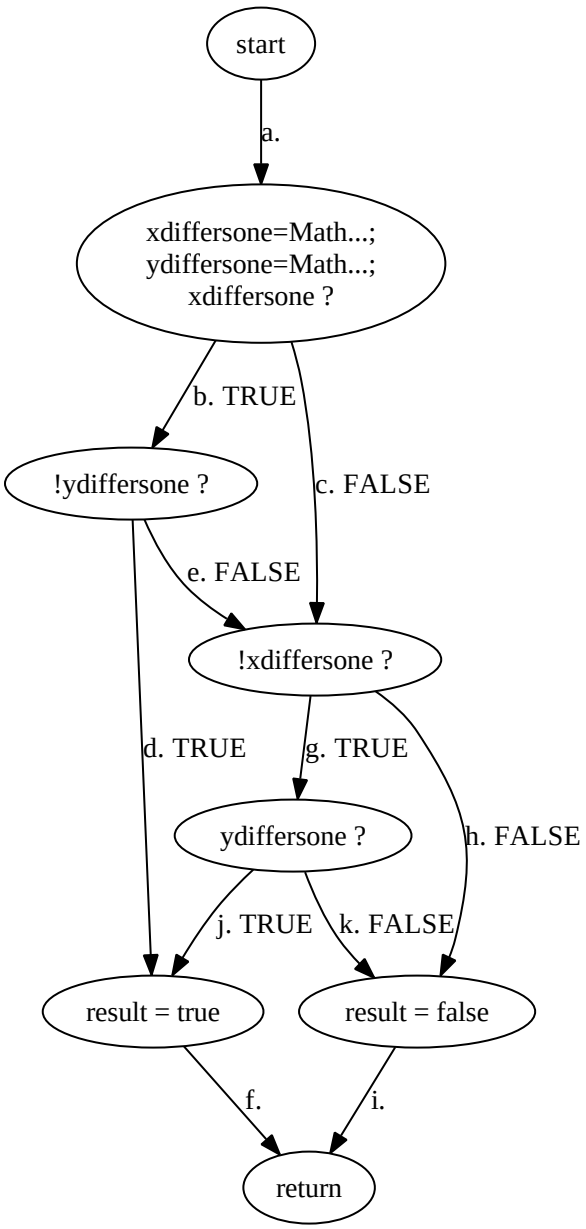
Figure 1: Implementation of adjacent

2. While implementing the "adjacent" method in class Cell from JPacman, your partner proposes the solution from Figure 1. The intent of the method is that it returns true if the cell is immediately next to `otherCell`, and false in all other cases.

   While studying the implementation, you quickly discover a an incorrect case (at distance greater than one), which was not covered by the two test cases provided by your partner. You therefore deicde to analyze whether there is a test adequacy criterion with which your partner should have found the bug.

   (a) (1 point) Construct the control flow graph of the "adjacent" implementation, taking Java's short-circuit evaluation into account.

   Solution:



Note: Single entry, single exit, group statements, separate nodes for each basic condition.

(b) (1 point) How many paths from start to end does this graph contain?

Solution: 7

(c) (1 point) How many paths are needed at least to achieve statement coverage?

Solution: 2

(d) (1 point) How many paths are needed at least to achieve branch coverage?

Solution: 4

(e) (1 point) Are any of the paths in the graph infeasible? If so, which ones?

Solution: The following paths are infeasible:

1. a–b–e–g is impossible, as it would require "xdiffersone" and "!xdiffersone" to be both true.
2. a–c–h is impossible, as it would require "xdiffersone" and "!xdiffersone" to be both false.
3. a–b–e–g–k is impossible, as it would require "ydiffersone" and "!ydiffersone" to be both false

Note that there is no path requiring "ydiffersone" and "!ydiffersone" to be both true.

(f) (1 point) Does adhering to either statement or branch coverage necessarily lead to the detection of the fault? Explain your answer.

Solution: No. Testing for a difference with one is wrong, and this fault is not identified via branch / statement coverage.

3. Next, you decide to analyze the intended behavior of the adjacent method from Figure 1:

   (a) (1 point) Create a decision table in which you indicate what the result of the adjacent method should be. As basic conditions, disinguish the $(2*3=6)$ cases that the absolute difference between the x (respectively y) coordinate of the current and the other cell equals zero, equals one, and is greater than one. (You may want to give names $C_1, C_2, \ldots$ to the columns, and $R_1, R_2, \ldots$ to the rows, to make it easier to refer to them later on.)

   Solution:

		C1	C2	C3	C4	C5	C6
R1	absdx=0	T	F	F	-	T	F
R2	absdx=1	F	T	F	-	F	T
R3	absdx>1	F	F	T	-	F	F
R4	absdy=0	F	T	-	F	-	F
R5	absdy=1	T	F	-	F	F	-
R6	absdy>0	F	F	-	T	-	-
	Outcome	T	T	F	F	F	F

   Note: See section 14.3, Figure 14.4, P&Y

   (b) (1 point) Are there any constraints on permitted combintations of the basic conditions? If so, which ones?

   Solution: Only one of the three dx (R1-R3) resp. dy (R4-R6) values can be true in each column.

   (c) (1 point) How many test cases are needed at least to achieve *basic condition coverage*? Explain your answer.

   Solution: 6: One for each column on the table.

   (d) (1 point) How many test cases are needed at least to achieve *compound condition coverage*? Explain your answer.

   Solution: Given the six basic conditions in the rows, in principle there are $2^6 = 64$ columns, but due to the given constraints this reduces to only $3*3=9$.

   (e) (2 points) How many test cases are needed at least to achieve Modified Condition/Decision Coverage (MC/DC)? Explain your answer.

   Solution: In MC/DC, the starting point is to have all columns as test cases. Next, we verify that each condition (row) independently affects the outcome of the decision: i.e., if there are two columns which only differ in the true/false value for the given condition and in the resulting action. For conditions for which this is not the case, we add a column to meet the criterion.

   For the given decision table, due to the constraint, it is imposibble to just change, e.g., R1 to true without changing R2/R3.

   For the given decision table it is not necessary to add columns to achieve MC/DC coverage. To see why this is the case, we select for each row two columns which only differ in the outcome and the value for the given condition.

Condition	True Column	False Column
R1	C1	C3
R2	C2	C5
R3	C3	C1
R4	C2	C6
R5	C1	C5
R6	C4	C1

4. Next, you decide to try to rethink the robustness of the `adjacent` method.
   (a) (1 point) Identify assumptions the `adjacent` method relies on.

   Solution: The method relies on the assumption that `otherCell` is not null. Furthermore, it assumes that the cells are on the same board, and that the x and y coordinates are within the boarders.

   (b) (1 point) Provide a design of the interface of this method that adheres as much as possible to the principles of defensive programming.

   Solution: We return "false" as default value if our assumptions are violated. Thus we add a conditional to check for our assumptions, and return false if they do not hold.

   (c) (1 point) Provide a design that adheres as much as possible to the principles of design by contract.

   Solution: We turn the assumptions into preconditions of the method, which we can verify by means of an assert statement.

   (d) (1 point) Let class A be a subclass of class B, and assume that this subclass relation adheres to the principles of design by contract. Furthermore, let m be a method in B that is overridden in A. Which of the method implementations, $A.m$ and $B.m$, is the most defensive? Explain your answer.

   Solution: A method is more than defensive than another if it can handle more cases. If it can handle more cases, there are fewer preconditions that have to be met before the method can be invoked. Thus the method with the weakest precondition is the most defensive.

   In design by contract, weaker preconditions are in the subclasses, and hence A.m is the most defensive.

   In general, a more defensive implementation has fewer preconditions (instead of assertions, they are "moved" into the regular logic of the method implementation).

   (e) (1 point) Briefly compare the main risks and benefits of defensive programming versus design-by-contract, in light of the `adjacent` method.

   Solution: The benefit of defensive programming is that it simplifies the life of the client using a contract, since fewer assumptions must be checked. The intent is to make it harder for a program to crash, and hence more robust.

   The disadvantage of defensive programming is that it leads to redundant checking, since often the client will already be certain that a given assumption is valid. For the adjacent method, for example, often the client will already know that the other cell is not null.

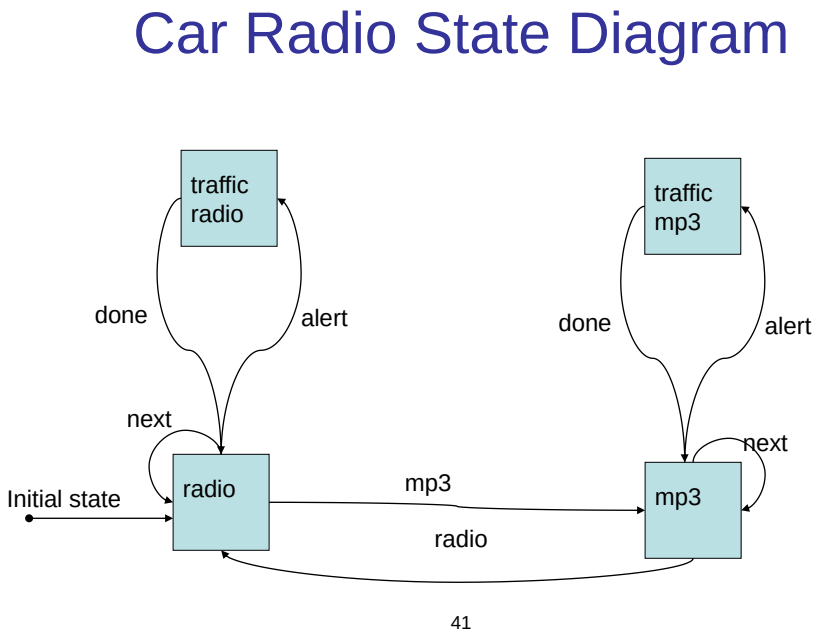
   The benefit of design by contract is that it makes the assumptions explicit, and that at run time assertion checking can be enabled (leading to a defensive execution style) or disabled (avoiding redundant checking).

5. You are responsible for testing a new radio/mp3 player to be used in cars. Its specification lists the following features:

   - When listening to the radio, pressing the "next" button searches for the next radio station. Pressing the "mp3" button causes the player to start playing the first mp3-song stored in memory.
   - When listening to an mp3-track, pressing "next" will skip to the next song. Pressing "radio" returns to playing the last radio station.
   - When either listening to the radio station or an mp3-song, a traffic alert may fire. This causes the player to switch to the radio station broadcasting traffic information. As soon as the traffic broadcast is completed the player returns to its previous mode (either mp3 or radio playing).

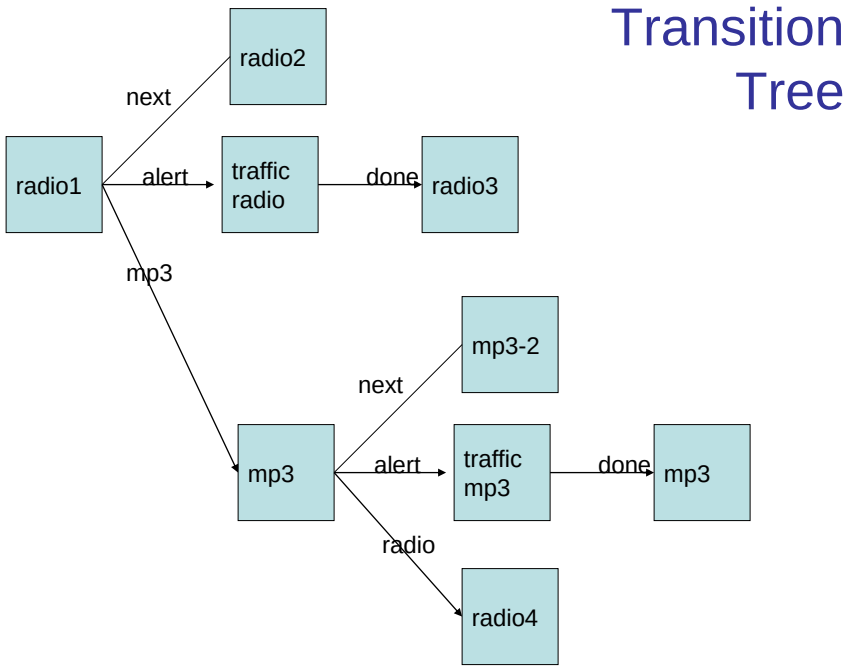
   (a) (1 point) Turn these requirements into a testable UML state diagram.

Solution:



   (b) (1 point) Create and describe a series of test case specications that achieves transition coverage.

Solution:
To do this, we turn the state machine in the following transition tree:



Next, each path from root to a leaf in this tree corresponds to a test case. In this way we obtain transition coverage (or, even stronger, all round trip path coverage). Thus we obtain 5 test cases in this way.

(c) (1 point) Identify omitted ⟨state, stimulus⟩ pairs that are ignored, and derive test cases from them (i.e., derive a sneak path test suite).

Solution:
To do so these omitted pairs, we turn the state diagram into a tabular format:

State Transition Table

States	Events				
	next	mp3	alert	radio	done
radio	radio	mp3	traffic radio		
traffic radio					radio
mp3	mp3		traffic mp3	radio	
traffic mp3					mp3

43

Next we identify the cells for which no explicit transition is given, which results in 12 cells. The response for these events should be that the state does not change and that the event is ignored.

The test specification thus consist of 12 test cases verifying that for the given (state, stimulus) pair the state does not change and that the stimulus is ignored.

6. After project completion, you (as project manager) get one of the players for free, and install it in your car. Happily listening to your favorite mp3 songs, a traffic alert message is broadcasted, after which your player continues to play the radio station instead of returning to your song.

   You suddenly realize that this is not conform the specifications, and that the player must contain a fault. You start worrying whether your test strategy was properly executed, and, if it was, whether your strategy was capable of finding this fault in the first place.

   (a) (1 point) Does the model created in Question 5 contain sufficient information to reason about the causes of this failure? If so, what are possible causes of this failure in terms of that model?

   Solution: Yes, this fault can be explained in terms of this model: This corresponds to the implementation of the “done” event in the “traffic mp3” state. This event incorrectly remains hanging in the “traffic mp3” state, instead of going back to the mp3 state.

   (b) (1 point) Which, if any, of the test cases you listed in the previous questions will necessarily reveal this fault? Why?

   Solution: The mp3–alert–done test case was designed to spot this; it should then also check that after a “done” event the new state is indeed “mp3”.

7. In addition to that, you discover that when pressing “radio” while listening to an mp3 track, the player appears to return to a random radio station instead of the one tuned in most recently – another deviation from the specifcation.

(a) (1 point) Does the model created in Question 5 contain sufficient information to reason about the causes of this failure? If so, what are possible causes of this failure in terms of that model?

Solution: The state machine does not include information on the actual songs or stations listened to, so at present this is not possible. This would require adding such information in the actions of the state machine.

(b) (1 point) Which, if any, of the test cases you listed in the previous questions will necessarily trigger this fault? Why?

Solution: None. The test case for mp3–radio–mp3 could be extended to check for such actions, though.

8. You are involved in software for electronic payments. Your function should check whether the given credit card data is valid. Data to be take into account include:

   - A 16-digit card number. If the card number starts with a 4 it is a Visa card, if it starts with 34 or 37 it is an American Express card.
   - An expiry date, which can be at most two years from now.
   - A Card Security Code (CSC). For Visa, this code consists of 3 digits, for Amercan Express it consists of 4 digits.

   The validation function should conduct basic sanity checks on the card data. These checks include applying a "mod10" algorithm to the card number (the details of the algorithm are irrelevant) which tells if the card number is OK or not, checking that the card is either Visa or American Express, checking that the expiry date has not passed yet, and that a correct number of digits is given for the Card Security Code.

   Elaborate the category partition testing method for the above setting. To that end, do the following:

   (a) (1 point) Identify the function to be tested.

   Solution: The credit card validator

   (b) (1 point) Idenitfy the parameters to be tested.

   Solution: The number itself, the experiation date, and the CSC code

   (c) (1 point) For each parameter, determine the categories (characteristics) that should be considered when testing the function

   Solution:

   - Parameter card: *length*, *mod10-compliant* and *type encoding*
   - Date: *two-years*
   - CSC code: *length*

   (d) (1 point) For each category, determine the choices (classes of representative values)

   Solution:

   - Card.length: 15, 16 or 17 digits
   - Card.mod10: valid and invalid

- Card.type: 4x, 34x, 37x, other;
- Date.two-years: beyond 2 years; before 2 years; within 2 years.
- CSC.length: 0, 1, 2, 3, 4, more than 4 digits

(e) (1 point) Identify constraints on combinations of choices

Solution: Most choices are independent from each other. This means that it is not necessary to test all combinations (in terms of the book, the constraint [single] applies, meaning that a single test exercising the given value suffices.)

The CSC code and the credit card type do depend on each other. Thus, we do need CSC codes of different lengths for Visa resp. American Express card.

Furthermore, one can expect that the mod10 algorithm relates to the actual digits in the code. Since the first one or two digits have special meaning, it makes sense to combine all four *Card.type* possibilities with each of the two *Card.mod10* possibilities.

(f) (1 point) Provide actual values for 3 test cases

Solution:

- Valid visa: 4578 4578 4578 4578; now + 1 month; 123
- Invalid visa: 4578 4578 4578 4578; now + 1 month; 1234
- Invalid date: 4578 4578 4578 4578; now + 3 years; 1234

End of exam