CSE 331 19sp Midterm Exam 5/6/19 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 cannot happen and there are no fractional parts to numbers) and integer division is truncating division as in Java, i.e., 5/3 => 1.

Question 1. (12 points) (Backward reasoning) A 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)

```
{  |x+1| > 3  } => { x+1>3 || x+1<-3 }
                                    => { x>2 || x<-4 }
y = x + 1;
{  |2y| > 6  } => { |y| > 3 }
z = 2 * y;
{ |z| > 6 }
```

(b) (7 points)

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

Question 2. (16 points) Fibonacci! Recall that the Fibonacci numbers fib(k) are defined as fib(0) = 0, fib(1) = 1, and fib(k) = fib(k-1) + fib(k-2) for k ≥ 2. The following method is alleged to return fib(n). Write a suitable invariant and appropriate assertions to prove that it works correctly. You should assume that the method works correctly for n < 2 and do not need to handle that case. Provide the correct assertions and proof for n ≥ 2.

```
/** @return fib(n) for n >= 0. @requires n >= 0. */
public int fib(int n) {
   if (n < 2) return n;  // base case - ignore in proof
   { n ≥ 2 }
   int k = 1;
   int fibk = 1;
   int fibprev = 0;
   { inv: fibk = fib(k) && fibprev = fib(k-1) }
   while(k < n) {
      { inv }
      int fibnext = fibk + fibprev;
      { inv && fibnext = fib(k+1) }
      fibprev = fibk;
      { fibk = fib(k) && fibnext = fib(k+1) && fibprev = fib(k) }
      fibk = fibnext;
      { fibk = fib(k+1) && fibprev = fib(k) }
      k = k + 1;
      { inv: fibk = fib(k) && fibprev = fib(k-1) }
   } // end of loop
   { post: k=n && fibk=fib(k) } => { fibk = fib(n) }
   return fibk;
}
```

Note: the loop condition should have been k!=n instead of k<n (a typo). With k!=n as the loop condition it is trivial to conclude that k=n at the end of the loop. It is possible to prove by induction that we must have k=n at the end of the loop given that k=1 && n≥2 initially and k increases by 1 on each loop iteration. But since that is clearly true we didn't expect answers to prove that or even to argue it informally.

Many assertions and invariants contained things like fib(k+1) = fibprev+fibk. But that does not contain enough information to actually assert which values are contained in which variables as the values are changed by the assignments. It is also impossible to assert anything about the final value of fibk to prove that the method returns the correct result.

Question 3. (9 points) Here are three specifications and three methods that might implement them. Only the parts of the specifications that are different are shown. All of the specifications should include `@param amount`, but that is omitted to save space.

Spec. A:
```
/** @effects decrease balance by amount */
```

Spec. B:
```
/** @requires amount >= 0 and amount <= balance
 *  @effects decrease balance by amount */
```

Spec. C:
```
/** @throws InsufficientFundsException if balance < amount
 *  @effects decreases balance by amount */
```

Impl. 1:
```
void withdraw(int amount) {
    balance = balance - amount;
}
```

Impl 2:
```
void withdraw(int amount) {
    if (balance >= amount) {
        balance = balance - amount;
    }
}
```

Impl 3:
```
void withdraw(int amount) {
    if (amount < 0) {
        throw new IllegalArgumentException();
    }
    balance = balance - amount;
}
```

In the following grid, place an X in the square if the given implementation satisfies the give specification. If the implementation does not satisfy the specification, leave the square blank.

	Spec A	Spec B	Spec C
Impl 1	**X**	**X**	
Impl 2		**X**	
Impl 3		**X**	

Question 4. (14 points, 2 each) `equals` and method calls. One of the summer interns who does not know Java very well has been playing around trying to define classes for things made at a bakery. The intern is completely baffled by the behavior of this code, which attempts to define equality for cakes and chocolate cakes. (Yes, this code does not define `equals` correctly, but the question is asking about what actually happens given the code that's here. We will also leave aside the philosophical question of whether other cakes can ever be equal to chocolate cakes in real life.) The code does compile and execute without crashing.

```
public class Cake {
  protected int size;  // visible in subclasses but not
                       //    to outside clients

  public Cake(int size) {
    this.size = size;
  }

  public boolean equals(Cake other) {
    return this.size == other.size;
  }
}

public class ChocolateCake extends Cake {
  private String kind;   // kind of chocolate

  public ChocolateCake(int size, String kind) {
    super(size);
    this.kind = kind;
  }

  public boolean equals(ChocolateCake other) {
    return this.size == other.size &&
           this.kind.equals(other.kind);
  }

  public static void main(String[] args) {
    Cake cake1 = new Cake(1);
    Cake cake2 = new Cake(1);
    Object objectCake1 = (Object) cake1;
    Object objectCake2 = (Object) cake2;
    ChocolateCake chocolateCake =
                    new ChocolateCake(1, "dark chocolate");
    // answer questions on the next page
    // about code inserted here.
  }
}
```

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

Question 4. (cont.) For each line of code below, indicate what happens if it is inserted by itself at the end of the main method on the previous page and executed. For each one, indicate which method is called during execution (`Object.equals`, `Cake.equals`, or `ChocolateCake.equals`) and whether the method call returns true or false. Circle the correct answers.

(a) `cake1.equals(cake2);`

Class whose `equals` method is executed: `Object` `Cake` `ChocolateCake`

Result: `true` `false`

(b) `cake1.equals(chocolateCake);`

Class whose `equals` method is executed: `Object` `Cake` `ChocolateCake`

Result: `true` `false`

(c) `chocolateCake.equals(cake1);`

Class whose `equals` method is executed: `Object` `Cake` `ChocolateCake`

Result: `true` `false`

(d) `objectCake1.equals(cake1);`

Class whose `equals` method is executed: `Object` `Cake` `ChocolateCake`

Result: `true` `false`

(e) `objectCake1.equals(chocolateCake);`

Class whose `equals` method is executed: `Object` `Cake` `ChocolateCake`

Result: `true` `false`

(f) `objectCake1.equals(cake2);`

Class whose `equals` method is executed: `Object` `Cake` `ChocolateCake`

Result: `true` `false`

(g) `cake1.equals(objectCake1);`

Class whose `equals` method is executed: `Object` `Cake` `ChocolateCake`

Result: `true` `false`

Consider the following class that represents items that are stored in a warehouse. A few of the methods in this class are provided below. Answer questions about this class on the following pages. (This code does compile without errors.)

```
/** A StockItem is a mutable object that represents an item
 *  in a warehouse inventory.  The information in a StockItem
 *  includes the name of the item, a category for the item,
 *  the location in the warehouse where it is stored, and the
 *  number of copies of this item currently in the warehouse.*/
public class StockItem {
  // instance variables
  private String name;
  private int quantity;
  private String category;
  private String location;

  // creators
  /** construct a new StockItem with given properties */
  public StockItem(String name, int quantity, String category,
                                                 String location) {
    this.name = name;
    this.quantity = quantity;
    this.category = category;
    this.location = location;
  }

  // observers
  public String getName() { return name; }
  public int getQuantity() { return quantity; }
  public String getCategory() { return category; }
  public String getLocation() { return location; }

  // mutator
  public void setQuantity(int q) { quantity = q; }

  // equals
  /** return true if this StockItem is equal to o */
  @Override
  public boolean equals(Object o) {
    if ( !(o instanceof StockItem) )
      return false;
    StockItem other = (StockItem)o;
    return this.name.equals(other.name) &&
           this.category.equals(other.category) &&
           this.location.equals(other.location);
  }
```

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. (8 points, 2 each) (hashCode) Since our `StockItem` class includes an `equals` method, we need to provide a suitable `hashCode` method to go with it. Here are four possible `hashCode` implementations. Each of them compiles. For each one you should indicate whether the implementation satisfies the contract (specification) for `hashCode` given the existing `equals` method in `StockItem` and, if it does, whether it is a good, adequate, or poor choice for `hashCode`. Put an X next to the best answer.

(a)
```
public int hashCode() {
        return this.name.hashCode();
    }
```
_____ Incorrect (does not satisfy the contract for `hashCode`)
__*__ Correct but poor quality
__X__ Correct with adequate quality (not terrible but not particularly great)
_____ Correct with good/high quality

***Note: A hashCode using the string `name` will be significantly better quality than something like the constant returned in part (b). However, the wording of the question was unclear enough that we decided to give credit for "poor" in this case.**

(b)
```
public int hashCode() {
        return 1;
    }
```
_____ Incorrect (does not satisfy the contract for `hashCode`)
__X__ Correct but poor quality
_____ Correct with adequate quality (not terrible but not particularly great)
_____ Correct with good/high quality

(c)
```
public int hashCode() {
        return this.name.hashCode() ^ this.quantity;
    }
```
__X__ Incorrect (does not satisfy the contract for `hashCode`)
_____ Correct but poor quality
_____ Correct with adequate quality (not terrible but not particularly great)
_____ Correct with good/high quality

Note: `quantity` is not used in `StockItem.equals`, so it cannot be used in `hashCode`. If it is used, then two `StockItems` that are equal might not have the same `hashCode`.

(d)
```
public int hashCode() {
        return this.name.hashCode() ^ this.category.hashCode() ^
          this.location.hashCode();
    }
```
_____ Incorrect (does not satisfy the contract for `hashCode`)
_____ Correct but poor quality
_____ Correct with adequate quality (not terrible but not particularly great)
__X__ Correct with good/high quality

The next questions use the `StockItem` class from the previous question. Suppose we now define a class to hold a collection of `StockItems`. Here is the start of the class definition:

```
/** A collection of StockItems {s1, s2, ..., sn}.  No two
 *  StockItems have the same name. */
public class Stock {
  // instance variable
  Private List<StockItem> items;  // StockItems in this
                                  //    Stock collection

  ...
}
```

Question 6. (6 points) (JavaDoc and specs) Our `Stock` class contains the following constructor which, alas, is missing the usual CSE 331-style specification. Complete the JavaDoc comment so it properly specifies the operation of this constructor with appropriate JavaDoc tags and fields. (For CSE 331-specific tags like requires, you can use either `@requires` or `@spec.requires` – both will receive full credit. Also, you almost certainly won't need all this space.)

```
  /** construct new empty Stock collection
   *
   *
   * @effects construct an empty Stock collection
   *
   *
   *
   *
   *
   */
  public Stock() {
    items = new ArrayList<StockItem>();
  }
```

Notes: `@modifies` is not appropriate in a constructor specification since the constructor initializes a new object and does not modify the state of an existing one. Similarly, `@returns` is not appropriate either since a constructor is not a value-returning method.

Question 7. (11 points) RI, AF, and checkRep (Hint: these are pretty simple – don't panic if the answers are short.) More questions about the `Stock` class, from the previous page. Your answers should be consistent with the instance variable and constructor code given there.

(a) (4 points) Give a suitable representation invariant (RI) for class `Stock`.

`items != null` and each entry in `items != null`,
and if 0 <= i, j < `items.size()` and i != j, then
`items.get(i).getName().equals(items.get(j).getName())` is false.

(It would also be fine to write that no two `StockItems` in `items` have the same name instead of the last part of the above)

(b) (3 points) Give a suitable abstraction function (AF) for class `Stock`.

A collection of `StockItems` where each element of `items` is a `StockItem` in the collection.

(c) (4 points) Complete the implementation of `checkRep()` for class `Stock`.

```
// terminate execution with an assertion failure if a violation
// of the rep invariant is discovered, otherwise return silently
private void checkRep() {
  assert items != null: "items is null";
  for (StockItem item: items) {
    assert(item != null): "element in items is null";
  }
  for (int i = 0; i < items.size(); i++) {
    for (int j = i+1; j < items.size(); j++) {
      assert !items.get(i).getName()
                          .equals(items.get(j).getName())
          : "duplicate item name found in items";
    }
  }
}
```

Note: Some of the explicit `null` checks could be omitted since this `checkRep` will fail if a method call is attempted on a `null` entry in items. The null checks were included in this solution to provide a place for useful error messages if an assertion fails. It would also be fine to check all possible pairs of entries in `items` for duplicate names as long as an entry is not compared to itself.

Question 8. (8 points) (Another specification) Our `Stock` class has the following method, which returns a list of all of the `StockItems` whose location matches the method parameter. As before, complete the JavaDoc comment so it properly specifies the operation of this method using CSE 331 specification conventions.

```
/** Return a list of StockItems whose location match the
 *  the method parameter.
 *
 * @param location the location whose matching StockItems
 *                 are to be included in the returned list.
 *
 *
 * @returns a new List containing all Stockitems in this whose
 *          location matches the requested location parameter.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */
public List<StockItem> getItemsAtLocation(String location) {
  List<StockItem> result = new ArrayList<StockItem>();
  for (StockItem item: items) {
    if (item.getLocation().equals(location)) {
      result.add(item);
    }
  }
  return result;
}
```

Note: It would also be plausible to have a precondition `@requires location != null`, and solutions that included that received full credit. But it turns out the method will work properly (by accident) even if `location` is `null`, because `equals` will return `false` in that case without causing an error.

Question 9. (10 points) Representation exposure. Take another look at the `getItemsAtLocation` method on the previous page.

(a) (2 points) Does this method create any potential representation exposure problems, either for this class or any other class? (circle)

(Yes) No

(b) (5 points) Give a brief, but complete explanation and justification for your answer to part (a). A few sentences should be sufficient. Answers that are correct but excessively long will not necessarily receive full credit.

The returned list contains references to `StockItems` that are also referenced by the `Stock` collection. Since these are mutable, a client can use the result from `getItemsAtLocation` to modify the representation of the `Stock` object.

(c) (3 points) If there are any potential representation exposure problems, give a brief but complete description of how to fix them (you do not need to write actual Java code). If there are no potential representation exposure problems, just write "none" to receive full credit for this part of the question.

A straightforward way would be to make a (deep) copy of the `StockItems` that have a matching location and are to be returned in the result list.

However, it would not be correct to return an unmodifiable collection or change the specification of `StockItem` to require it to be immutable. Both of those would violate the client's expectations about the return value promised by the original method specification.

Question 10. (12 points, 3 each) Testing. Describe four distinct black-box tests that could be used to verify that the `getItemsAtLocation` 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 **should not** provide JUnit or other code, just a clear, precise description of each test, and your descriptions should be a few lines each, at most. If you want to write a specific StockItem as part of a test you can use something like (name, quantity, category, location), i.e., (gum, 17, food, bin42), but you don't have to do this.

There are a huge number of possible tests. Here are a few. In general the tests should have precisely specified inputs and outputs

(a) Input or test setup:

Create an empty Stock object
Compute getItemsAtLocation("seattle")

Expected output:

Empty list

(b) Input or test setup:

Create a Stock object containing the single StockIem (ufo, 5, spaceship, area51)
Compute getItemsAtLocation("seattle")

Expected output:

Empty list

(continued on next page)

Question 10. (cont.)

(c) Input or test setup:

Create a Stock object containing the single StockIem (ufo, 5, spaceship, area51)
Compute getItemsAtLocation("area51")

Expected output:

List containing the single StockIem (ufo, 5, spaceship, area51)

(d) Input or test setup:

Create a Stock object containing these items:
(skittles, 17, candy, aisle3)
(doritos, 42, junk, aisle3)
(tofu, 4, notmeat, aisle4)
Compute getItemsAtLocation("aisle3")

Expected output:

List containing:
(skittles, 17, candy, aisle3)
(doritos, 42, junk, aisle3)

Some short-answer questions to wrap up.

Question 11. (3 points) Test metrics. There are several metrics that are used to measure test coverage. In alphabetical order, some of the coverage metrics we looked at were *branch*, *loop*, *path*, and *statement* coverage. One tradeoff between these is that the most comprehensive metrics are also the most expensive. Write a list of these four metrics **from *least* to *most*** expensive and comprehensive.

Statement

Branch

Loop

Path

Question 12. (3 points) Tests for bugs. One of the guidelines for testing is that when a bug is discovered, you should create a test for it and add it to the test suite permanently. The question is why should we retain this test forever? After all, once we've fixed the bug we no longer need the test, do we? (Be brief!)

The bug represents some sort of defect that was introduced into the program. Whatever circumstances led to that happening could potentially happen in the future for the same or different reasons. We want to retain the test that reveals this bug to ensure that we continue to check for it in case the problem recurs as the program evolves.

Queston 13. (3 points) Preconditions in specs. Suppose are writing a method and we have a choice between using `@throws IllegalArgumentException if x<0` and `@requires x>=0` in the method specification. Neither option is significantly harder or more expensive to implement. Which is the better choice to include if the method is going to be included in a widely distributed library, and why? (briefly!)

Use `@throws`. That means that the behavior of the method has a complete instead of partial specification, and clients can know exactly what will happen for all inputs, unlike a precondition, where the behavior can't be predicted or tested if the precondition is not met.

Question 14. (3 points) Checked vs. unchecked exceptions. Java makes a distinction between checked exceptions (things like `FileNotFoundException`) and unchecked ones (like `NullPointerException`). Checked exceptions are required to be included in a `throws` clause in a method heading if the method can throw them, while unchecked exceptions do not have this requirement. Why did the Java designers decide to treat checked exceptions this way rather than treating them like unchecked exceptions, which do not need to be declared as part of the method heading? (briefly!!)

Checked exceptions represent unusual results that are legitimate, if unusual or unexpected, outcomes of a computation. Because of the Java requirements, client code must either handle these situations or indicate as part of the client specification that they do not handle them. Either way the outcome cannot be accidentally ignored since client code is required to deal with it in some way.

Question 15. (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?

A popular choice here was forward reasoning. That was omitted to make the test a bit shorter.

(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 _____.

Variations on "no" were most popular, but there were a significant number of "yes" answers.