CSE 331: Software Design & Implementation

Practice Final Exam, Adapted from Spring 2019

SOLUTIONS

Name: _____ UW Email: _____@uw.edu

This exam contains 18 pages and 10 problems, worth 122 points total.

Instructions

- Do not open the exam until instructed to do so.
- This is a closed-book, closed-notes exam. No electronic devices (and, per the original: closed telepathy).
- Write your name and UW email at the top of this page.
- Two reference-code pages appear in the middle of the exam, each right before the group of questions that uses it.
- Keep your answers brief and to the point.
- 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.
- Budget your time. Point values are a rough guide to how long each problem should take.

Problem	Topic	Points	Score
1	Correct code via proofs	20	
2	Specifying CharQueue	12	
3	Specifying CharQueue's methods	12	
4	Debugging CharQueue	14	
5	Food class behavior	14	
6	Generics	10	
7	React	12	
8	Design patterns	8	
9	String subtyping	8	
10	Multiple choice	12	
Total		**122**	

Problem 1: Correct code via proofs (20 points)

Recall that Java supports multi-dimensional arrays, but they are different than the ones found in C and other languages. A 2-D Java array is really an array of arrays. The top-level array has one entry for each row pointing to the array containing the elements of that row, and each row can have a different length. So, for example, if we declare `int[][] a2d = {{1,2,3}, {4}, {}, {5,6,7,8}};` we have an array with 4 rows. The first row has three elements (1, 2, 3), the second row has one element (4), the third row is empty, and the fourth row has four elements. An individual element of an array can be referenced using the notation `a[r][c]`, and `a[r]` by itself references the array that is stored in row `r`. The expression `a.length` returns the number of rows in array `a`, and `a[r].length` returns the number of elements in row `r`.

(a) (12 points) The following method is alleged to return the total number of elements in a 2-D Java array. Provide a suitable loop invariant `inv` and assertions to prove that it returns the correct result.

```
public static int numElts(int[][] lst) {
   { pre: lst != null && for all 0<=i<lst.length, lst[i]!=null }
  int num = 0;
  int i = 0;
   { inv: 0 <= i <= lst.length and num = sum of lst[0].length to lst[i-1].length }
  while (i != lst.length) {
      { inv && i != lst.length }
    num += lst[i].length;
      { num = sum of lst[0].length to lst[i].length }
    i++;
      { inv }
  }
   { post: inv && i = lst.length =>
           num = sum of lst[0].length to lst[lst.length-1].length }
  return num;
}
```

Note: when grading the precondition, `lst!=null` needed to be specified, but we didn't deduct if the answer didn't mention that the elements of `lst` should not be `null` since that is a fairly subtle point. We also didn't deduct if `i!=lst.length` was omitted from the first assertion at the beginning of the loop since it is not used later in the proof.

(b) (8 points) The following method, `flatten`, returns all of the elements of a 2-D array in a single 1-D array. For example, the result of `flatten` on our example 2-D array containing `{{1,2,3}, {4}, {}, {5,6,7,8}}` is the 1-D array whose contents are `{1,2,3,4,5,6,7,8}`. Your only job in this part of the question is to provide a suitable loop invariant `inv` to complete the proof. All of the rest of the code and proof are provided for you. In the postcondition we write `a[i..j]` with the usual CSE 331 meaning: the elements of `a` at indices from `i` (inclusive) up to `j` (exclusive). This is not legal Java code, but is helpful for writing assertions in the proof.

Write the appropriate loop invariant in the blank line right before the start of the loop.

Note: the loop invariant is a bit subtle. It needs to capture three separate things: (i) all of the elements in previous rows of `lst` have been copied to the proper place in `res`, (ii) all of the elements prior to the current one in the current row have been copied to the proper place, and (iii) `iRes` has the correct relationship to the values in `iInner` and `iOuter`.

```
public static int[] flatten(int[][] lst) {
   { pre: lst != null && for all 0<=i<lst.length, lst[i] != null }
  int[] res = new int[numElts(lst)];
  int iOuter = 0;
  int iInner = 0;
  int iRes = 0;

   { inv: iRes = numElts(lst[0..iOuter]) + iInner &&
         for all 0 <= r < iOuter && 0 <= c < lst[r].length we have
               res[numElts(lst[0..r])+c] = lst[r][c] &&
         for all 0 <= c < iInner, we have
               res[numElts(lst[0..iOuter])+c] = lst[iOuter][c] }
  while (iRes != res.length) {
    if (iInner != lst[iOuter].length) {
      res[iRes] = lst[iOuter][iInner];
      iInner++;
      iRes++;
    } else {
      iOuter++;
      iInner = 0;
    }
  }
   { post: for all 0 <= i < lst.length and 0 <= j < lst[i].length,
           res[numElts(lst[0..i]) + j] = lst[i][j] }
  return res;
}
```

Reference code: the `CharQueue` class

The following code implements a very simple queue ADT. In the usual CSE 331 exam style, it is woefully lacking in various forms of documentation, it may not be correct (i.e., it may be buggy), and there may be other problems (or maybe it actually does work).

```
// a finite queue of characters
// add() can only be called a finite number of times before the
// queue is "used up", no matter how many times get() is called
public class CharQueue {
  // instance variables
  private char[] chars;
  private int current;  // current position
  private int end;      // end of data

  // construct new CharQueue with given capacity
  public CharQueue(int n) {
    chars = new char[n];
    current = 0;
    end = 0;
  }

  // append new character to queue if room.
  // return true if character appended, otherwise return false.
  public boolean add(char ch) {
    if (end < chars.length) {
      chars[end] = ch;
      end = end + 1;
      return true;
    } else {
      return false;
    }
  }

  // return oldest character in the queue and remove it
  // throw NoSuchElementException if queue contains no characters
  public char get() {
    if (current <= end) {
      char result = chars[current];
      current++;
      return result;
    } else {
      throw new NoSuchElementException();
    }
  }
}
```

Problems 2–4 refer to this code.

Problem 2: Specifying CharQueue (12 points)

Class specification. The `CharQueue` class on the reference page is lacking the appropriate CSE331-style documentation. For this question, supply a proper abstract description of the class, a rep invariant, and an abstraction function. You should base your specifications on the intended behavior inferred from the original code and comments.

(a) (4 points) Give an appropriate abstract description of the class that should appear in the JavaDoc comment right before the first line of the class.

```
/**
 * A CharQueue is a queue of characters c0,c1,...,cn where
 * c0 is the oldest element inserted in the queue and not
 * yet removed, and cn is the most recent element
 * inserted in the queue.  There is a finite limit on the
 * number of add operations that can be performed on the
 * queue regardless of how many elements have been removed.
 *
 */
public class CharQueue { ... }
```

(b) (5 points) Give a suitable representation invariant for this class. You should use the existing instance variables that are already present in the code.

Solution. `chars != null && 0 <= current <= chars.length &&`
`0 <= end <= chars.length && current <= end`

Note: `char` values can never be `null` – they are a primitive type – so including something like `chars[k]!=null` is an error.

(c) (3 points) Give a suitable abstraction function for this class. Your answer should use information from your answers to parts (a) and (b) of the question as needed.

Solution. The current elements in the queue are stored in `chars[current..end]`, where `chars[current]` is the oldest element `c0` in the queue and `chars[end-1]` is the most recently inserted element in the queue. (This implies that the queue is empty if `current == end`.)

Note: there are other possible rep invariants and abstraction functions, but these correspond closely to the existing code. For solutions that used a different RI and/or AF, we referenced the answers to this question when grading the next two.

Problem 3: Specifying CharQueue's methods (12 points)

Method specifications (6 points each). Give appropriate CSE331-style JavaDoc specifications for methods `add` and `get` of the `CharQueue` class on the reference page. Write the CSE 331 tags (`@requires`, `@modifies`, `@effects`, `@returns`, `@throws`) the way we do in class. You should base your specifications on the intended behavior inferred from the original code and comments.

```
/**
 * Add ch to the queue if there is room
 *
 * @param ch the character to be added
 *
 * @modifies this
 *
 * @effects adds ch to the end of the queue if room
 *
 * @returns true if ch was successfully added, or false
 *          if ch was not added because there was no room
 *          for it in the queue.
 *
 */
public boolean add(char ch) { ... }
```

```
/**
 *
 * Return the oldest character in the queue and delete that
 * character from the queue.
 *
 * @modifies this
 *
 * @effects removes the oldest element from the queue
 *
 * @returns the oldest element in the queue
 *
 * @throws NoSuchElementException if the queue is empty
 *
 */
public char get() { ... }
```

Solution. Note: some solutions included an additional precondition for the `add` method specifying `ch!=null`. That is an error, since `char`, like `int` and `double`, is a Java primitive type and variables with those types can never have the value `null` (`null` is not a value of any primitive type).

Problem 4: Debugging CharQueue (14 points)

There's a bug in the code!! (well, at least one)

(a) (4 points) What problematic behavior do you observe in the code and where is it located?

Solution. If `get` is called when a queue is empty, it will return an erroneous character value instead of throwing a `NoSuchElementException`.

(`get` can also throw an `IndexOutOfBoundsException` if it tries to retrieve a character one position past the end of the queue array.)

The implementation of `get` is incorrectly checking for an empty queue in the `if (current <= end)` test.

(b) (6 points) Write a JUnit test that demonstrates the presence of the defect by failing when executed with the original code.

There are several ways to do this, depending on which version of JUnit you're using. We allowed lots of partial credit if the intent was clear but the exact syntactic details were not quite right. However, we did not allow credit if an exception value was checked using something like `assertEquals` – a thrown exception is not a normal returned method result.

The JUnit 5 form we use in CSE 331 passes the exception class and a lambda to `assertThrows`:

```
@Test
public void testEmptyGet() {
  CharQueue q = new CharQueue(5);
  assertThrows(NoSuchElementException.class, () -> q.get());
}
```

Here is a standard JUnit 4 way to check for an exception:

```
@Test(expected = NoSuchElementException.class)
public void testEmptyGet() {
  CharQueue q = new CharQueue(5);
  char ch = q.get();
}
```

Another version (less elegant, but works fine and was standard before the above syntax was added to JUnit some years ago) is:

```
@Test
public void testEmptyGet() {
  try {
    CharQueue q = new CharQueue(5);
    char ch = q.get();
    fail("NoSuchElementException not thrown");
```

```
  } catch (NoSuchElementException e) {
    // ok to do nothing - expected behavior
  }
}
```

(c) (4 points) Describe how to repair the defect to fix the problem. After the repair, the test from part (b) should succeed. You can either give a very precise description of what to fix in the original code, or rewrite the original code for the modified method(s) below showing how to repair the defect. Be sure it is clear what you have changed.

In method `get()`, replace

```
if (current <= end) {
  ...
```

with

```
if (current < end) {
  ...
```

Reference code: the picnic `Food` classes

It's summer and it's time for picnics! But since it's a CSE 331 picnic, we have to write some code to keep track of the food. Here is the code.

```
/** food for picnics */
abstract class Food {
  /** request n servings of this Food item */
  abstract public void order(int n);
}

class Hotdog extends Food {
  public void order(int n)
                   { System.out.println(n + " hotdogs"); }
}

class Dessert extends Food {
  public void order(int n)
                   { System.out.println(n + " desserts"); }
  public void order()     { this.order(1); }
}

class Icecream extends Dessert {
  public void order()      { this.order(2); }
  public void order(int n) { System.out.println(n + " scoops"); }
}

class Cake extends Dessert {
  public void order()
              { this.order(2); System.out.println("cake"); }
}

class ChocolateCake extends Cake {
  public void order(int n)
              { System.out.println(n + " chocolate cake"); }
}
```

The following questions refer to this code.

Problem 5: Food class behavior (14 points)

(2 points each) Here are several groups of statements that might be found in a program that uses the classes on the `Food` classes reference page. For each group, if the statements compile and execute successfully without any errors, write the output that is produced. If there is an error, explain in a sentence what is wrong.

(a)

```
Food meat = new Hotdog();
meat.order(4);
```

Solution. `4 hotdogs`

(b)

```
Dessert yum = new ChocolateCake();
yum.order();
```

Solution. `2 chocolate cake`
`cake`

(c)

```
Cake yummy = new ChocolateCake();
yummy.order(2);
```

Solution. `2 chocolate cake`

(d)

```
Food strawberry = new Icecream();
strawberry.order();
```

Solution. Won't compile. Class `Food` does not contain an `order()` method.

(e)

```
Icecream vanilla = new Icecream();
vanilla.order();
```

Solution. `2 scoops`

(f)

```
Dessert chocolate = new Icecream();
chocolate.order();
```

Solution. `2 scoops`

(g)

```
Food lemon = new Cake();
lemon.order(3);
```

Solution. `3 desserts`

Problem 6: Generics (10 points)

(1 point each) And now for the dreaded generics question. ⌣̈ Using the `Food` class hierarchy from the `Food` classes reference page (before Problem 5), assume we have the following variables:

```
Object obj; Food food; Hotdog hot; Dessert des;
Icecream ice; Cake cake; ChocolateCake choc;

List<? extends Dessert> extd;
List<? extends Cake> extc;
List<? super Cake> supc;
```

For each of the following, circle OK if the statement has correct Java types and will compile without type-checking errors; circle ERROR if there is some sort of type error.

OK **ERROR** `extd.add(des);`

OK **ERROR** `extd.add(cake);`

OK ERROR `supc.add(cake);`

OK ERROR `supc.add(choc);`

OK **ERROR** `supc.add(des);`

OK ERROR `cake = extc.get(1);`

OK **ERROR** `cake = supc.get(1);`

OK **ERROR** `choc = extc.get(1);`

OK **ERROR** `choc = supc.get(1);`

OK **ERROR** `food = supc.get(1);`

Problem 7: React (12 points)

A little bit of React. We have discovered the following fragment of a React web app. `App` is the top-level component, and the page is rendered once. This code runs without errors.

```
import { useState } from "react";

export default function App() {
  const [favorite, setFavorite] = useState("Raspberry");
  return (
    <div>
      <Basket fruit={favorite} />
      <Basket fruit={"Apple"} veggie={"Broccoli"} />
    </div>
  );
}

function Basket(props) {
  const [fruit, setFruit] = useState("Blueberry");
  console.log(props);
  return <p>Basket of: {props.fruit}</p>;
}
```

(a) (6 points) What output is written to the console log when this code is executed?

Solution.
```
{fruit: "Raspberry"}
{fruit: "Apple", veggie: "Broccoli"}
```

Each `Basket` logs the props it was given, and nothing else. The `fruit` state declared inside `Basket` is not part of `props`, so `"Blueberry"` appears in neither answer.

(b) (6 points) What html code is generated when this code is executed?

Solution.
```
<div>
<p>Basket of: Raspberry</p>
<p>Basket of: Apple</p>
</div>
```

Problem 8: Design patterns (8 points)

(2 points each) Here is an approximate list of the design patterns (and other 331 "principles") we covered this quarter.

Adapter, Builder, Decorator, Dependency Injection, Factory (method or object), Interning, Observer, Proxy, Separating UI from data, Singleton, Visitor.

For each of the situations below, identify the most specific pattern or principle used in that situation. You only need to write the pattern name, you do not need to explain further. If there is more than one possible answer, pick the one that is the best match.

(a) Create an object by calling a single method to return the appropriate object rather than using the Java `new` operator.

Solution. Factory

(b) Creating an object by calling successive methods to add details and adjust the created object step-by-step rather than having a single constructor do all the work.

Solution. Builder

(c) Registering a function to be called when a user event happens (like a click on a GUI button)

Solution. Observer

(d) Writing a new class with the same behavior as another class, but while declaring that the new class implements an interface that the old class did not

Solution. Adapter

Problem 9: String subtyping (8 points)

A couple of short questions about types to wrap up.

In the Java language, type `String` is a Java subtype of `Object`. It also is true that the array type `String[]` is a Java subtype of `Object[]`.

(a) (5 points) From what we know about behavioral subtyping, is `String[]` actually a behavioral subtype of `Object[]`? Give a brief justification for your answer, either explaining why this subtyping relationship is correct, or give an example showing how this allows programs to typecheck with no errors at compile time and yet have a type error during execution.

This violates behavioral subtyping. `String[]` should not be a subtype of `Object[]` since it does not satisfy the specification for `Object[]`. An `Object` array can store any kind of object, but a `String` array cannot hold non-`String` objects, and cannot be substituted for an `Object` array. Given the Java array subtyping rules, the following code is correct (i.e., a compiler will not report any type errors when the code is compiled):

```
Object[] a = new String[10];
...
Object o = new Object();
a[0] = o;
```

(Java does insert a runtime type test that will cause this program to fail at runtime, but the program does not contain any static type errors.)

(b) (3 points) Why did the Java designers decide to specify that `String[]` is a Java subtype of `Object[]`? (You can explain this in terms of your answer to part (a) if that is useful.)

Solution. The original version of Java did not have generic types (type parameters). Including this subtyping rule made it possible for `Object[]` to be used as a "generic" container to store objects of any type and to be used in the underlying implementation of collections like lists.

Problem 10: Multiple choice (12 points)

Circle the best answer for each of the following questions.

(a) (2 points) Here are two specifications for a method `indexOf(int[] a, int x)`:

```
S1.  @requires a != null and x appears in a
     @return an index i such that a[i] == x

S2.  @requires a != null
     @return an index i such that a[i] == x, or -1 if x
             does not appear in a
```

A. `S1` is stronger than `S2`.
B. `S2` is stronger than `S1`.
C. The two specifications are equally strong.
D. Neither specification is stronger than the other.

Solution. Answer: B

`S2` has the weaker precondition because it accepts arrays that do not contain `x`. On the inputs that both specifications accept, `S2` promises exactly what `S1` promises. A weaker precondition with an equally strong postcondition makes `S2` the stronger specification: every implementation of `S2` also satisfies `S1`.

(b) (2 points) Here are two specifications for a method `f(int x)`:

```
S3.  @requires x >= 0
     @return an int that is greater than x

S4.  @requires x > 5
     @return x * 2
```

A. `S3` is stronger than `S4`.
B. `S4` is stronger than `S3`.
C. The two specifications are equally strong.
D. Neither specification is stronger than the other.

Solution. Answer: D

Each is stronger in one dimension and weaker in the other, so they are incomparable. `S3` has the weaker precondition: it accepts `0 <= x <= 5`, which `S4` rejects. `S4` has the stronger postcondition: it pins down the exact result, where `S3` allows any larger `int`. Neither one's set of legal implementations contains the other's.

(c) (2 points) A `Duration` class defines the `equals` method below, and defines no other `equals` method. What is printed?

```
public boolean equals(Duration d) {
  return this.min == d.min && this.sec == d.sec;
}
```

```
Duration d1 = new Duration(1, 2);
Duration d2 = new Duration(1, 2);
Object   o1 = d1;
Object   o2 = d2;
System.out.println(d1.equals(d2));
System.out.println(o1.equals(o2));
```

A. `true` then `true`
B. `true` then `false`
C. `false` then `false`
D. `false` then `true`

Solution. Answer: B

`equals(Duration)` does not override `Object.equals(Object)`; it *overloads* `equals`, so the class now has two of them. Java chooses between overloads using the *static* types of the operands. `d1` and `d2` are declared `Duration`, so the first call reaches the new method and prints `true`; `o1` and `o2` are declared `Object`, so the second call reaches `Object.equals`, which is reference equality, and prints `false`. The fix is to override `equals(Object)` instead. And of course we should override `hashCode` along with it.

(d) (2 points) Is `List<Integer>` a Java subtype of `List<Number>`?

A. Yes. A generic type is a subtype of another whenever its type argument is, so the relationship carries over from `Integer` and `Number`.
B. No. `List<Integer>` and `List<Number>` are unrelated types, neither one a subtype of the other.
C. No. The relationship is the other way around: `List<Number>` is a subtype of `List<Integer>`.
D. Yes. but only when the list is never modified through the supertype reference.

Solution. Answer: B

This is the invariance puzzle from the generics lecture: generic types are *invariant* in their type argument, so `List<Integer>` and `List<Number>` are unrelated types even though `Integer` is a Java subtype of `Number`. Java's arrays follow the opposite rule, whose consequences Problem 9 explores.

(e) (2 points) Consider this Dafny method and the client that calls it. Dafny rejects the assertion `w == 20`. Why?

```
method foo(x: int) returns (z: int)
  requires x >= 0
  ensures z >= x + 10
{
  return x + 10;
}

method Main() {
  var w := foo(10);
  assert w == 20;
}
```

A. The assertion is false: `foo(10)` does not return `20`.
B. Dafny reasons about the call from `foo`'s specification rather than its body, and `ensures z >= x + 10` guarantees only that `w >= 20`.
C. The call violates `foo`'s precondition.
D. Dafny cannot prove assertions about a variable assigned from a method call.

Solution. Answer: B

This is what having a specification means: the caller is verified against the `requires`/`ensures` contract, not against the implementation. The body does return exactly `20`, but the postcondition is weaker than the body, so all a caller may assume is `w >= 20`. Strengthening the specification to `ensures z == x + 10` makes the assertion verify.

(f) (2 points) You click a button in your web app and nothing appears on the page. The browser's network tab shows the request to `/api/messages` coming back with status `404`. What does that point to?

A. The server has no route for that path. Either it was never registered, or the path is spelled differently on the two sides.
B. The server's handler for that path threw an exception while it ran.
C. The request body the frontend sent was not valid JSON.
D. The `fetch` call is missing an `await`, so the response has not arrived yet.

Solution. Answer: A

A `404` is the server reporting that it has nothing registered at that path, so the mismatch is between the URL the frontend requested and the routes the backend defines. A handler that ran and failed would give a `500`, and a malformed body would typically give a `400`, and both of those mean the route was found. A missing `await` is a frontend bug that would produce no status code at all.