Name:_____

CSE331 Autumn 2019, Midterm Examination
October 28, 2019

Please do not turn the page until 10:30.

Rules:

- After the exam starts, rip out the last page and do not turn it in.
- The exam is closed book, closed notes, closed electronics, closed mouth, open mind.
- **Please stop promptly at 11:20.**
- There are **100 points**, distributed **unevenly** among **7** questions (all with multiple parts):
- **The exam is printed double-sided.**

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. (**18** points) Consider this code, where we assume `a`, `b`, `c`, and `d` are all arrays holding `int` values.

```
// precondition: a.length = b.length and a.length = c.length and a.length = d.length
int i = 0;
while(i != a.length) {
    c[i] = min(a[i],b[i]); // min returns min of its arguments
    d[i] = max(a[i],b[i]); // max returns max of its arguments
    i = i + 1;
}
```

   (a) Given the precondition above, for each of the following potential post-conditions, indicate whether it *always holds* (i.e., it is a valid post-condition), *sometimes holds*, or *never holds*.

   i. `for all 0 <= i < a.length, (a[i]+b[i] = c[i]+d[i])`

   ii. `for all 0 <= i < a.length, (a[i] = c[i] OR a[i] = d[i])`

   iii. `(for all 0 <= i < a.length, a[i] = c[i]) OR (for all 0 <= i < a.length, a[i] = d[i])`

   iv. `for all 0 <= i < a.length, d[i] < c[i]`

   v. `for all 0 <= i < a.length, c[i] < d[i]`

   vi. The sum of all elements in `a` is $\leq$ the sum of all elements in `d`

   (b) Assume now the specified post-condition is just the first one above:
   `for all 0 <= i < a.length, (a[i]+b[i] = c[i]+d[i])`
   For each possible loop invariant for the loop in the code, indicate A, B, or C as follows:

   A. It is not a correct loop invariant.
   B. It is a correct loop invariant, but it is too weak to establish the post-condition.
   C. It is a correct loop invariant and it is strong enough to establish the post-condition.

   i. `for all 0 <= j < i, (a[j]+b[j] = c[j]+d[j])`

   ii. `for all 0 <= j < i, (a[j] = c[j] or a[j] = d[j]) and (b[j] = c[j] or b[j] = d[j])`

   iii. `for all 0 <= j < i, (a[j] = c[j])`

   iv. `for all 0 <= j < i, (c[j] <= d[j])`

   v. `for all 0 <= j < i, (c[j] = min(a[j],b[j]) and d[j] = max(a[j],b[j]))`

   vi. `for all 1 <= j < i, (c[j-1] <= c[j])`

Solution:

(a) i. always holds
   ii. always holds
   iii. sometimes holds

iv. never holds
v. sometimes holds
vi. always holds

(b) i. C
ii. B
iii. A
iv. B
v. C
vi. A

Name:_____

2. (**12** points) Here are four specifications and three implementations:

```
S1: @requires i >= 0 && i < max
    @returns i+1

S2: @returns i+1

S3: @requires i >= 0
    @throws BadBadBad if i >= max
    @returns i+1

S4: @throws BadBadBad if i < 0 or i >= max
    @returns i+1
```

```
I1: int addOne(int i, int max) {
      if(i < max) {
        return i+1;
      } else {
        return -1;
      }
    }

I2:  int addOne(int i, int max) {
       return i+1;
     }

I3:  int addOne(int i, int max) {
       if (i >= max) {
          throw new BadBadBad();
       }
       return i+1;
     }
```

(For all questions below, if the answer is none, explicitly write "none".)

(a) List all the specifications above that `I1` satisfies.

(b) List all the specifications above that `I2` satisfies.

(c) List all the specifications above that `I3` satisfies.

(d) List all the specifications above that are stronger than `S1` (do not include `S1`).

(e) List all the specifications above that are stronger than `S2` (do not include `S2`).

(f) List all the specifications above that are stronger than `S3` (do not include `S3`).

(g) List all the specifications above that are stronger than `S4` (do not include `S4`).

Solution:

(a) S1

(b) S1, S2

(c) S1, S3

(d) S2, S3, S4

(e) none

(f) S4

(g) none

Name:_____

3. (**27** points) See the two classes defined on the last page of the exam. Rip out that page and do not turn it in. The two classes are two different attempts to implement an ADT that represents a mutable integer in the limited range from negative `max` to `max`, where `max` should be a positive number passed to the constructor. Operations are increment (add one), decrement (subtract one), negate, and convert to a `String`. If incrementing or decrementing would produce an abstract value outside the limited range, then the increment and decrement operations should instead do nothing.

   (a) Given the informal overview above, give a CSE331-style Javadoc specification for the constructor.

   (b) Given the informal overview above, give a CSE331-style Javadoc specification for the increment operation.

   (c) What does this code print?

```
MaxMagnitudeInt1 i1 = new MaxMagnitudeInt1(10);
i1.negate();
i1.increment();
System.out.println(i1.toString());
```

   (d) What does this code print? (Hint: It is *not* the same as the previous question.)

```
MaxMagnitudeInt2 i2 = new MaxMagnitudeInt2(10);
i2.negate();
i2.increment();
System.out.println(i2.toString());
```

   (e) Complete this code snippet to produce an even shorter example demonstrating that the two classes do not behave the same *and* indicate what is printed.

```
MaxMagnitudeInt1 i1 = new MaxMagnitudeInt1(10);

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

System.out.println(_____________);

MaxMagnitudeInt1 i2 = new MaxMagnitudeInt2(10);

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

System.out.println(_____________);
```

Problem continues with parts (f)-(j) on the next page.

Name:_____

(f) Rewrite the `negate` method in `MaxMagnitudeInt2` so that `MaxMagnitudeInt1` and `MaxMagnitudeInt2` always behave the same, meaning they satisfy all the same specifications.

(g) Assuming the new `negate` method you wrote in (f), write a `checkRep()` method for `MaxMagnitudeInt2` to describe its representation invariant.

(h) Assuming the new `negate` method you wrote in (f), in 1-2 English sentences, give the abstraction function for `MaxMagnitudeInt2`.

(i) Does `MaxMagnitudeInt2` suffer from representation exposure? (No explanation required.)

(j) If there are other methods for `MaxMagnitudeInt2` beyond those you see, is it possible that `MaxMagnitudeInt2` suffers from representation exposure? (No explanation required.)

Solution: See next page

Name:_____

Solution:

(a)
```
@spec.requires m > 0
@spec.effects a new max-magnitude int with an allowed range from -m to
m and current value 0
```

(b)
```
@spec.modifies this
@spec.effects increases the value represented by this by 1 unless the current
value is already (positive) max in which case it has no effect
```

(c) 1

(d) --1

(e)
```
MaxMagnitudeInt1 i1 = new MaxMagnitudeInt1(10);
i1.negate();
System.out.println(i1.toString()); // prints 0

MaxMagnitudeInt2 i2 = new MaxMagnitudeInt2(10);
i2.negate();
System.out.println(i2.toString()); // prints -0
```

(f)
```
public void negate() {
   if (magnitude != 0)
      isPositive = ! isPositive;
}
```

(g)
```
private void checkRep() {
   assert (max > 0 && magnitude >= 0 && magnitude <= max);
   if(magnitude == 0) {
      assert(isPositive);
   }
}
```

(h) The number currently represented is the value in `magnitude` if `isPositive` is true and the negation of `magnitude` if `isPositive` is false. The number cannot be changed to a value outside of the range `-max` and `max`.

(i) No

(j) No

Name:_____

4. (**11** points) This problem considers adding to `MaxMagnitudeInt1` (*not* `MaxMagnitudeInt2`) a logging/history feature that makes the history of values the object has had part of the abstract state, implemented as follows:

   - `negate`, `increment`, and `decrement` all have a call to `log();` added at the *beginning* of the method bodies.
   - We add this code to the class:

```
private List<Integer> history = new ArrayList<Integer>();
private void log() {
    history.add(val);
}
public int getNthNewestValue(int n) {
    if (n==0) {
        return val;
    } else {
        return history.get(history.size() - n);
    }
}
public List<Integer> getAllHistory() {
    return history;
}
```

   (a) Can `getNthNewestValue` cause representation exposure? If so, fix the method to avoid it.

   (b) Can `getNthNewestValue` throw an exception? If so, in one English sentence, give an appropriate precondition for clients that is sufficient to avoid an exception and does not leak implementation details.

   (c) Can `getAllHistory` cause representation exposure? If so, fix the method to avoid it.

   (d) Can `getAllHistory` throw an exception? If so, in one English sentence, give an appropriate precondition for clients that is sufficient to avoid an exception and does not leak implementation details.

Solution: See next page

Name:_____

Solution:

(a) No

(b) Yes, argument passed to `getNthNewestValue` must be greater than or equal to zero and less than or equal to the total number of previous calls to `negate`, `increment`, or `decrement` on the receiver.

(c) Yes

```
public List<Integer> getAllHistory() {
   return new ArrayList<Integer>(history);
}
```

(d) No

Name:_____

5. (**12** points) This problem considers `MaxMagnitudeInt2` (*not* `MaxMagnitudeInt1`). You can assume `MaxMagnitudeInt2` has your revised `negate` method from 3(f), but it doesn't actually matter.

   We consider adding this definition of `equals` to the class:

```
public boolean equals(Object o) {
    if(! (o instanceof MaxMagnitudeInt2))
        return false;
    MaxMagnitudeInt2 m = (MaxMagnitudeInt2) o;
    return m.magnitude == this.magnitude
           && (this.magnitude == 0 || m.isPositive == this.isPositive);
}
```

   (a) Is this definition reflexive?

   (b) Is this definition symmetric?

   (c) Is this definition transitive?

   (d) Is it possible for `o1` and `o2` to be bound to instances of `MaxMagnitudeInt2` such that the assertion below fails?

```
if(o1.equals(o2)) {
  o1.increment();
  o1.increment();
  o1.decrement();
  o1.decrement();
  assert(o1.equals(o2));
}
```

   (e) Is it possible for `o1` and `o2` to be bound to instances of `MaxMagnitudeInt2` such that the assertion below fails?

```
if(o1.equals(o2)) {
  o1.increment();
  o2.increment();
  o1.decrement();
  o2.decrement();
  assert(o1.equals(o2));
}
```

   (f) Is this a correct `hashCode` implementation?

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

   (g) Is this a correct `hashCode` implementation?

```
public int hashCode() {
  return magnitude + max;
}
```

Solution:

(a) yes

(b) yes

(c) yes

(d) yes

(e) yes

(f) yes

(g) no

Name:_____

6. (**15** points) (Testing)

We expect each answer below to be *brief*, roughly one English sentence.

(a) What makes a test a *black-box* test?

(b) Given only a test that passes, the specification for the code being tested, and the code being tested, can you tell if a test is a *black-box* test? Explain.

(c) What makes a test an *implementation* test?

(d) Given only a test that passes, the specification for the code being tested, and the code being tested, can you tell if a test is an *implementation* test? Explain.

(e) What makes a test a *regression* test?

(f) Given only a test that passes, the specification for the code being tested, and the code being tested, can you tell if a test is a *regression* test? Explain.

Solution:

(a) A black-box test is written without looking at the implementation for the code being tested.

(b) No, whether a test is black-box is about what the test-writer used to write the test, which cannot be determined from this information.

(c) An implementation test tests some property of the implementation that should hold given the implementation but which is not required by the specification.

(d) Yes, if the specification does not require the expected output of the test but the test passes with the implementation, then it is an implementation test — the provided information is enough because we can reason about the specification and see if other outputs would be allowed.

(e) A regression test is a test that some previous version of the code did not pass.

(f) No, we would need documentation of the test or previous versions of the software to know if the test is a regression test.

Name:_____

7. (**5** points) (Readings questions)

   (a) What does DRY stand for in the readings assigned?

   (b) Can a Java for-each loop always be used instead of a Java for loop even though style guidelines might prefer one or the other in different circumstances? (No explanation needed.)

   (c) Does "Design By Contract" mean, among other things, explicitly documenting method preconditions? (No explanation needed.)

   (d) Does "Design By Contract" mean, among other things, writing specifications before writing implementations? (No explanation needed.)

   (e) Does "Design By Contract" mean, among other things, that methods in Java should not use assertions to check that preconditions hold? (No explanation needed.)

Solution:

(a) Don't Repeat Yourself

(b) No

(c) Yes

(d) Yes

(e) No

Name:_____

This page is blank. If you put any answers on it, clearly indicate on the page with the question that you have done so.

Name:_____

Rip this page out and do not turn it in.

```
public class MaxMagnitudeInt1 {
    private int max;
    private int val = 0;

    public MaxMagnitudeInt1 (int m) {
        max = m;
    }

    public void negate() {
        val = - val;
    }

    public void increment() {
        if(val < max) {
            val++;
        }
    }

    public void decrement() {
        if(val > - max) {
            val--;
        }
    }

    public String toString() {
        return "" + val;
    }
}
```

```
public class MaxMagnitudeInt2 {
    private int max;
    private int magnitude = 0;
    private boolean isPositive = true;

    public MaxMagnitudeInt2 (int m) {
        max = m;
    }

    public void negate() {
        isPositive = ! isPositive;
    }

    public void increment() {
        if(magnitude == 1 && ! isPositive) {
            magnitude = 0;
            isPositive = true;
        } else if (isPositive) {
            if (magnitude < max) {
                magnitude++;
            }
        } else {
            magnitude--;
        }
    }

    public void decrement()  {
        if(magnitude == 0) {
            magnitude = 1;
            isPositive = false;
        } else if (isPositive) {
            magnitude--;
        } else {
            if (magnitude < max) {
                magnitude++;
            }
        }
    }

    public String toString() {
        String sign = "";
        if(!isPositive) {
            sign = "-";
        }
        return sign + magnitude;
    }
}
```