CSE 331: Software Design & Implementation

Practice Midterm

SOLUTIONS

Name: _____ UW Email: _____@uw.edu

This exam contains 10 pages and 7 problems, worth 73 points total.

Instructions

- Do not open the exam until instructed to do so.
- This is a closed-book, closed-notes exam. No electronic devices.
- Write your name and UW email at the top of this page.
- For every question involving proofs, assertions, and invariants, assume each numeric quantity is an unbounded integer: overflow cannot happen and numbers have no fractional parts. Integer division truncates as in Java, so `5/3` evaluates to `1`.
- If you need more room, continue on the back of the page and write "continued on back" next to the problem.
- If a problem is ambiguous, write down the interpretation you chose and answer under that interpretation.
- Many of these problems have short answers even when the problem itself is long. Don't be alarmed.
- Budget your time. Point values are a rough guide to how long each problem should take.

Problem	Topic	Points	Score
1	Forward reasoning	5	
2	Backward reasoning	8	
3	ADT, rep invariant, abstraction function	10	
4	Writing specifications	12	
5	Loop invariant proof	16	
6	Testing	12	
7	Comparing specifications	10	
Total		**73**	

Problem 1: Forward reasoning (5 points)

Starting from the given assertion, fill in an appropriate assertion at each blank. Simplify your final answer where possible, but do not weaken it in the process.

```
{ x >= 5 }
y = 3 * x;
{ x >= 5 and y = 3*x }
x = x + 2;
{ x >= 7 and y = 3*(x-2) }
y = y + 1;
{ x >= 7 and y = 3*x - 5 }
```

Solution. The last assertion comes out of the substitution as `x >= 7 && y-1 = 3*(x-2)`, which simplifies to `x >= 7 && y = 3*x - 5`. Either form earns full credit; the simplification must not drop the `x >= 7` conjunct.

Problem 2: Backward reasoning (8 points)

Find the weakest precondition under which the code below establishes the given postcondition. Fill in an appropriate assertion at each blank and simplify your final answer where possible.

```
{ (x > 0 and y > 0) or (x <= 0 and x > 1)     ⟺     x > 0 and y > 0 }
if (x > 0) {
    { y + 1 > 0 and y > 0     ⟺     y > 0 }
    x = y + 1;
    { x > 0 and y > 0 }
} else {
    { x > 0 and x - 1 > 0     ⟺     x > 1 }
    y = x - 1;
    { x > 0 and y > 0 }
}
{ x > 0 and y > 0 }
```

Solution. Notice how the simplification at the top completely eliminates everything after the "or". One way to think about this is to reason forwards: The else branch is taken if `x <= 0`, which means after `y = x - 1`, we would have `y <= -1`, which violates the desired postcondition. So, working backwards, the weakest precondition to establish the desire postcondition must guarantee the else branch is not taken.

Reference code for Problems 3–6

Problems 3 through 6 all concern the two classes below. Problem 7 is independent of them.

Point

`Point` is an immutable 2-D point on the plane in rectangular coordinates.

```
public class Point {
  // instance variables -- coordinates
  private final int x, y;

  // constructor
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  // observer methods
  public int getX() { return this.x; }
  public int getY() { return this.y; }

  public boolean equals(Object o) {
    // you may assume this works properly
  }

} // end class Point
```

FinitePointBag

`FinitePointBag` is a mutable unordered collection of `Point` objects, possibly containing duplicates, with a finite capacity.

```
public class FinitePointBag {
  // instance variables: uses an array and size instead of a List
  private Point[] points;
  private int size;

  // construct a new FinitePointBag with given capacity, which
  // cannot be negative
  public FinitePointBag(int maxItems) {
    points = new Point[maxItems];
    size = 0;
  }
  // = capacity of this
  public int capacity() { return points.length; }
  // = current number of items in this
  public int size() { return size; }
  // add a new point if space available; return true if succeeded
  public boolean add(Point p) {
    if (size < capacity()) {
      points[size] = p;
      size++;
      return true;
    } else {
      return false;
    }
  }
  // return true if p is contained in this, otherwise false
  public boolean contains(Point p) {
    for (int k = 0; k < size(); k++) {
      if (points[k].equals(p)) {
        return true;
      }
    }
    return false;
  }
  // remove all copies of p from this
  public void removeAll(Point p) {
    int i, k;
    i = 0;
    k = 0;
    while (k != size) {
      if (!points[k].equals(p)) {
        points[i] = points[k];
        i = i + 1;
      }
      k = k + 1;
    }
    size = i;
  }
} // end of FinitePointBag
```

Problem 3: ADT, rep invariant, abstraction function (10 points)

Before we can work with `FinitePointBag` we need to understand the ADT it represents. Your answers must be consistent with the instance variables and code given above.

(a) (3 points) Give a suitable description of the `FinitePointBag` ADT and its abstract value, as would normally appear in the Javadoc comment directly above the class definition. (Hint: the answer might be quite short.)

Solution. A `FinitePointBag` is a mutable, unordered collection of points {p1, p2, …, pn}, possibly containing duplicates, with a finite capacity.

(b) (4 points) Give a suitable representation invariant (RI) for `FinitePointBag`.

Solution. `points != null && 0 <= size <= points.length`, and all entries in `points[0..size]` are not equal to `null`.

(c) (3 points) Give a suitable abstraction function (AF) for `FinitePointBag`.

Solution. `points.length` is the capacity of `this`. The points in `this` are `points[0..size]`.

Problem 4: Writing specifications (12 points)

As usual, whoever writes these exams doesn't provide proper specifications for things. Write a correct CSE 331-style specification for the `contains` and `removeAll` methods of `FinitePointBag`. Your answers must be consistent with the given code and what it actually does when executed.

```
/** Return true if p is contained in this, otherwise false
  *
  * @param p the Point to search for in this
  * @requires p != null
  * @return true if p is an element of this, false otherwise
  */
public boolean contains(Point p) { ... }
```

```
/** Remove all copies of a point from this FinitePointBag
  *
  * @param p the Point to remove from this
  * @requires p != null
  * @modifies this
  * @effects all items that are equal to p are removed from this
  */
public void removeAll(Point p) { ... }
```

Problem 5: Loop invariant proof (16 points)

We think `removeAll` is correct, but we would like to prove that it really does remove every point equal to `p`.

To keep this manageable, you do **not** need to track history variables or anything else required to prove that the values copied are the original values from the array. Just show that the `FinitePointBag` contains no copies of `p` after the method executes.

You may write `{inv}` to refer to the loop invariant rather than restating it each time.

```
// remove all copies of p from this FinitePointBag
public void removeAll(Point p) {
  { p != null and RI }
  int i, k;
  i = 0;
  k = 0;
  {inv: 0 <= i <= k <= size <= points.length
        points[0..i] are points not equal to p }
  while (k != size) {
    { inv and k != size }
    if (!points[k].equals(p)) {
      { points[0..i] != p and points[k] != p }
      points[i] = points[k];
      { points[0..i+1] != p }
      i = i + 1;
      { points[0..i] != p }
    } else { // no code -- place for any needed assertions
      { points[0..i] != p }
    } // end if
    { points[0..i] != p }
    k = k + 1;
    { inv }
  } // end while
  { inv and k = size, so points[0..i] != p and all examined }
  size = i;
  { points[0..size] != p }
}
```

Problem 6: Testing (12 points)

The proof gives us good confidence that `removeAll` works, but we still need to test it. Write tests for this method. Then explain your approach to testing.

Solution. Many answers are acceptable, as long as you achieve complete coverage and discuss your reasoning.

```
Point p1 = new Point(1,2);
Point p2 = new Point(3,4);
FinitePointBag g = new FinitePointBag(5);

g.removeAll(p1);               // 0 iterations
assertEqual(0, g.size());

g.add(p1);
g.removeAll(p1);               // 1 iteration, not going into if
assertEqual(0, g.size());
assertFalse(g.contains(p1));

g.add(p1);
g.add(p2);
g.add(p1);
g.removeAll(p2);               // 2+ iterations, both in and out of if
assertEqual(2, g.size());
assertTrue(g.contains(p1));
assertFalse(g.contains(p2));

// some other cases potentially worth testing:
// * size > 1 and all elements are removed
// * size > 1 and no elements are removed
// * size > 1 and the element to be removed is the most recently added
// * size > 1 and the element to be removed is the least recently added
// * the element to be removed appears twice in adjacent indices
```

Problem 7: Comparing specifications (10 points)

Here are four possible specifications for a method that allocates an `int` array whose length is given by the parameter `n` and returns a reference to the newly allocated array.

```
A.  @param n
    @return a new int array with n elements if n > 0,
            otherwise return null

B.  @param n
    @requires n > 0
    @return a new int array with n elements

C.  @param n
    @requires n >= 0
    @return a new int array with n elements if n > 0,
            otherwise return null

D.  @param n
    @return a new int array with n elements
    @throws IllegalArgumentException if n <= 0
```

- List all specifications stronger than A (do not include A): **none**
- List all specifications stronger than B (do not include B): **A, C, D**
- List all specifications stronger than C (do not include C): **A**
- List all specifications stronger than D (do not include D): **none**
- Is it possible for a single implementation to satisfy both A and B? (yes or no) **yes**
- Is it possible for a single implementation to satisfy both B and C? (yes or no) **yes**

Solution.

- B has the most restrictive precondition (`n > 0`) and promises the least, so anything that tightens the postcondition or accepts more inputs is stronger than it.
- A and D are incomparable to each other: A accepts all `n` and returns null on non-positive input, while D accepts all `n` and throws instead. Neither refines the other.
- For the last two, an implementation only has to behave correctly on inputs each spec admits, so overlapping specs with compatible behavior on their shared domain can be satisfied at once.