TECHNISCHE UNIVERSITEIT DELFT
Faculteit Elektrotechniek, Wiskunde en Informatica

TU Delft

Exam TI2210
Software Quality and Testing

17 April 2013, 14:00-17:00, Room TNW-4.25

- There are 5 questions worth of 29 points in total.
- Total number of pages: 4.
- Use of books and readers is not allowed
- You can answer in English or in Dutch
- Please list your answers in the right order with question numbers clearly indicated.
- Write clearly and avoid verbose explanations: Points may be deducted for unclear or sloppy answers.
- If you want your graded to be filed under code IN3205 please indicate so clearly on your exam.
- The tentative grading scheme is:

Question:	1	2	3	4	5	Total
Points:	9	6	3	8	3	29
Score:						

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

$$E = 1 + 9 * p/29$$

- Your final grade F is determined based on your results for the labwork L and exam E:

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

Note that you can only pass if both $E \geq 6.0$ and $L \geq 6.0$.

GOOD LUCK!

1. Current versions of Linux provide around 10,000 configuration options (specified in `kconfig` files), which are technically implemented and enforced in the code by means of preprocessor macros. To simplify our reasoning, in this exercise, we will assume that

   - each configuration option can have 3 distinct values,
   - It is possible to combine any option with any other.

   Before any such configuration can be tested, it must be compiled first. In this exercise we study how many configurations should be created.

   Answer the following questions:

   (a) (1 point) How many different kernels can be compiled (= configured)?

   (b) (1 point) Define the *pairwise* testing strategy.

   (c) (1 point) Explain how *pairwise* testing can be applied to the problem of selecting which Linux kernel configurations to compile and test.

   (d) (2 points) Assume we are willing to compile and test 10 different kernel configurations, and that you use pairwise testing to derive those.

      1. How many pairs of configuration options would you be able to test? (Hint: Remember from your Probability courses that there are $\binom{n}{t} = \frac{n!}{t!(n-t)!}$ ways to choose a combination of t variables from a set of n variables)
      2. How many 3-way combinations of configurations would you be able to test?
      3. How many full configurations would your test suite cover?

   (e) (2 points) Now assume that you have information about actual installations in the field, i.e., you know what the most common configurations are. How would this affect the 10 different kernel configurations you would test?

   (f) (2 points) Your colleague proposes to use the *category-partition strategy* instead of *pairwise testing*

      1. Briefly describe the main steps of the *category-partition strategy*.
      2. Your colleague decides to give every configuration option the `[single]` attribute. What is the smallest number of kernels to be compiled given this assessment?

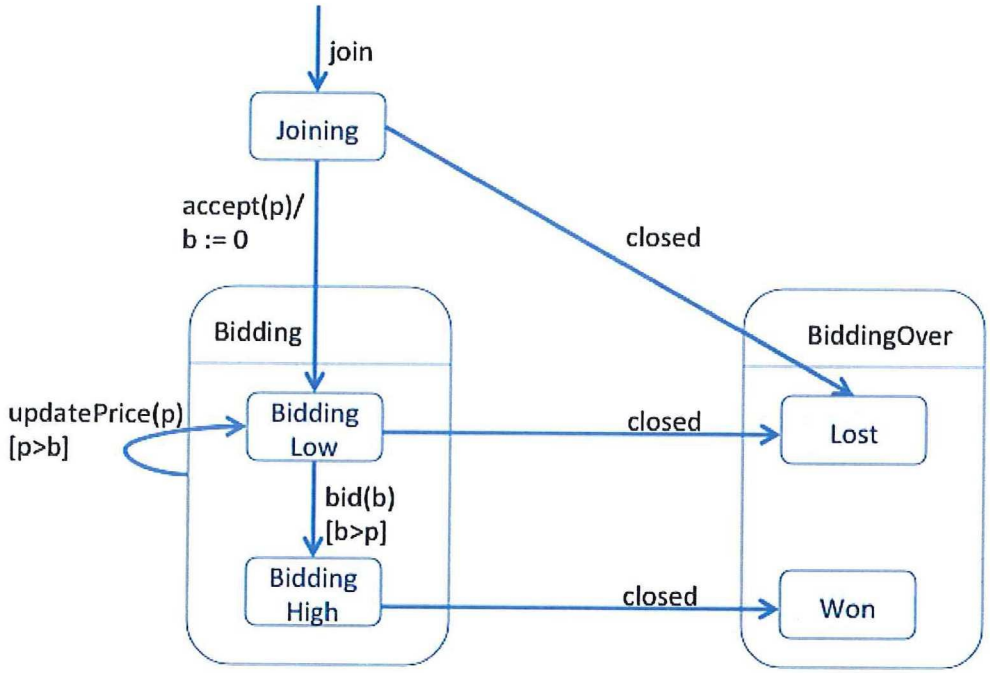

Figure 1: State machine for bidder in an auction

2. You are involved in a software system for an auction house. The design documents include the figure shown in Figure 1, displaying a UML state diagram for a bidder involved in an auction[1]

   A bidder can join an auction, after which it can start bidding. After receiving a price p, it can issue a bid b, provided $b > p$. During bidding, the bidder can be notified of price updates (caused by other bidders), leading to a new price p which is larger than the bidder's last bid b. When the auction closes the moment the bidder holds the highest bid, this bidder wins the auction of this item.

   Your job is to test the implementation of the bidder.

   (a) (1 point) Draw a flattened diagram of the state machine.

   (b) (1 point) Turn the state machine into a transition tree.

   (c) (1 point) Derive a test suite achieving *all-roundtrip path* coverage. How many test cases does it contain?

   (d) (2 points) Turn the state machine into a state transition table, in order to derive a "sneak path" test suite. How many test cases does it contain? What is a sensible default action for (event, state) pairs for which no action is defined?

   (e) (1 point) You next consider adopting the *boundary interior loop coverage* criterion for your test suite. To how many additional test cases does this lead? Explain your answer.

[1] Based on a state machine from Freeman & Ryce, *Growing Object-Oriented Software, Guided by Tests*, Addison-Wesley, 2010.

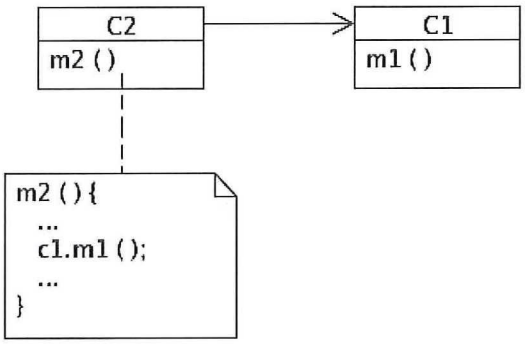

Figure 2: Class diagram for question 3

3. You are writing a test for a method m_1 in a class C_1, making use of method m_2 in another class C_2. Your test case fails since m_2 raises one assertion failure.
   (a) (1 point) Assume it is the precondition of m_2 that is failing. Which of the following statements is true? Explain your answer.
      1. This points to a problem in C_1;
      2. This points to a problem in C_2;
      3. None of the above.
   (b) (1 point) Assume it is the postcondition of m_2 that is failing. Which of the following statements is true? Explain your answer.
      1. This points to a problem in C_1;
      2. This points to a problem in C_2;
      3. None of the above.
   (c) (1 point) Assume it is the invariant of C_2 that is failing. Which of the following statements is true? Explain your answer.
      1. This points to a problem in C_1;
      2. This points to a problem in C_2;
      3. None of the above.

```
1:     public static int binarySearch(int[] a, int key) {
2:         int low = 0;
3:         int high = a.length - 1;
4:
5:         while (low <= high) {
6:             int mid = (low + high) / 2;
7:             int midVal = a[mid];
8:
9:             if (midVal < key)
10:                 low = mid + 1
11:             else if (midVal > key)
12:                 high = mid - 1;
13:             else
14:                 return mid; // key found
15:         }
16:         return -(low + 1);  // key not found.
17:     }
```

Figure 3: Binary Search in Java

4. Figure 3 contains an implementation of a binary search algorithm, written by Joshua Bloch from an early version of `java.util.Arrays`[2]. The algorithm searches the specified array of ints `a` for the specified value `key`.
   (a) (1 point) Draw a control flow graph of the method `binarySearch`.
   (b) (1 point) Define the *statement adequacy criterion*.
   (c) (1 point) Provide a minimal suite of test cases achieving statement coverage for this method.
   (d) (1 point) Define the *branch adequacy criterion*.
   (e) (1 point) Provide a minimal suite of test cases achieving branch coverage for this method.
   (f) (1 point) Define *loop boundary adequacy*
   (g) (1 point) Provide a minimal suite of test cases achieving loop boundary coverage.
   (h) (1 point) The provided method was extensively tested and widely used, yet contained an error which was uncovered for 9 years: Line 6 causes an integer overflow for very large arrays (the maximum positive integer is $2^{32}-1$). Can you think of a systematic testing strategy that could reveal this bug?

5. A mocking framework like mockito contains two groups of methods: `when`-methods and `verify`-methods.
   (a) (1 point) Explain which of these groups can be used to increase the *controllability* of a class-under-test, and why this is so.
   (b) (1 point) Explain which of these groups can be used to increase the *observability* of a class-under-test, and why this is so.
   (c) (1 point) A colleague of yours argues that mocking is like subclassing, and that therefore any mock-class should adhere to the Liskov Substitution Principle. Do you agree? Why (not)?

[2] `http://googleresearch.blogspot.nl/2006/06/extra-extra-read-all-about-it-nearly.html`

End of exam