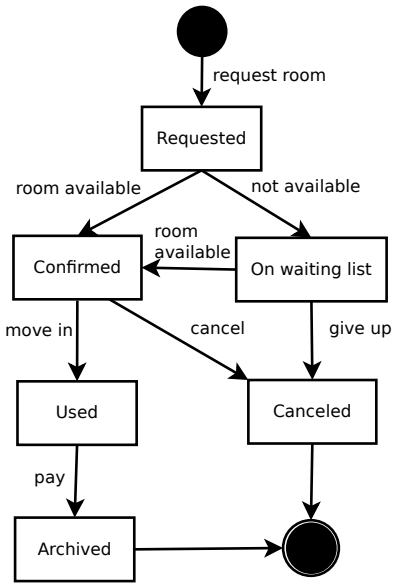
Practice Exam — with answers

Monday 19 December 2011

Choose four out of the five questions. You have two hours, if you need it. Write your answers on separate sheets of paper, with your name on each sheet and the problem number clearly labeled. You may not use books, notes, computers, or other devices. You may leave when you have completed the exam.

1. (Requirements) Examine the state/transition diagram below and answer the following questions.



(a) According to the diagram, is every confirmed hotel room eventually paid for? Explain.

(b) Is every request eventually confirmed? Explain.

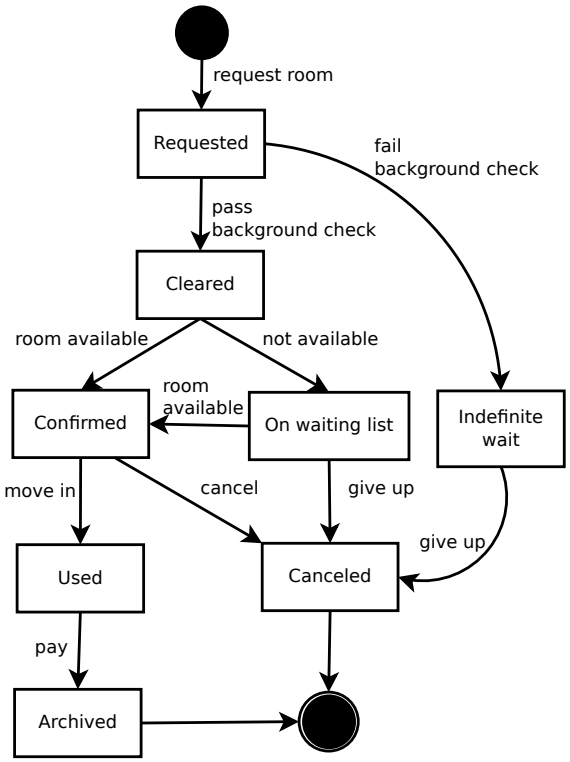
(c) Suppose this is an expensive hotel that hosts presidents and other heads of state. Every guest must pass a background check before they can move in; those that fail are kept 'waiting' indefinitely. Change the state diagram to reflect this new policy.

The key to these is to answer *directly from the diagram,* adding as little of your own interpretation as possible. Remember, in a state/transition diagram, the states are represented by boxes, and the transitions are represented by arrows. The direction of the arrows is important. The transitions are also known as actions, so they are generally described by *verbs.*

(a) The keywords in this question are 'confirmed' (which is a state), and 'paid', which is an action. So really the question asks: is there any path *from* the confirmed state to the end (the double circle) that does not pass through the *pay* action? Of course, there is such a path. The room can be canceled instead, and then no payment occurs.

(b) Again, find the keywords: 'request' is an action (leading to the 'requested' state) and 'confirmed' is a subsequent state. According to the diagram: no, not every request is eventually confirmed. The room might not be available, and then the client might 'give up' instead of waiting.

(c) A background check is an action. The resulting state could be something like 'Cleared' or 'Secure'. We must have this *before* moving in, which is another action, and we have to make sure there is no loophole that allows us to bypass the background check.



If we had placed the background check directly between 'Requested' and 'Confirmed', then the waiting list may have become a loop-hole. Here, we only allow cleared guests onto the waiting list.

From the 'indefinite wait' state, where we put clients with failed background checks, there is no path back to the 'move in' action.

2. (Design) Recall that *cohesion* is a measure of how well the various responsibilities and methods of a class fit together. A class with poor cohesion can usually be refactored into multiple classes.

   Data cohesion is one kind of cohesion, where we evaluate what data members each method accesses, to see what the methods have in common.

   The following class has two data members and five methods. Analyze the data cohesion of the class, and recommend a possible way to refactor it.

```
class BankAccount
{
    private int balance;
    private String owner;

    public void withdraw(int amount) {
        balance = balance - amount;
    }

    public void deposit(int amount) {
        withdraw( - amount );
    }

    public void getOwner() {
        return owner;
    }

    public void setOwner(String newOwner) {
        sendNotification(owner);
        sendNotification(newOwner);
        owner = newOwner;
    }

    public void sendNotification(String recipient) {
        // ...
    }
}
```

   There are clearly two different types of functionality offered by this class, and they correspond exactly to the two data members.

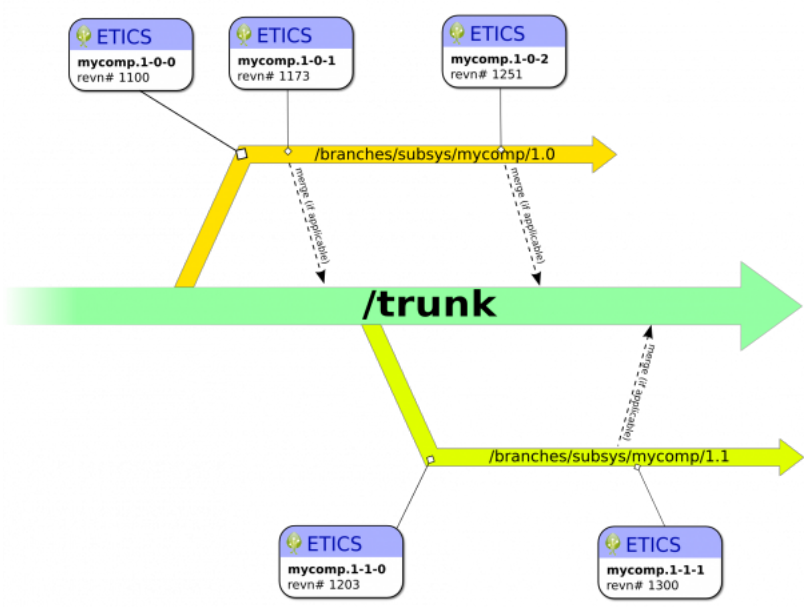
   For the 'balance' attribute, we have methods 'withdraw' and 'deposit'.

   For the 'owner' attribute, we have methods 'getOwner', 'setOwner', and 'sendNotification'. It's probably reasonable to break these into two separate classes, and then each class would have much better *cohesion*, compared to this one: their responsibilities would be more focused.

   I'd probably continue to call the 'balance' class 'BankAccount', but I'd break out a separate 'AccountOwner' class.

3. (Implementation) Briefly describe how the *branching* and *merging* features of a version control system can be helpful to software developers.

   Branching is great when we need to continue development of different versions of the system in parallel. For example, suppose we just released version 1.0 of a software product. Our development team may then get started working on brand new features for version 1.1. (New features usually go onto the main branch, which is called the 'trunk'.) Meanwhile, bug reports arrive from our customers in the field. We can't just release version 1.1 before it's ready, and they can't wait very long for those bug fixes. So, we use a *release branch* to fix the bugs in 1.0, and when they're ready, we can release version 1.0-1 from that branch.



   We also might periodically merge relevant bug fixes from the release branches back into the trunk.

   This sort of structure also happens with *experimental branches,* where a developer might want to work on a wild idea that may or may not make it into the final product. If the idea doesn't work out, then we just abandon the branch (but it is always there in the repository). If it does work out, we can eventually merge the contents of that branch into the trunk for the next release.

4. (Verification) As a test engineer for Boeing, you have been assigned to test the following pseudo-code. It has two parameters, `altitude` and `pitch`.[1]

```
01: FUNCTION autoPilot (altitude, pitch : integer)
02: BEGIN
03:     IF altitude > 10000 AND pitch < 0
04:     THEN RETURN pitch / 2
05:     ELSE IF altitude > 5000 AND pitch < 70
06:          THEN RETURN pitch * 0.3
07:          ELSE IF pitch < -25
08:               THEN RETURN (- pitch) / 2
09:               ELSE RETURN altitude + pitch
10: END
```

Your colleague suggests the following three test cases. Each case contains proposed inputs (values for `altitude` and `pitch`) and the expected output (return value).

test case #	altitude	pitch	return
1	10384	70	10454
2	986	–32	16
3	768	0	768

Do these three test cases produce good *coverage?* If not, suggest one or two additional cases that will make your testing more comprehensive. (Be specific: give proposed inputs and expected output.)

(I added line numbers to the program, and identifying numbers to the test cases, so we can talk more explicitly about coverage.) The key here is to trace the code for each test case. Line 3 begins with the expression about altitude and pitch. For test case 1, is it true or false? Altitude is greater than 10000, but pitch is > 0, so on the whole this conditions is false. That means we jump to the 'ELSE' on line 5. Is that condition true or false? Altitude is over 5000, but pitch is not < 70, so false. Jump to 'ELSE' on line 7. Is pitch less than -25? No. So we're on to the 'ELSE' at line 9 and return altitude + pitch == 10454, as expected.

That first test *visited* (covered) lines 3, 5, 7, and 9. It *skipped* 4, 6, 8.

Continue this reasoning with the other test cases. Test case 2 covers lines 3 (false), 5 (false), 7 (true), 8.

Test case 3 covers lines 3 (false), 5 (false), 7 (false), 9.

Did we hit every significant line of code? (Lines 1, 2, and 10 are not really relevant.) No! We never executed line 4 or line 6! So we have to find at least two more test cases:

```
Test 4: altitude=10002, pitch=-4, return -2 (hits line 4)
Test 5: altitude=6000, pitch=50, return 15 (hits line 6)
```

[1] It doesn't matter to this problem, but *altitude* is the distance of a plane off of the ground, and *pitch* is the angle of the nose with respect to the ground.

5. (Maintenance) The first law of software evolution[2] states that "a program that is used undergoes continual change or becomes progressively less useful." Do you agree with this statement? It's not obviously true – if the program was useful at one time, why would it not continue to be useful into the future? Explain, and give concrete examples of systems for which you think the law is and is not true.

   This one is pretty subjective, so as long as you answer thoughtfully about some of the issues below, you'd get full credit.

   The main reason that unmaintained software becomes progressively less useful is because the environment around us is always changing. Eventually these changes impact the requirements for the software.

   We can place these 'environmental' changes in a few different categories. In many fields, there are laws and regulations that evolve every few years, and so unmaintained software in those fields will eventually find itself out of date with respect to our expectations.

   The hardware on which the system runs, and other software it depends on (like the operating system and various libraries) are also changing. Your old machine won't last forever, but if the software is unmaintained, will it continue to run on a new machine?

   Our expectations change over time too. Once it was okay for certain kinds of software to be unaware of the Internet, but now the net is so pervasive that those programs have limited appeal.

   All that said, there may be examples where software remains useful for a long time, even as the world around it changes. If you need a system to do simple numerical calculations, well, the mathematics you're using probably hasn't changed in hundreds of years!

[2] formulated in a 1974 article by Lehman and Belady