STUDENT NAME: ______________________________

CSE331 Autumn 2011 Midterm Examination

October 28, 2011

- 50 minutes; 75 points total.
- Open note, open book, closed neighbor, closed anything electronic (computers, web-enabled phones, etc.)
- An easier-to-read answer makes for a happier-to-give-partial-credit grader

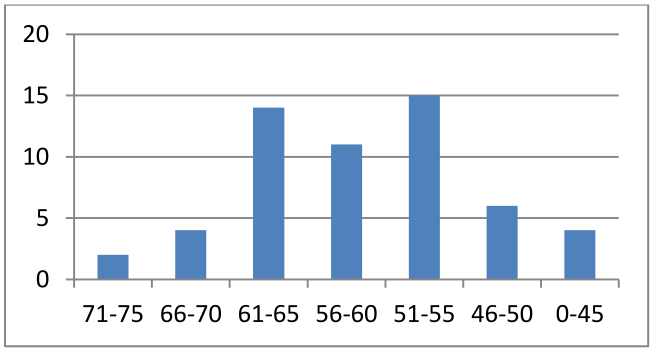
	Points	Your score
Part I: T/F with explanation	12	
Part II: Testing	22	
Part III: Specification	18	
Part IV: ADTs	20	
Part V: Miscellaneous	3	
Total	**75**	

Don't turn the page until the proctor gives the go ahead!

THIS VERSION CONTAINS THE ANSWER KEY
AND VARIOUS PERTINENT COMMENTS.

Statistics (per Part and Overall) and histogram:

Mean	8.2	19.1	14.7	12.3	3.0	57.2
Median	8	20	15.5	12	3	57
Mode	10	22	16	11	3	55
Min	4	10	8	6	3	39
Max	12	22	18	20	3	71



STUDENT NAME: ____

Part I: True/False with a one-sentence justification (12 points total)

1. A representation invariant for an ADT implementation must hold before and after each statement in every method of that implementation.

   FALSE: When ADT implementations make modifications to the internal representation during execution of a method they can break a representation invariant, and they must then re-establish the representation invariant before the method returns.

   Comments:
   • It's easy to find examples – for instance, the one (intended to show benevolent side-effects) on Slide 26 of the lecture ADT II is clear. The CharSet in the previous lecture also would show this, although the bodies of the implementation are only a single statement.
   • Many people who missed this simply parroted "representation invariants must hold at all times" without considering either than the data in a representation must change (usually meaning that the RI will be temporarily broken) or that during execution is different from between executions.

STUDENT NAME: ______________________________

2. Consider the `Duration` class presented in Lecture #:

```
public class Duration {
  private final int min;
  private final int sec;
  public Duration(int min, int sec) {
     this.min = min;
     this.sec = sec;
  }
}
```

True or false: The following `HashCode` definition is safe for instances of the `Duration` class:

```
public int hashCode() {
   return sec*min;
}
```

TRUE: This hashCode, deterministically based on sec and mind, never lets two equivalent Duration instances have different hashCodes.

Comments:

• It is essential to remember that hashCode is a prefilter for equals – to be safe, hashCode can return "false positives" (that is, the objects with identical hashCodes can be different when the complete equals check is done) but cannot return "false negatives" (that is, two objects with different hashCodes are in fact equal).

• A simple example to see this is in the slides about Duration, with the hashCode that always returns 1. It is safe (never rejects equal objects), albeit inefficient (always requires full equal checks).

3. If all constructors of a Java class C are declared to be `private`, no client of the class can create objects of type C.

FALSE: static public methods can do this, as can be seen in the JDK Boolean class with valueOf.

Comments:

• We discussed this with respect to the intern pattern, but it doesn't take patterns to understand or answer the question.

4. Extending a Java abstract class can sometimes produce a true subtype.

   TRUE: The resulting class can (but Java does not constrain it to be so) have a stronger specification than the abstract class.

   Comments:
   • Since strength of specification – the key for true subtyping – is a question of specification not implementation, it is not material that the abstract class omits (some) implementations.
   • This was a weak question – it was general enough to make it hard to tell how much people knew and how much they were just repeating material from the slides.

5. When a JUnit test fails, the programmer must change the program being tested.

   FALSE: The test itself might be incorrect and thus need to be changed.

6. Java generics shift the time at which programmers see errors from run-time to compile-time.

   TRUE: Without using generics, any instance of Object can be placed in a Collection. When the client accesses one of those instances, it can (try to) apply operations that may not apply to it. Generics eliminate this problem from run-time by adding type-checking to the client to make sure that the operations can be applied to instances of the subtype of Objects declared in the generic.

   Comments:
   • The question should have been more precise – a number of people interpreted this as relating to "all errors" (which is a poor interpretation from anything but a legalistic standpoint).

STUDENT NAME: ______________________________

Part II: Testing (22 total points)

1. (6 points) A basic implementation of `getGreeting` from `RandomHello` is:

```
/**
 * @return a random greeting from a list of
 * five different greetings.
 */
public static String getGreeting() {
  Random randomGenerator = new Random();
  return(greetings[randomGenerator.nextInt(5)]);
}
```

If you replace the last statement with

```
return(greetings[(randomGenerator.nextInt(5)+103312)%5])
```

where `%` is the modulus operator. Which of the tests (shown next and discussed in lecture), if any, fail for this implementation? Briefly justify.

- `testDoes_getGreeting_returnDefinedGreeting`
- `testDoes_getGreetingNeverReturnSomeGreeting`
- `test_UniformGreetingDistribution`

They all pass as they did given the original program. The reason is that the changed expression still returns uniformly within the range [0-4]. It is possible that the `test_UniformGreetingDistribution` will fail at times, but with no greater frequency that for the original program.

2. (8 points) Consider a simple form of random test generation, where the data passed to a method is selected randomly. For example, for `binarySearch` the length and contents of the array to be searched can be selected at random, as can the key to search for. Describe (in at most two sentences for each) two challenges that would arise from this simplistic method of testing.

There were a number of acceptable reasons, including:

● Hard to ensure coverage of edge/boundary cases.

● Hard for programmer to understand the results of the tests – both because the tests may not be easy to repeat (due to randomization) and also because the tests cannot clearly characterize the subdomain(s) being tested.

● Hard to know what the oracle is.

Comments:

● With respect to these answers, a few observations: (a) repeating tests can be handled by saving the random choices automatically; and (b) determining the oracle can be done by using another implementation of the specification.

● Concerns about problems with randomization such as "might not get sorted array" can be handled easily without complicating anything much – generate each successive value in the array by changing the range based on the previously generated value. Want a key that is found or not found? Flip a coin and then either pick a value in the array or one not in the array.

STUDENT NAME: ____

3. (8 points) Consider the following specification that is a variant of the binary search specification – basically, it now takes an unsorted array:

```
/**
 * @param data is an int array in which to search for the key
 * @param key the key to search for in data
 * @returns some i such that data[i] = key if such an i exists,
 *          otherwise -1
 */
public static int unsortedSearch(int[] data, int key)
```

As usual with a specification and black-box testing, you have no idea how this specification is implemented.

Write four black-box tests for this method. Each test should be intended to exercise a different possible subdomain, and you must briefly describe what that intended subdomain is for each test. For example, here is one test (no, you cannot repeat this one) and description:

```
@Test
public void testFoundKeyNearMiddle() {
  int[] td = {9, 32, -18, 99, 77};
  assertEquals(2,UnsortedSearch.unsortedSearch(td,-18));
}
```

Description: This is a basic test of the subdomain representing success in finding a key in the middle of the array.

Most common tests (not showing the actual tests):

1) Empty array

2) Key not found

3) Key found at first element

4) Key found at last element

5) Key found in multiple places

Comments:

● There were other tests that were correct, and identified the subdomain, but where it was not clear that there was a reason this subdomain would be interesting. (I gave credit for these in general.) But why would searching for a negative number or zero likely expose an error? Why would searching in a sorted array likely expose an error when searching in an unsorted works? (If this was tested, it should at least test on the same data as another test, except for the sorting of the array.) It's not that these aren't subdomains – it's that they have to be less important than (say) finding a key in multiple locations.

STUDENT NAME: ______________________________

Part III: Specifications (20 total points)

1. (10 points) Consider the following code (which compiles, runs, etc.):

```
static void uniquify(List<Integer> lst) {
  for (int i=0; i < lst.size()-1; i++)
    if (lst.get(i) == lst.get(i+1))
      lst.remove(i);
}
```

Fill-in the following specification for this code:

```
static void uniquify(List<Integer> lst)
  requires ???
  modifies ???
  effects  ???
  returns  ???
```

requires: true

modifies: lst

effects: two adjacent equal values are replaced by a single instance of that value;
that replaced single instance is not compared to the following value

returns: void

Comments:

● I took away some credit for requirements that the list be sorted. It is surely a legitimate precondition, but it's not necessary nor suggested by the code itself.

● For those who required a sorted list, it doesn't make a difference if it's sorted non-decreasing or non-increasing.

● Most people had it right that** modifies **lists the elements that are modified, while** effects **describes how they are modified.

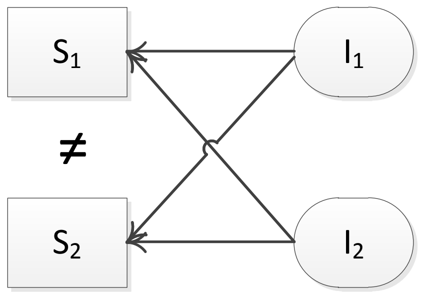
STUDENT NAME: ____________________________________

2. (8 points) Is the following situation possible (**S** means specification, **I** means implementation)?

   `S`$_1$ `is satisfied by I`$_1$ ∧ `S`$_2$ `is satisfied by I`$_1$ ∧
   `S`$_1$ `is satisfied by I`$_2$ ∧ `S`$_2$ `is satisfied by I`$_2$ ∧
   `S`$_1$ ≠ `S`$_2$

   Briefly justify your answer.

This is basically setting up the following situation:



It is possible: the simplest abstraction description is that this holds whenever S1 is stronger than S2 or vice versa. There are a number of concrete examples. One given a couple of times was (roughly) I1 and I2 as LinkedList and ArrayList, respectively, and S1 and S2 being List and Collection, respectively.

Comments:

● Some students gave answers of the form: "Because of x, y or z, nothing stops this situation from arising." This is different from justifying why it can happen.

● Some students seemed to confuse specifications with implementations.

STUDENT NAME: ______________________________

Part IV: ADTs (20 total points)

1. (12 points total) For each of the five following pairs of ADTs, select
   - **`T1< T2`**: **`T1`** is a true subtype of **`T2`**
   - **`T1> T2`**: **`T2`** is a true subtype of **`T1`**
   - **`T1` ≠ `T2`**: **`T1`** and **`T2`** are incomparable
   - **Other:** Cannot tell from this information

	< > ≠ Other? [Write answers in this column]	**T1**	**T2**
(a)	**3D points are a true subtype of 2D points**	**2-dimensional points** [Standard mutable 2D (x,y) points and operations]	**3-dimensional points** [Standard mutable 3D (x,y,z) points and operations]
(b)	**Integer is a true subtype of int**	**Java `Integer`**	**Java `int`**
(c)	**Not comparable**	**Java `String`**	**Java `Character[]`**
(d)	**Set<E> is a true subtype of AbstractionCollection<E>**	**`AbstractCollection<E>`** [`AbstractCollection<E>` provides a skeletal implementation of the `Collection` interface, to minimize the effort required to implement this interface.]	**`Set<E>`** [The `Set` interface places additional stipulations, beyond those inherited from the `Collection` interface, on the contracts of all constructors and on the contracts of the `add`, `equals` and `hashCode` methods.]

Comments:

● Bad question in the details. Sorry. For example, I couldn't tell whether some people were wrong or "just" got it all backwards. And the lack of clarity of the specifications I should have anticipated as lousy.

2. (8 points total) Consider the following Java code (which is legal and not intended to be surprising or confusing):

```
public class DayHour {
  private int day;
  private int hour;

  public DayHour(int d, int h) { // constructor
    day = d;
    hour = h;
  }

  public int getDay() {
    return day;
  }

  public int getHour() {
    return hour;
  }

  public String getHour12() {
    if (hour > 12)
      return hour + "AM";
    else
      return (hour - 12) + "PM";
  }
}
```

a. (4 points) What is the most appropriate representation invariant for the class?

0 <= hour <= 23
[I was going to include 0 <= day <= 31 – but the code simply doesn't convey this.]

Comments:
● A representation invariant is a constraint on the legal values the instance variables can take. It is not their type (yes, this constrains the values, but at the programming language level, not the class-implementation level).

b. (4 points) If the class has some representation exposure, briefly describe it. If not, briefly justify why not.
No. There are no ways in which the client can (mis)use any representation information. Indeed, the client cannot even extract any such information

Comments:
● A getter method doesn't imply rep exposure. The constructor can take ints and convert them to strings or something, and the getters can convert them back.

STUDENT NAME: ____

Part V: Miscellaneous (3 total points)

What part of the course so far has taken the most time "stuck" on something? A particular Java language construct or concept? A particular function of Eclipse or of JUnit? In other words, what have you spent the most time on for the least value in the course?

Everyone got credit for this.

The general breakdown was:

ADTs & AF/RI	16
svn	10
JUnit/testing	9
subtyping	7
java constructs	7
hw	5
design/patterns	3
eclipse	2

Some of the topics – like JUnit/testing – varied from "how to use JUnit" to "how to pick subdomains effectively"