CSE 331 18wi Midterm Exam 2/8/18 Sample Solution

Remember: For all of the questions involving proofs, assertions, invariants, and so forth, you should assume that all numeric quantities are unbounded integers (i.e., overflow can not happen) and that integer division is truncating division as in Java, i.e., 5/3 => 1.

Question 1. (12 points) (Forward reasoning) The traditional warmup question. Using forward reasoning, write an assertion in each blank space indicating what is known about the program state at that point, given the precondition and the previously executed statements. Your final answers should be simplified. Be as specific as possible, but be sure to retain all relevant information.

(a) (5 points)

```
{ y > 0 }
x = 3;
{ y > 0 && x = 3 }
y = x + y;
{ y > 3 && x = 3 }
z = x * y;
{ y > 3 && x = 3 && z > 9 }
```

Note: y >= 4 && z >= 12 is also possible in the final assertion since y > 3 implies y >= 4 for integer values.

(b) (7 points)

```
{ |x| < 3 }
if (x < 0)
   { |x| < 3 && x < 0 } => { -3 < x < 0 }
   y = 3 + x;
   { -3 < x < 0 && 0 < y < 3 }
else
   { |x| < 3 && x >= 0 } => { 0 <= x < 3 }
   y = 2 * x;
   { 0 <= x < 3 && 0 <= y < 6 }

{ (-3 < x < 0 && 0 < y < 3) ||
                          (0 <= x < 3 && 0 <= y < 6) }
```

Note: It would also be possible in the else part of the conditional to simplify 0<=x<3 to get 0<=x<=2, in which case the condition after the y=2*x assignment would be 0<=x<=2 && 0<=y<=4, and the final assertion would need to be changed accordingly.

Question 2. (12 points) (Backward reasoning) The other traditional warmup question. 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 final answers if possible.

(a) (5 points)

```
{ 2*a+a-6 > 0 && a-6 > 0 } => { 3*a-6 > 0 && a-6 > 0 }
                      => { a > 2 && a > 6 } => { a > 6 }
b = a - 6;
{ 2*a+b > 0 && b > 0 }
c = 2*a + b;
{ c > 0 && b > 0 }
```

Note: Answers did not have to show all of the intermediate simplification steps, but the final answer should be simplified as shown.

(b) (7 points)

```
{ (x<=2 && x > -1 && x<2) || (x>2 && x>2 && x<5 } =>
   { (x > -1 && x < 2) || (x > 2 && x < 5) }
if (x <= 2) {
   { x+1 > 0 && x+1 < 3 } => { x > -1 && x < 2 }
   y = x + 1;
   { y > 0 && y < 3 }
} else {
   { x-2 > 0 && x-2 < 3 } => { x > 2 && x < 5 }
   y = x - 2;
   { y > 0 && y < 3 }
}
{ y > 0 && y < 3 }
```

Question 3. (15 points) Proofs. The following method is supposed to return true if its integer argument *n* is a power of 2 (i.e., 2^k for some *k*) and false otherwise. Write a suitable invariant and appropriate assertions to prove that it works correctly.

Hint: an extra variable *i* is included in the code. It is not part of the algorithm, but it might be helpful in your proof. If it is not useful, you can ignore it.

```
/** @param n Input number.
 *  @requires n >= 1
 *  @return true if n = 2^k for some integer k. */
public boolean isPowerOfTwo(int n) {
```

{ pre: **n_pre = n && n >= 1** }

```
   int i = 0;
```

{ inv: **n_pre = 2^i * n && n >= 1** }

```
   while(n%2 == 0) {
```

{ n_pre = 2^i * n && n >= 1 && n%2 = 0 } => { n_pre = 2^i * n && n > 1 }

```
      i = i + 1;
```

{ n_pre = 2^(i-1) * n && n > 1 }

```
      n = n / 2;
```

{ n_pre = 2^i * n && n >= 1 }

```
   } // end of loop
```

{ n_pre = 2^i * n && n >= 1 && n%2 != 0 } =>

{ n_pre = 2^i * n && n >= 1 && n%2 = 1 }

```
   boolean answer = (n==1);
```

{ post: **(answer && n_pre = 2^i*n && n>=1 && n%2=1) ||**
(!answer && n_pre = 2^i*n && n>=1 && n%2=1) }

=> { answer && n_pre = 2^i } ||
(!answer && n_pre = 2^i * n && n>1 && n%2=1) }

```
   return answer;
}
```

We allowed a fair amount of latitude in how the final postcondition was stated as long as the reasoning was there to show that answer is true if and only if the original value of `n` was a power of 2. The key observation is that when the loop terminates, `n` is odd and `n` >= 1. If `n` = 1 then we have determined that `n_pre = 2^i` for some value of i. If `n` > 1 and `n` is odd, then `n` cannot be a power of 2. The invariant says that `n_pre = 2^i*n`, and, if either of the factors in the product (`n` in this case) is not a power of 2, then `n_pre` cannot be a power of 2.

Question 4. (12 points, 3 points each) Specifications and implementations. Each of the following code fragments contains a short method specification and implementation. Each one contains an error: either (i) the specification itself is flawed because it is inconsistent, incomplete, or includes inappropriate things, or is contradictory [specerror]; or (ii) the specification is fine, but the implementation does not satisfy the specification [incorrectimpl]; or (iii) the code is implemented badly or is of poor quality or style or has other problems even though it satisfies the spec. [quality].

For each part, circle the error type, and give a brief (20 words or less) description of the problem and how to fix it. If there is more than one kind of error, identify and describe only the most important one. Specifications have been shortened to omit some parts and save space; the errors are not due to omitted JavaDoc parts in the specifications.

(a)
```
private Random random;

/** @return a random color.
  * @requires this.random is non-null
  */
public String getRandomColor() {
  String[] colors = { "blue", "red", "yellow" };
  return colors[this.random.nextInt(3)];
}
```

Error kind: (specerror) incorrectimpl quality

Reason and how to fix?

Information about private rep variable included in spec. Fix: Remove `@requires` clause.

(b)
```
/** @param list a List of integers
 *  @requires list has at least one element
 *  @return the smallest integer in the list
 */
public int min(List<Integer> list) {
  Collections.sort(list);
  return list.get(0);
}
```

Error kind: specerror (incorrectimpl) quality

Reason and how to fix?

Implementation modifies argument list. Fix: search the list to find the value without modifying the list.

(continued on next page)

Question 4. (cont.)

(c)
```
public class IntQueue {
  private List<Integer> queue;

  /** @return the elements of this queue as a List.
    */
  public List<Integer> getIntegers() {
    return queue;
  }
  ...
```

Error kind: specerror incorrectimpl (quality)

Reason and how to fix?

Rep exposure. Fix: Either return a copy of the list or return a wrapped unmodifiable copy.

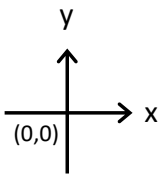
(d)
```
public class IntQueue {
  private List<Integer> queue;

  /** @return number of elements in the queue.
    */
  public int length() {
    int length = 0;
    for (Integer i : queue) {
      ++length;
    }
    return length;
  }
  ...
```

Error kind: specerror incorrectimpl (quality)

Reason and how to fix?

Unnecessary code. Fix: use `queue.size()` to compute answer.

The next several questions concern two classes for representing points and rectangles on a 2-D plane. These classes use standard Cartesian *x, y* coordinates, with the origin in the center, *x* coordinates increasing to the right, and *y* coordinates increasing going up.



We are given a class `Point` with the following specifications (condensed to save space).

```
/** A Point is an immutable object that represents a
  *  location (x,y) on a 2-D plane. */
public class Point {
  // rep and other private details omitted

  // Creators:
  /** Construct a new Point with coordinates (x,y) */
  public Point(int x, int y) { ... }

  // Observers: return x or y coordinate
  public int getX() { ... }
  public int getY() { ... }

  // equals/hashcode
  /** Return true if other is a Point with the same (x,y)
    *  coordinates as this, otherwise return false */
  public boolean equals(Object other) { ... }

  /** Return an appropriate hash code for this */
  public int hashCode() { ... }
} // end of class Point
```

We will use this class `Point` to implement class `Rectangle`, which has the following overview:

```
/** A Rectangle is a mutable object on a 2-D plane.  Its
 *  corners are specified by two points on the plane ul and
 *  lr, which give the coordinates of the upper-left and
 *  lower-right corners of the Rectangle.  All Rectangle
 *  objects have a height and width of at least 1, i.e.,
 *  a rectangle is not empty. That implies the coordinates of
 *  the top and bottom differ by at least one, and also the
 *  coordinates of the left and right sides differ by at
 *  least one.  It is possible to change the coordinates of
 *  the corners of a Rectangle as long as the upper-left
 *  corner remains above and to the left of the lower
 *  right corner.
 */
public class Rectangle { ... }
```

Remove this page from the exam and use it to answer the following questions. Do not write on this page or include it with the rest of the exam when you turn it in.

Question 5. (4 points) Constructor and rep exposure. The representation of a `Rectangle` consists of two `Point` objects giving the corner coordinates. Here are the declarations of those instance variables and the constructor code that initializes them. (We will use this rep in all remaining questions about `Rectangles`.)

```
public class Rectangle {
  private Point ul;   // upper-left corner
  private Point lr;   // lower-right corner

  /**
   * Construct a new Rectangle
   * @effects Make a new rectangle with given corners
   * [other specification details omitted]
   */
  public Rectangle(Point ul, Point lr) {
    this.ul = ul;
    this.lr = lr;
    checkRep();
  }
```

Are there any potential representation exposure problems if we have these instance variables and they are initialized in this way by this constructor? (circle)

Yes (No)

Give a brief (a couple of sentences max) explanation for your answer.

`Point` objects are immutable. Sharing them with client code does not create a rep exposure problem.

Question 6. (12 points) RI, AF, and checkRep. (Hint: these are pretty simple – don't panic if the answers are short.)

(a) (3 points) Give a suitable Representation Invariant (RI) for class `Rectangle`.

`ul != NULL`
`lr != NULL`
`ul.getX() < lr.getX()` (left coordinate is at least one less than right)
`ul.getY() > lr.getY()` (top coordinate is at least one greater than bottom)

(b) (3 points) Give a suitable Abstraction Function (AF) for class `Rectangle`.

AF(this) = a 2-D rectangle with upper-left corner coordinates given by `this.ul` and lower-right corner coordinates given by `this.lr`.

(Note: did not require a specific format for the answer as long as it clearly described how the AF was a function from concrete rep variables to the abstract value.)

(c) (6 points) Complete the implementation of method `checkRep` for class `Rectangle`. This method should verify as much of the rep invariant as is reasonable. If no error is detected, `checkRep` should return quietly without doing anything further.

```
/** Terminate execution with an assertion failure if a
 *  violation of the rep invariant is discovered. */
public void checkRep() {

    assert ul != null;
    assert lr != null;
    assert ul.getX() < lr.getX();
    assert ul.getY() > lr.getY();

}
```

Question 7. (18 points) Specification and implementation. We would like to add two methods to our `Rectangle` class. For each of these methods, give a proper JavaDoc specification and an implementation of the method. The first line of the JavaDoc comment and the method heading are supplied for you. You need to write the rest of the Javadoc (including @ tags) and the method code. Hint: don't forget `checkRep`.

If there is more than one reasonable way to write a specification or method, pick the one that seems best. You do not need to explain your choice(s).

(a) (10 points) `contains`. If `r` is a `Rectangle` and `p` is a `Point`, then `r.contains(p)` should return true if `p` is inside `r` and false if it is not. If `p` lies on the border (an edge or corner) of `r`, then it is considered to be "inside" `r`, and the method should return true.

```
/** Test whether a point is contained in this.
 *
 * @param p Point to test for containment
 *
 * @requires p != null
 *
 * @return true if p is inside or on the border of this
 *
 */
public boolean contains(Point p) {

    checkRep();

    int px = p.getX();

    int py = p.getY();

    return px >= ul.getX() && px <= lr.getX() &&
           py <= ul.getY() && py >= lr.getY();

}
```

Grading notes:

- **The specification needs to say how the case of `p==null` is handled. A precondition is reasonable (as above), or the method could be defined to throw an `IllegalArgumentException()` if `p` is `null`. If the specification requires throwing an exception if `p` is `null`, the necessary code must be included in the implementation.**
- **A real implementation of `contains` should call `checkRep()`, but we did not deduct points if that was omitted for a test question.**

Question 7. (cont.) (b) (8 points) `setUL`. If `r` is a `Rectangle` and `p` is a `Point`, then `r.setUL(p)` should change the upper-left corner of the rectangle so it has the coordinates given by point `p`. (Hint: the implementation might be very simple, but it is important to provide an appropriate specification. Think about the rep invariant (RI), and abstraction function (AF) for `Rectangle` and be sure the specification covers any unusual cases.)

```
/** Change the upper-left corner of this.
 *
 * @param p Point with new upper-left corner coordinates
 *
 * @requires p != null && p.getX() < this.lr.getX() &&
 *                        p.getY() > this.lr.getY()
 *
 * @modifies this
 *
 * @effects upper-left corner coordinates set to p
 *
*
 *
 */
public void setUL(Point p) {

  this.ul = newUL;

  checkRep();

}
```

Notes: As with `contains`, the specification must have a precondition or specification of how a `null` value for `p` is handled, and must specify what happens if the new upper-left corner is not, in fact, above and to the left of the lower-right. This could be done as above by including an appropriate precondition, or by specifying that an appropriate exception will be thrown and including code to do that.

As with part (a), we did not deduct points if the call to `checkRep` was omitted, although it should be included in any realistic implementation, especially if the method otherwise does not verify its argument.

Question 8. (9 points, 3 each) Testing. Describe three distinct black-box tests that could be used to verify that the `contains` method from the previous problem works properly. Each test description should describe the test input and expected output. For full credit each test should be different in some significant way from the other tests (think about boundary conditions and subdomains, etc.). You do not need to provide JUnit or other code, just be sure you give a clear, precise description of each test.

There are many possible tests. Here are a several.

For each test, we assume we have a rectangle `r` with upper-left corner at x=0, y=2 and lower right corner at x=3 and y=0.

(Test inside) evaluate r.contains(new Point(1,1)). Result should be `true`.

(Test above and right) evaluate r.contains(new Point(5,5)). Result should be `false`.

(Test below and left) evaluate r.contains(new Point(-1,-1)). Result should be `false`.

(Test above) evaluate r.contains(new Point(1,3)). Result should be `false`.

(Test below) evaluate r.contains(new Point(1,-1)). Result should be `false`.

(Test left) evaluate r.contains(new Point(-1,1)). Result should be `false`.

(Test right) evaluate r.contains(new Point(4,1)). Result should be `false`.

(Test left edge) evaluate r.contains(new Point(0,1)). Result should be `true`.

(Test right edge) evaluate r.contains(new Point(3,1)). Result should be `true`.

(Test top) evaluate r.contains(new Point(1,2)). Result should be `true`.

(Test upper-left corner) evaluate r.contains(new Point(0,2)). Result should be `true`.

(Test lower-right corner) evaluate r.contains(new Point(3,0)). Result should be `true`.

(Test test bottom) evaluate r.contains(new Point(2,0)). Result should be `true`.

etc., etc.

For full credit the answer should have included specific coordinates for the rectangle and points involved in the tests. Answers that were less specific but clearly described reasonable tests received almost full credit.

Question 9. (12 points) `equals` and `hashCode`. Below, give implementations of suitable equals and `hashCode` methods for class `Rectangle`.

For `equals`, two `Rectangles` are considered equal if they have the same coordinates, i.e., their corners are equal.

For `hashCode`, your solution should return a good quality `hashCode` value, i.e., while "`return 42;`" does, in fact, meet the requirements for a `hashCode` method, it is not a good implementation.

You do not need to add to or complete the JavaDoc specifications – just write the code.

```
/** Return true if other is a Rectangle with the same
 *  coordinates as this.  */
@Override
public boolean equals(Object other) {

  if (! (other instanceof Rectangle)) {
    return false;
  }
  Rectangle r = (Rectangle) other;
  return this.ul.equals(r.ul) && this.lr.equals(r.lr);
}

/** Return a suitable hash code for this */
@Override
public int hashCode() {

  return 31*ul.hashCode() + lr.hashCode();

}
```

Notes: `Rectangle`'s `equals` should use `Point`'s `equals` method to test whether the corners for equality. While testing the x and y coordinates will likely give the "right" result, we want to be sure that `equals` and `hashCode` are consistent. We can only do that if `Rectangle`'s `equals` and `hashCode` use the corresponding functions in `Point`, which are known to be consistent with each other.

Other `hashCode` implementations are possible, but for high quality they should combine `ul.hashCode` and `lr.hashCode` in some reasonable way. This particular implementation is based on the advice in *Effective Java.*

Final questions, not related to the `Rectangle` or `Point` classes….

Question 10. (12 points, 4 points each part) Specifications and implementations.

Each part of this question gives two specifications and one implementation. For each part, you should (i) indicate which specification is stronger than the other, or indicate they are incomparable if neither one is stronger than the other, and (ii) indicate which of the specifications (one or both or neither) are satisfied by the given implementation.

(a)

Specification S1:

```
@requires x >= 10
@return -1 if x > 10 or 0 otherwise
```

Specification S2:

```
@requires x > 10
@return -1
```

Implementation:

```
public static int m1(int x) {
  int i = 0;
  if (x > 10) {
    i = -1;
  }
  return i;
}
```

(i) Which specification is stronger? (circle) (S1) S2 incomparable

(ii) Which specifications are satisfied by the implementation? (circle one, both, or neither as appropriate):

(S1) (S2)

(continued on next page)

Question 10. (cont.) (b)

Specification S1:

```
@throws IllegalArgumentException if k <= 0
@return the square of k
```

Specification S2:

```
@return the square of k
```

Implementation:

```
public static int m2(int k) {
  if (k <= 0) {
    throw new IllegalArgumentException();
  }
  return k*k;
}
```

(i) Which specification is stronger? (circle) S1 S2 (incomparable)

(ii) Which specifications are satisfied by the implementation? (circle one, both, or neither as appropriate):

(S1) S2

(c) Specification S1:

```
@requires a!=null && a.size() > 0
@modifies a
@effect remove the last element and all negative
        elements of a
```

Specification S2:

```
@requires a!=null && a.size() > 0 && all elements of a
          are positive (i.e., > 0)
@modifies a
@effect remove the last element of a
```

Implementation:

```
public static void m3 (List<Integer> a) {
  int sz = a.size()-1;
  for (int k = sz; k >= 0; k--) {
    if (k == sz || a.get(k) < 0) {
      a.remove(k);
    }
  }
}
```

(i) Which specification is stronger? (circle) (S1) S2 incomparable

(ii) Which specifications are satisfied by the implementation? (circle one, both, or neither as appropriate):

(S1) (S2)

Question 11. (2 free points) (All reasonable answers receive the points. All answers are reasonable as long as there is an answer. ☺)

(a) (1 point) What question were you expecting to appear on this exam that wasn't included?

All answers received credit.

(b) (1 point) Should we include that question on the final exam? (circle or fill in)

Yes

No

Heck No!!

$!@$^*% No !!!!!

No opinion / don't care

None of the above. My answer is _____.