Name:____

CSE331 Fall 2014, Midterm Examination
October 31, 2014

Please do not turn the page until 2:30.

Rules:

- The exam is closed-book, closed-note, etc.
- **Please stop promptly at 3:20.**
- There are **110 points (not 100 points)**, distributed **unevenly** among **8** questions (many with multiple parts):

Question	Max	Earned
1	16	
2	16	
3	18	
4	10	
5	12	
6	14	
7	10	
8	14	

Advice:

- Read questions carefully. Understand a question before you start writing.
- **Write down thoughts and intermediate steps so you can get partial credit. But clearly indicate what is your final answer.**
- The questions are not necessarily in order of difficulty. **Skip around.** Make sure you get to all the questions.
- If you have questions, ask.
- Relax. You are here to learn.

Name:____

1. (**16** points) Here is a correct Java method:

```
@requires: arr != null and arr.length > 0
@returns: number of elements in arr that are strictly greater than all
          elements earlier in the arr
int f(int[] arr) {
   int max = arr[0];
   int count = 1;
   int i = 1;
   while(i != arr.length) {
       if(arr[i] > max) {
          max = arr[i];
          count = count + 1;
       }
       i = i + 1;
   }
   return count;
}
```

To prove this method correct would require a loop invariant for the loop in the code. For each suggested loop invariant below, indicate which of the following is true (no explanation required):

A. The invariant is correct and is strong enough to prove the method correct.

B. The invariant is correct but is not strong enough to prove the method correct.

C. The invariant is false because it may not hold initially.

D. The invariant is false because it holds initially but may not hold after the loop body.

(a) count holds the number of elements in arr[0]..arr[i] inclusive that are greater than all earlier elements in the array

(b) count holds the number of elements in arr[0]..arr[i-1] inclusive that are greater than all earlier elements in the array

(c) count holds the number of elements in arr[0]..arr[i] inclusive that are greater than all earlier elements in the array
and max holds the largest value in arr[0]..arr[i] inclusive

(d) count holds the number of elements in arr[0]..arr[i-1] inclusive that are greater than all earlier elements in the array
and max holds the largest value in arr[0]..arr[i-1] inclusive

(e) count holds the number of elements in arr[0]..arr[i-1] inclusive that are greater than all earlier elements in the array
and max holds the largest value in arr[0]..arr[i-1] inclusive
and count >= 1

(f) count < max

(g) count = 1 and max >= arr[0]

(h) true

Name:____

2. (**16** points) Fill in the blanks below so that the program is correct *and* the assertions you write down are true and sufficient to prove the program is correct. Put assertions in the blanks with "{" and "}" and code in the other blanks. For code, use Java syntax. For assertions, use syntax similar to the provided assertions. Assume `arr` is an array of ints and `x`, `y`, and `i` are ints. Notice the initial pre-condition, final post-condition, and a little of the code are provided to you. You need to understand the post-condition to determine what the program should do.

```
 {arr != null and arr.length > 1}
 arr[0] = x;
 arr[1] = y;

 {                                                                  }
--------------------------------------------------------------------


--------------------------------------------------------------------

 {                                                                  }
--------------------------------------------------------------------


 {inv:                                                                }
--------------------------------------------------------------------

 while (i != arr.length) {


    --------------------------------------------------------------------

    {                                                                  }
    --------------------------------------------------------------------


    --------------------------------------------------------------------

    {                                                                  }
    --------------------------------------------------------------------
 }
 { arr[0] = x and arr[1] = y and
   for all j from 2..(arr.length-1) inclusive, arr[j] = arr[j-2] + arr[j-1] }
```

Name:____

Background: Problems 3–5 involve an ADT for the state of a checkerboard during a game of checkers. You do not need to know how to play checkers, and even if you do, do not use any information beyond what is listed here:

- There are two sides, "red" and "black."
- Each side has a total number of pieces between 0 and 12 inclusive. If one side has 0 pieces, the game is over. Both sides cannot have 0 pieces.
- In addition to color, each piece is either "regular" or a "king."
- There are 32 positions on a checkerboard. *We assume these positions are numbered 0,1,...,31 and this numbering is part of the (public) specification, but what each position number means will not matter on this exam.*
- Each position holds 0 or 1 piece.

3. (**18** points) In this problem, we consider an implementation of a checkerboard where the concrete representation is an array of length 32 and the abstract values are the state of a checkers game as described above.

```
class CheckerBoardProblem3 {
    private int[] board = new int[32];

    ... many methods not shown ...
}
```

In the concrete implementation, the value in `board[i]` indicates what piece, if any, is at board-position `i` as follows:

- 0 means no piece
- 1 means a regular red piece
- 2 means a king red piece
- -1 means a regular black piece
- -2 means a king black piece

(a) Write a `checkRep` method for class `CheckerBoardProblem3`.

(b) Write an `equals` method for class `CheckerBoardProblem3`.

(c) Explain in approximately 1-2 English sentences what is wrong with this method for `CheckerBoardProblem3` and how to fix it:

```
@returns the board positions of all red kings
public List<Integer> redKingPositions() {
   List<Integer> ans = new ArrayList<Integer>();
   for(int i=0; i<board.length; i++) {
      if(board[i] == 2)
         ans.add(board[i]);
   }
   return ans;
 }
```

The next page has room for your answers.

Name:____

Put your answers to problem 3 here.

Name:____

4. (**10** points) In this problem, we consider a different implementation of the same abstraction of a checkerboard. In this approach, the concrete implementation has a list of the positions for each kind of piece:

```
class CheckerBoardProblem4 {
   private List<Integer> regularBlackPositions;
   private List<Integer> kingBlackPositions;
   private List<Integer> regularRedPositions;
   private List<Integer> kingRedPositions;

     ... many methods not shown ...
}
```

(a) In English, describe two things a `checkRep` for `CheckerBoardProblem4` should check that your implementation of `checkRep` in Problem 3 did not have to check.

(b) Explain in approximately 1-2 English sentences what is wrong with this method for `CheckerBoardProblem4` and how to fix it:

```
@returns the board positions of all red kings
public List<Integer> redKingPositions() {
   return kingRedPositions;
}
```

Name:____

5. (**12** points) This problem uses both `CheckerBoardProblem3` and `CheckerBoardProblem4`.

   (a) Suppose `CheckerBoardProblem3` and `CheckerBoardProblem4` both implement the same checkerboard abstraction correctly. Would an instance of `CheckerBoardProblem3` and an instance of `CheckerBoardProblem4` with the same abstract state be **reference-equivalent**? Answer "always", "sometimes," or "never." No explanation required.

   (b) Same question as (a) but replace reference-equivalent with **behavior-equivalent**.

   (c) Can two instances of `CheckerBoardProblem3` with different concrete values represent the same abstract value? Answer "yes" or "no" *and explain your answer in 1-2 sentences.*

   (d) Can two instances of `CheckerBoardProblem4` with different concrete values represent the same abstract value? Answer "yes" or "no" *and explain your answer in 1-2 sentences.*

Name:____

6. (**14** points) Consider these five possible specifications for a method that takes one parameter, an `int x`:

   A. @returns some number between $x - 10$ and $x + 10$

   B. @returns some number between $x - 5$ and $x + 5$

   C. @requires $x > 0$
   @returns some number between $x - 5$ and $x + 5$

   D. @requires $x > 0$ or $x < -5$
   @returns some number between $x - 5$ and $x + 5$

   E. @requires $x > 0$
   @throws IllegalArgument if $x > 100$
   @returns some number between $x - 10$ and $x + 10$

   (a) List all specifications above that are stronger than A.

   (b) List all specifications above that are stronger than B.

   (c) List all specifications above that are stronger than C.

   (d) List all specifications above that are stronger than D.

   (e) List all specifications above that are stronger than E.

   (f) Yes or no: Is it possible for a single method to satisfy A, B, C, and D?

   (g) Yes or no: Is it possible for a single method to satisfy C, D and E?

Name:____

7. (**10** points) Two questions on testing

(a) Can you use a black-box testing methodology to try to find representation-exposure bugs in an ADT implementation? If so, explain how. If not, explain why not. Aim for 2–3 English sentences.

(b) Consider this method:

```
@returns the least of the 3 arguments
int min3(int x, int y, int z) {
   int a;
   if(x < y) {
     a = x;
   } else  {
     a = y;
   }
   if(z < a)  {
     return z;
   } else {
     return x;
  }
}
```

i. Give a test suite for this method with full branch coverage and where all tests pass.

ii. Give a test that does not pass.

Name:____

8. (**14** points) Short answer (only (e) requires more than a word or letter)

   (a) Can defining this method in a Java class violate any contracts specified by `Object`?

   ```
   int hashCode() {
      return 42;
   }
   ```

   (b) In Java, can a method overriding a method defined in a superclass throw a checked exception that is not part of the method signature in the superclass?

   (c) In Java, can a method overriding a method defined in a superclass throw an unchecked exception that is not part of the method signature in the superclass?

   (d) When is it decided whether Java assertions are executed or ignored:

      A. When code containing assertions is compiled
      B. When the Java program is started
      C. When an assertion is encountered by checking whether there is a DEBUG flag that is true
      D. It is not an option: Java assertions are always executed

   (e) Rewrite this code to be shorter but do the same thing:

   ```
   try {
      f();
   } catch(Exception e) {
      g();
      throw e;
   } catch(Error e) {
      g()
      throw e;
   }
   g();
   return 0;
   ```

   (f) Why should you avoid using strings to store data that is not naturally a string?

      A. So the type system can catch more bugs
      B. Performance
      C. Both (A) and (B)
      D. Neither (A) nor (B)

   (g) True or false: Assuming unlimited time and developer resources, a stronger specification is always better than a weaker one.