CSE 331 Midterm Exam 2/20/13

Name _____

There are 8 questions worth a total of 100 points. Please budget your time so you get to all of the questions. Keep your answers brief and to the point.

The exam is closed book, closed notes, closed electronics, closed telepathy, etc.

Many of the questions may have short solutions, even if the question is somewhat long. Don't be alarmed.

If you don't remember the exact syntax of some command or the format of a command's output, make the best attempt you can. We will make allowances when grading.

Relax, you are here to learn.

Please wait to turn the page until everyone is told to begin.

Score _____ / 100

1. _____ / 16
2. _____ / 12
3. _____ / 10
4. _____ / 12
5. _____ / 20
6. _____ / 18
7. _____ / 6
8. _____ / 6

Question 1. (16 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)

```
{ _____ }
z = x + y;

{ _____ }
y = z - 3;
{ x > y }
```

(b)

```
{ _____ }
p = a + b;

{ _____ }
q = a - b;
{ p + q = 42 }
```

Question 2. (12 points) Specification madness. Suppose we are implementing a BankAccount class that has the following heading and instance variable:

```
public class BankAccount {
  int balance;  // current account balance
```

Here are three possible specifications for a method `withdraw(amount)` to withdraw funds from the account. To save space, we have omitted the `@modifies this` clause, which would be part of all of the specifications.

A @effects decreases balance by amount

B @requires amount <= balance and amount >= 0
@effects decreases balance by amount

C @throws InsufficientFundsException if balance < amount
@effects decreases balance by amount

Now, here are four possible implementations of method `withdraw`:

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

II
```
void withdraw(int amount) {
     if (balance >= amount) balance -= amount;
}
```

III
```
void withdraw(int amount) {
    if (amount < 0) throw new IllegalArgumentException();
    balance -= amount;
}
```

IV
```
void withdraw(int amount) throws InsufficientFundsException {
    if (balance < amount) throw new InsufficientFundsException();
    balance -= amount;
}
```

(continued on next page – you may remove this page for reference.)

Question 2. (cont.) In the following table, put an X in each square where the implementation whose number is given on the left properly implements the specification whose letter is given at the top. If a given implementation does not meet a specification, or if a specification is improperly formed, inconsistent, or otherwise defective, leave the square blank.

	Spec A	Spec B	Spec C
Impl I			
Impl II			
Impl III			
Impl IV			

The next several questions concern the following class that represents immutable, integer-valued polynomials, similar to the one sketched in examples in lecture.

The basic class definition and rep are as follows:

```
/**
 * An IntPoly is an immutable, integer-valued polynomial
 * with integer coefficients.  A typical IntPoly value
 * is a_0 + a_1*x + a_2*x^2 + ... + a_n*x_n.  An IntPoly
 * with degree n has coefficent a_n != 0, except that the
 * zero polynomial is represented as a polynomial of
 * degree 0 and a_0 = 0 in that case.
 */
public class IntPoly {
  // rep
  int a[];    // Coefficients

  // AF(this) = a has n+1 entries, and for each entry,
  //            a[i] = coefficient a_i of the polynomial.
```

Two of the methods of the class are the following:

```
  /**
   * Return the coefficients of this IntPoly
   */
  public int[] getCoeffs() {
    return a;
  }

  /**
   * Return a new IntPoly that is the sum of this and other
   */
  public IntPoly add(IntPoly other) {
    // implementation omitted
  }
```

See the questions concerning this class on the following pages. You may remove this page for reference if you wish.

Question 3. (10 points) The above `add` method does not have a careful specification. Below, give a complete, appropriate specification for method `add`. If some part of the specification would be empty or "none", say so explicitly.

```
/**
 * Return a new IntPoly that is the sum of this and other
 *

 * @requires


 * @modifies


 * @effects


 * @return


 * @throws

 */
public IntPoly add(IntPoly other) {
  ...
}
```

Question 4. (12 points) (rep exposure) The observer `getCoeffs` method shown above returns to the client an array with the coefficients of this `IntPoly`.

```
/**
 * Return the coefficients of this IntPoly
 */
public int[] getCoeffs() {
  return a;
}
```

(a) One of your colleagues is worried that this creates a potential representation exposure problem. Another colleague says there's no problem since an `IntPoly` is immutable. Is there a problem? Give a brief justification for your answer.

(b) If there is a representation exposure problem, give a new or repaired implementation of `getCoeffs` that fixes the problem but still returns the coefficients of the `IntPoly` to the client. If it saves time you can give a *precise* description of the changes needed instead of writing the detailed Java code.

Question 5. (20 points) Loop development. We would like to add a method to this class that evaluates the `IntPoly` at a particular value x. In other words, given a value x, the method `valueAt(x)` should return $a_0 + a_1x + a_2x^2 + ... + a_nx^n$, where a_0 through a_n are the coefficients of this `IntPoly`.

For this problem, develop an implementation of this method and prove that your implementation is correct. For full credit, you must invent an appropriate loop invariant and show it is established by any initialization code, is maintained as the loop executes, and that after termination of the loop plus any additional code, the proper result (value) is returned to the caller. Your proof does not need to be completely formal, but needs to be sufficiently careful to convince the reader that the code and proof are correct (i.e., you can skip over tedious, obvious logic steps and simplifications – provided they really are obvious).

You may not call any other methods. This method should be self-contained.

Write
Your
Answer
On the
Next page …

(Use the rest of this page for scratch work, but please put your answer **on the next page**.)

Question 5. (cont.) Write your code and correctness proof below.

```
/** Return the value of this IntPoly at point x */
public int valueAt(int x) {









}
```

Question 6. (18 points) Suppose we are defining a class to represent items stocked by an online grocery store. Here is the start of the class definition, including the class name and instance variables (rep):

```
public class StockItem {
  String name;         // item name (e.g., "Fancy Feast")
  String size;         // size ("small", "12oz", etc.)
  String description;  // item description (e.g., "yummy food")
  int    quantity;     // number of copies of this item in stock

  /** Construct a new StockItem with the given data */
  public StockItem(String name, String size,
                   String decription, int quantity) { ... }
```

A summer intern was asked to implement an equals function for this class that treats two `StockItem` objects as equal if their `name` and `size` fields match. Here's the result:

```
  /** return true if the name and size fields match */
  public boolean equals(StockItem other) {
    return name.equals(other.name) && size.equals(other.size); }
```

(a) (4 points) This `equals` method seems to work some of the time but not always. Give an example showing a situation where it fails.

(continued next page)

Question 6 (cont.) (b) (4 points) Show how to fix the `equals` method given above so it works properly and has the intended meaning (e.g., `StockItem`s are equal if their names and sizes are equal)

(you won't need all this space probably….)

(continued next page)

Question 6. (cont.) (8 points) (c) Here are four possible `hashCode` methods for class `StockItem`. For each of these `hashCode` methods, circle *legal* if the method is a correct implementation of `hashCode` for `StockItem`, as specified on the previous pages. Circle *wrong* if it is not a correct implementation.

(i) legal wrong

```
public int hashCode() {
  return name.hashCode();
}
```

(ii) legal wrong

```
public int hashCode() {
  return name.hashCode()*17+size.hashCode();
}
```

(iii) legal wrong

```
public int hashCode() {
  return name.hashCode()*17+quantity;
}
```

(iv) legal wrong

```
public int hashCode() {
  return quantity;
}
```

(d) (2 points) Of the four `hashCode` methods above, which one is the best and (briefly) why?

Question 7. (6 points) (specifications) Suppose we are specifying a method and we have a choice between either requiring a precondition (e.g., @requires: n > 0) or specifying that the method throws an exception under some circumstances (e.g., @throws IllegalArgumentException if n <= 0).

Assuming that neither version will be significantly more expensive to implement than the other and that we do not expect the precondition to be violated or the exception to be thrown in normal use, is there any reason to prefer one of these to the other, and, if so, which one? Give a brief (couple of sentences) justification along with your answer.

Question 8. (6 points) (specifications) Suppose we are trying to choose between two possible specifications for a method. One of the specifications S is stronger than the other specification W, but both include the behavior needed by clients. In practice, should we always pick the stronger specification S, always pick the weaker one W, or is it possible that either one might be the suitable choice? Give a brief justification of your answer, including a brief list of the main criteria to be used in making the decision.