CSE 331 Midterm Exam Sample Solution 2/13/12

Question 1. (14 points) (assertions) Using backwards reasoning, find the weakest precondition for each sequence of statements and postcondition below. Insert appropriate assertions in each blank line. You should simplify your answers if possible.

(a)

```
{ (x+1)*x + y - x > 0 } simplifies to { x^2 + y > 0 }
a = x + 1;
{ a*x + y - x > 0 }
b = y - x;
{ a*x + b > 0 }
```

(b)

```
{ (y+1)+3 > 0 & y+1 < 0 } simplifies to { -4 < y & y < -1 }
y = y + 1;
{ y+3 > 0 & y < 0 }
x = y + 3;
{ x > 0 & y < 0 }
```

Question 2. (25 points) Loop development. The factorial function, as you probably remember, is defined as *n*! = 1 * 2 * 3 * … * *n*, i.e., the product of the numbers 1 through *n*. For this problem, complete the following **non-recursive** method to compute and return *n*! and prove that it computes and returns the correct answer. The method heading is provided for you as is the declaration of variable `ans` and the `return` statement at the end. You should declare additional variables as you need them. Your answer should supply the method code and include assertions, preconditions, postconditions, and invariants as needed to prove it is correct.

```
// return the value n! = 1 * 2 * ... * n
// pre: n > 0
int fact(int n) {
   int ans;
   ans = 1;
   int k = 1;
   { ans = k! }
   while (k != n) {
      { inv: ans = k! }
      k = k + 1;
      { ans = (k-1)! }
      ans = ans*k;
      { ans = k! }
   }
   { ans = k! & k = n }
   { ans == n! (where n is the original argument value) }
   return ans;
}
```

[There are, of course, many variations on the possible invariants and details of the proof. As long as those were done correctly they didn't have to match this solution exactly.]

Question 3. (12 points) (Specifications) Consider the following specifications for a method that has one integer argument:

(a) Returns an integer ≥ the argument
(b) Returns a non-negative integer ≥ the argument
(c) Returns argument – 1
(d) Returns argument^2 (i.e., the square of the argument)
(e) Returns a non-negative number

Consider these implementations, where `arg` is the function argument value:

```
(i)     return arg + 5;
(ii)    return arg * arg;
(iii)   return arg % 10;
(iv)    return Math.abs(arg);
(v)     return Integer.MAX_VALUE;
```

Place a check mark in each box for which the implementation satisfies the specification. If the implementation does not satisfy the specification, leave the box blank.

Implementation	**Specification**				
	(a)	(b)	(c)	(d)	(e)
(i)	X				
(ii)	X	X		X	X
(iii)					X
(iv)	X	X			X
(v)	X	X			X

[The specifications and implementations were supposed to be simple statements about integers. However some people pointed out corner cases with integer overflow or with `Math.abs` when applied to the negative Java `int` with largest magnitude. We gave credit for answers that took those cases into account, but did not require it.]

Question 4. (12 points) A bit of code reading. Consider the following ADT, which we wish to use to represent a line of people waiting to buy tickets at a movie theater.

```
public class WaitLine {

   // instance variable
   private List<String> people;

   // AF: people represents a list of persons in a line (say
   // at a movie theater).  They are enqueued at the end of
   // the line when they arrive and dequeued in that order.

   // RI: entries in people are not null and, for persons A
   // and B, A appears in people before B if enqueue(A) was
   // called before enqueue(B).

   // constructor
   public WaitLine() {
      people = new LinkedList<String>();
      checkRep();
   }

   // mutators

   public void enqueue(String p) {
      people.add(p);
      checkRep();
   }

   public String dequeue() {
      if (people.size() == 0) return null;
      String p = people.remove(0);
      checkRep();
      return p;
   }

   // observers

   public int size() { return people.size(); }

   public List<String> getEveryone() { return people; }

   // internal method
   private void checkRep() {
      for (String p: people) assert p!= null;
   }

}
```

Answer the questions about this class on the next page. You may remove this page for reference if you wish.

Question 4. (cont.) Answer the following questions about the `WaitLine` class on the previous page. **Be brief:** you shouldn't need more than a short sentence or two for each answer, but you do need to justify your conclusions.

(a) The `dequeue` method returns `null` if the queue is empty. Is this a reasonable way for the method to behave if the queue is empty? Does it create problems or ambiguities for clients using this class?

Yes, this is one reasonable way to handle the empty queue. There is no ambiguity because the queue cannot contain `null`, so the `null` returned for an empty queue cannot be confused with a regular queue item value.

(b) Are there any potential representation exposure problems with this class? If so, what are they, and how could they be fixed, preferably without major changes to the class?

Yes. Method `getEveryone` returns a reference to the `people` list, which means that client code can access the representation and modify it. The fix is either to return a copy of the list, or return an `unmodifiableList` wrapper object that contains the original list.

[Any solution that said "return an immutable copy" or something to that effect received credit.]

(c) Is the definition and use of `checkRep` reasonable, given the operations in the class?

Not really. Most of the operations in the ADT run in constant time (*O*(1)). The `checkRep` method takes *O*(*n*) time, which makes almost all operations significantly more expensive. This `checkRep` would, however, be useful for debugging.

[Of course if we don't execute `checkRep` after every operation, the `enqueue` operation had better ensure that `null` can't be added to the queue, otherwise the RI might not hold.]

(d) Our new summer intern wants to add a `sellTicket` method to this class to sell a ticket to the first person in line and remove them from the queue. Is this a good design decision? Why or why not?

No. It reduces the cohesion of the `WaitLine` class, since it is an operation not related to the basic function of maintaining an ordered queue. It also likely increases coupling between `WaitLine` and other classes that handle operations related to ticket sales.

Question 5. (12 points) Testing. In homework 4 one of the questions involved a queue that was implemented using a finite array as a "circular list". New items were added at the end of the array, but after filling the last slot in the array, we wrapped around and stored the next queue element at the beginning – which worked provided that the oldest element(s) had been previously removed to make room.

Solutions were evaluated both on the proposed tests and on whether they were correctly categorized as black box or white/glass box. Tests needed to describe both a fairly specific test setup and the expected results to get full credit. Something like "add things to the queue" wasn't sufficient by itself.

(a) Describe two good **black box** tests for this queue implementation. The two tests should be from different revealing subdomains – i.e., they should not detect exactly the same set of errors. You do not need to give JUnit code – just describe the tests.

Some possible black-box tests:

- **Create a new queue and verify it is empty.**
- **Add two or more items to a queue, remove one item, and verify it was the first item added.**
- **Attempt to dequeue an item from an empty queue and verify proper behavior – either an exception is thrown or an appropriate value like null is returned.**
- **Add several items to the queue, remove all of them, verify they are removed in the expected order and that the queue is empty at the end.**

(b) Describe two good **white box** (or glass box) tests for this queue implementation. As with the black box tests, the two tests should be from different revealing subdomains. Again, no JUnit code required.

Some possible tests:

- **Create a new queue and check the initial state is as expected (`size==0`, etc.).**
- **Add enough items to the queue to fill the array, delete one item, add an additional item, and verify the pointer has wrapped around and the items are in the expected positions.**
- **Fill the queue then add an additional item and verify the array resizes or an exception is thrown, depending on the queue specification.**
- **Add enough items to the queue to fill it, remove an item, add another item; dequeue all items and verify they are returned in the expected order.**

A few short answer questions…

Question 6. (6 points) Given the representation in an ADT and the representation invariant (RI) that it satisfies, an abstraction function (AF) describes the meaning of this representation as an abstract value. Is the AF also a function in the other direction, i.e., do the abstract value and the AF determine a unique representation? If yes, give a brief justification for your answer; if no, give a counterexample explaining why not.

No. Given a particular representation and AF, there may be multiple representations that map to the same abstract value. One example: the unordered set {A,B} could be stored in an array as either [A,B] or [B,A]. Another example: the circular list representation of a queue from the previous problem. The same abstract queue X,Y,Z could be stored in multiple places in the array.

[Many answers explained how a single abstract value can have multiple concrete representations, such as using polar or rectangular coordinates to represent complex numbers. Those examples, however, use different AFs to map each different representation to abstract values. That is not the same issue as whether a single AF uniquely determines representations for particular abstract values.]

Question 7. (6 points) You are implementing your graph ADT using appropriate specifications, documentation, JUnit tests, and version control. You find a bug. What strategy should you use to deal with the problem and ensure it is unlikely to occur again? (i.e., what steps in the development process should come next after discovering the bug?)

- **Create a test to demonstrate the bug and verify that the test fails. Add the test to the repository.**
- **Fix the bug.**
- **Run all tests to verify that they pass and that the fix did not introduce new bugs.**
- **Commit the changes (test(s) plus fix(es)) to the repository.**

Question 8. (6 points) Suppose we create an ADT to implement an immutable data type. The ADT contains only creator, observer, and producer methods – no mutators. True or false: no method in the ADT can modify the ADTs instance variables once an instance has been created. Give a brief justification for your answer.

False. The abstract value cannot change, but the internal representation can be modified. An example would be a self-reorganizing set that moves frequently accessed items to the front of a list to increase the probability that future requests will take less time. This sort of change is sometimes called a "benevolent side effect", often done to improve performance.

Question 9. (6 points) Java contains both checked and unchecked exceptions. A method that might cause a checked exception either has to catch the exception or include a `throws` clause in its heading to indicate that it might generate that exception. Why aren't all exceptions treated this way? In particular, why are there unchecked exceptions that don't have to be handled or included in the method's heading? After all, they are as much a part of the method's possible behavior as the checked ones.

Unchecked exceptions generally indicate programming errors. They could appear anywhere in the code and documenting them would pointlessly clutter up interfaces.

Checked exceptions represent unusual or unexpected events such as being unable to open a file or a dropped network connection. While they are not necessarily expected to occur, they are things a client might want to handle. It is appropriate to document them as part of a method's interface if they might be thrown by the method, or require that the method catch them if they should not be visible outside.

Question 10. (1 point – all honest answers receive the point) What is the one question (if any) that you really thought would be on this test that we forgot to include??