STUDENT NAME: ____

CSE331 Autumn 2011 Final Examination

December 14, 2011

- This is intended to be a second midterm rather than a comprehensive final. (Connections to material on the first midterm are reasonable, of course.)
- You may take the entire 110 minutes allocated for the final, but it is intended to be finished in 50-60 minutes.
- Open note, open book, closed neighbor, closed anything electronic (computers, web-enabled phones, etc.)
- An easier-to-read answer makes for a happier-to-give-partial-credit grader

Don't turn the page until the proctor gives the go ahead!

STUDENT NAME: ______________________________________

Part I: True/False with brief explanation (30 points total)
[Each question: 2 points T/F, 1 point explanation]

1) All other things being equals, a client generally prefers to call a method with a weaker precondition than a method with a stronger precondition.

2) Consider a class **A** with one constructor **C**, three methods **M1**, **M2**, and **M3**, and representation invariant **RI**. Proving the following properties would ensure that the **RI** always holds:

- `true{C}RI`
- `RI{M1}RI`
- `RI{M2}RI`
- `RI{M3}RI`

STUDENT NAME: ____

3) The midterm included a question based on the following *safe* – `hashCode` is *safe* if equal objects always have equal `hashCode` values – definition of `hashCode` for the `Duration` class:

```
public int hashCode() {
   return sec*min;
}
```

T/F: Proving the Hoare triple

`true{hashCode.return = sec*min}0≤hashCode.return≤3600`

guarantees that this `hashCode` function is indeed *safe*. (`hashCode.return` simply refers to the return value of the function.)

4) The catch blocks that will be executed when any exception is raised at run-time can be determined by inspecting Java source code without running the program.

5) If a program **P** satisfies a specification **S**, then any test run on **P** will succeed.

6) Java interfaces allow substitutability at run-time, while Java generics allow substitutability at compile-time.

7) A representation invariant for an immutable object must hold at all times during program execution except during execution of the constructor of that object.

8) Consider a Java program that uses generic `Collection`s. It is possible to automatically convert this program to an equivalent one that does not use generic `Collection`s.

9) An advantage of explicitly considering an abstraction function and a representation invariant is that this simplifies debugging.

10) When a regression test fails, the programmer must change the program being tested.

STUDENT NAME: ______________________________

Part II (20 points total)

Part II.A (10 points) The Hoare rule describing a standard **`while`** loop is:

`P{while B do S}Q`

Proving a loop with this rule requires finding a loop invariant I and proving three sub-parts:

- **P⇒I**
- **I∧B{S}I**
- **(I∧¬B)⇒Q**

Consider the language construct **`repeat-until`**.

`P{repeat S until B}Q`

This defines a loop that executes **S** and then checks condition **B**: if **B** is **`false`**, the loop is repeated; if **B** is **`true`**, the loop terminates.

Describe how to prove a **`repeat-until`** loop (using the description of the **`while`** loop above as a general model).

STUDENT NAME: ____

Part II.B (10 points)

Consider the following program that accepts two integers **A** and **B** and computes B^A.

```
A > 0 ∧ B ≥ 0                   --This is the precondition
{
  m = A;
  n = B;
  p = 1;
  repeat
    p = p*n;
    m = m-1
  until m = 0
}
p == B^A                        --This is the postcondition
```

Using the proof structure you defined in part A, prove that this loop is correct. You need not consider termination at all – that is, simply focus on weak correctness. (If are really uncertain about your answer to part A, you can – for a maximum of ½ credit on part B – using the same pre- and post-conditions, convert the program to use a `while` loop instead of `repeat-until` and then prove it.)

Part III (15 points total)

State whether using design patterns, generics, or inheritance would be the best approach for each of the following problems. Justify your answer in at most two sentences.

1. (8 points) A program includes a Java class that defines a static method implementing a clever sorting algorithm over integer arrays that works especially well when sorting 1,000,000 or more integers. Your boss asks you to modify the program to (a) use this method for 1,000,000+ long integer arrays, (b) use quicksort for integer arrays from size 10 to 1,000,000, and (c) use insertion sort for smaller arrays of integers. Would this change be constructed using design pattern(s), generic(s), or inheritance? Why?

2. (7 points) A program is to accept a person's age and determine what actions the person is allowed to take based on the age, with more actions allowed as a person gets older. For example, in the US a person is allowed to purchase cigarettes at 18, to purchase alcohol at 21, to become a congressman at 25, to become a senator at 30, to become president at 35, and to join AARP (American Association of Retired People) at 50. Would the core of this program be constructed using design pattern(s), generic(s), or inheritance? Why?

STUDENT NAME: ____

Part IV (20 points total)

Consider the following [piece]() of a Java program, which uses two buttons and sets up one listener (i.e., event handler) for the first button and two listeners for the second button.

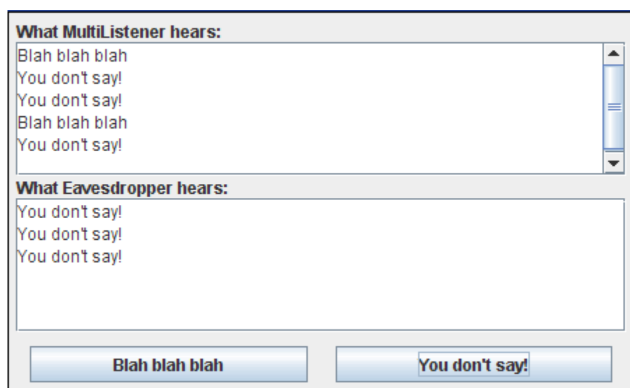
```
public class MultiListener ... implements ActionListener {
    ...
    //where initialization occurs:
        button1.addActionListener(this);
        button2.addActionListener(this);

        button2.addActionListener(new Eavesdropper(bottomTextArea));
    }

    public void actionPerformed(ActionEvent e) {
        topTextArea.append(e.getActionCommand() + newline);
    }
}

class Eavesdropper implements ActionListener {
    ...
    public void actionPerformed(ActionEvent e) {
        myTextArea.append(e.getActionCommand() + newline);
    }
}
```

The following figure shows the basics of executing the program: clicking the first button or the second button appends the button's title to the top text box, while clicking the second button appends the button's title in the bottom text box:



STUDENT NAME: ___________________________________

1. (2 points) Does this represent conventional flow-of-control or inversion-of-control?
2. (3 points) What dependences exist between the `MultiListener` and `Eavesdropper` classes in this code snippet? (As an example, if we wanted to know about the relationship between these two classes *and* `ActionListener`, we would say that both of them implement the `ActionListener` interface. But don't include `ActionListener` in your answer – only `MultiListener` and `Eavesdropper`.)
3. (5 points) Consider an added `int` count (say, declared globally), with count initialized to zero. Also, code to increment that count field is added to the bodies of the two `actionPerformed` methods (one in `MultiListener`, the other in `Easedropper`). In one or two sentences, explain why count does not represent the total number of times either button is pressed. (Only consider the value of count while the GUI is idle – that is, not during the execution of any of this code.)
4. (10 points) Assume you did want to count all button presses. Sketch a way to modify the original program cleanly and effectively to achieve this. Make sure that the solution works even if additional buttons and listeners are added; for example, simply incrementing `count` when `button1` is pushed is not clean or effective if another button is added that only appends to the bottom text box.)