Quiz (November 26, 2014)

Your name:________________________________________________________

Your Athena username:________________________________________________

You have 80 minutes to complete this quiz. It contains 13 pages (including this page) for a total of 100 points.

The quiz is closed-book and closed-notes, but you are allowed one two-sided page of notes.

Please check your copy to make sure that it is complete before you start. Turn in all pages, together, when you finish. Before you begin, write your name on the top of every page.

Please write neatly. No credit will be given if we cannot read what you write.

For questions which require you to choose your answer(s) from a list, do so clearly and unambiguously by circling the number(s) or entire answer(s). Do not use check marks, underlines, or other annotations – they will not be graded.

Good luck!

---

<table>
<thead>
<tr>
<th>Problem</th>
<th>Points</th>
<th>Grade</th>
<th>Grader</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Short Answer</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2: Operations</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3: Rep Invariant</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: Concurrency</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5: Specs 1</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6: Specs 2</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 1 (Short Answer) (24 points).

Circle all correct answers for the following questions.

(a) When implementing the Object contract (circle all that apply):

A. equals() must be reflexive, symmetric, and transitive
B. two objects with the same hashCode() must be equals()
C. two objects that are equals() must have the same hashCode()
D. having hashCode() return a constant for all objects breaks the Object contract

(b) Suppose you want to show that an equality operation is buggy because it isn’t transitive. What is the minimum number of objects you need to create for a counterexample to transitivity? (circle exactly one):

A. 1 object
B. 2 objects
C. 3 objects
D. all the objects in the type

(c) Which of the following strings will be matched completely by this regex: [a-z]( [A-Z]? [a-z0-9]+ )* (circle all that apply)

A. HelloWorld
B. mY?method+
C. flipImgLR
D. hellow0rld
E. callMeMaybe

(d) Suppose x is a local variable in a function. The statements x = 1; x = x * 2; are executed by 5 threads in parallel. Which of the following are possible values for x after all threads have executed these statements?

A. 0
B. 1
C. 2
D. 8
E. 32

(e) Which of the following statements are true about design patterns? (select all that apply)

A. The M in MVC stands for monitor pattern.
B. The Listener pattern is used to listen for messages from a java.io.Socket object.
C. The view tree pattern is an example of a recursive data type.
D. The Iterator pattern is an immutable data type.
E. The client-server pattern is an example of message passing concurrency.
(f) Suppose a mutable class MutableDate was declared as a subclass of an immutable class ImmutableDate. This would violate: (circle all that apply)

A. the rep invariant of ImmutableDate
B. the substitution principle
C. static checking
D. the precondition of ImmutableDate

(g) In well-written, defensive code, which of the following might be found in an assert statement? (circle all that apply)

A. a rep invariant
B. a postcondition
C. a precondition
D. an abstraction function

(h) The Java code `private List<Person> committee` demonstrates the concepts of: (circle all that apply)

A. representation independence
B. information hiding
C. static typing
D. view tree
E. abstract data type

(i) Given their definitions, which of the following immutable datatypes would be able to represent a pair of ints (and possibly other values as well)? Assume that null is forbidden as a value of the type.

A. \( T = A(\text{int}) + B(\text{A}, \text{A}) \)
B. \( T = D(\text{int}, \text{int}) \)
C. \( T = K(\text{int}) + L(\text{int}, T) \)
D. \( T = M() + N(\text{int}, T) \)
E. \( T = F(\text{int}, T) + G(\text{int}, T) \)

(j) Suppose you have to test Food.modify(), a mutator operation for a Food abstract datatype. What would you need in order to write black box test cases for this operation?

A. modify()’s preconditions and postconditions
B. Food’s rep invariant
C. modify()’s implementation code
D. the spec for at least one creator operation for Food
E. the spec for at least one observer operation for Food
F. the spec for at least one mutator operation for Food (other than modify)
G. the spec for at least one producer operation for Food
Problem 2 (Operations) (12 points).
Consider the following abstract data type:

```java
/**
 * Represents a student’s grades in a particular class.
 */
public interface StudentGradeRecord {

    // constants
    private static String ASSIGNMENT_PSET = "psets";
    private static String ASSIGNMENT_PROJECT = "project";
    private static String ASSIGNMENT_EXAM = "exam";

    // methods
    public void setStudentName(String name);
    public String getStudentName();

    public void setAssignmentGrade(String assignment, float grade);
    public float getAssignmentGrade(String assignment);

    public boolean removeAssignment(String assignment);
    public boolean addAssignment(String assignment);

    public static String[] getAssignmentTypes() {
        return new String[]{
            ASSIGNMENT_PSET, ASSIGNMENT_PROJECT, ASSIGNMENT_EXAM
        };
    }
}
```

For each of the following methods of the student record ADT, say what kind of operation it is, in the classification of ADT operations.

setStudentName ____________________________
getStudentName ____________________________
setAssignmentGrade _________________________
getAssignmentGrade _________________________
removeAssignment __________________________
addAssignment _____________________________
Problem 3 (Rep Invariant) (18 points).
Consider the following code:

```java
public class VirtualPet {
    public enum Condition {
        HUNGRY, TIRED, BATHROOM, CONTENT
    }

    private double happiness;
    private int discipline;
    private final Set<Condition> conditions;

    public VirtualPet(){
        happiness = 1;
        discipline = 0;
        conditions = new HashSet<Condition>(){
            conditions.add(CONTENT);
        }
    }

    /* @return the current happiness of this pet from 0 (very sad) to 1 (very happy). */
    public double happiness(){
        return happiness;
    }

    /* @return the current discipline of this pet from 0 (completely undisciplined) to 100 (fully obedient), in increments of 10, so that there are 11 possible discipline levels. */
    public int discipline(){
        return discipline;
    }

    /* @return the set of this pet's current condition, which could be any combination of the Conditions HUNGRY, TIRED, and BATHROOM, or CONTENT if it is none of those. */
    public Set<Condition> conditions() {
        return conditions;
    }
    ...
} // other operations not shown
```
(a) Write the rep invariant for this type. (As it might be written in a comment, not as executable Java code.)

(b) Suppose you are a client of VirtualPet, and you have a reference to a VirtualPet object in a variable pet. Using at most 3 statements of Java code, break the rep invariant of pet.

```java
VirtualPet pet = new VirtualPet();
```

(c) Define the discipline field in the rep and the corresponding implementation of the discipline() method, in such a way that discipline does not need to be mentioned in the rep invariant. The spec for the discipline() operation must remain unchanged.
Problem 4 (Concurrency) (12 points).

public interface Performance {

    // Invariant: no two users obtain paid tickets for the same seat.

    /**
     * @return seats currently unsold
     */
    public Set<Seat> availableSeats();

    /**
     * @param seat seat to purchase a ticket for, which must be unsold
     * @return a paid ticket for the given seat
     */
    public Ticket purchaseAvailableSeat(Seat seat);

    /**
     * @param seat seat to purchase a ticket for
     * @return a paid ticket for the given seat if it is unsold
     * @throws UnavailableException if the seat is already sold
     */
    public Ticket purchaseSeat(Seat seat) throws UnavailableException;

}

In this system, clients of Performance will use availableSeats to find available seats, then use the seat purchasing operation to buy tickets for one or more seats.
(a) Alyssa P. Hacker argues that the spec for purchaseAvailableSeat is not appropriate for use in a concurrent system. Louis Reasoner disagrees, saying that their implementation of Performance will use synchronized on all its methods and will not expose its rep. Alyssa replies, “Even if that’s true, Performance still won’t be usable with multiple threads.” Which is the best reason why not? (circle one)

A. concurrent clients will violate the invariant that no two users pay for the same seat
B. concurrent clients will violate purchaseAvailableSeat’s spec
C. concurrent clients will cause purchaseAvailableSeat to violate its spec
D. concurrent clients will deadlock trying to call purchaseAvailableSeat

(b) Ben continues developing the specification. He imagines that Performance defines:

```java
public interface Performance {

    // ...

    /**
     * @param seat desired seat
     * @return a paid ticket for the desired seat if it is unsold,
     * otherwise this method may return a paid ticket for a
     * different unsold seat
     * @throws UnavailableException may be thrown if the desired seat is already sold,
     * will be thrown if no seats are unsold
     */
    public Ticket purchaseOneSeat(Seat seat) throws UnavailableException;
}
```

This specification is... (circle all that apply)

A. operational
B. recursive
C. scoped
D. underdetermined
Problem 5 (Specs 1) (18 points).
To sort a List<E>, the sort() method requires the list’s element type E to implement this interface, which defines an ordering on the values of the type:

```java
interface Comparable<E> {
    /**
     * @param that element to compare this element with
     * @return < 0 if this is less than that in the ordering,
     * 0 if this is equivalent to that,
     * > 0 if this is greater than that.
     */
    public int compareTo(E that);
}
```

(a) Suppose String implements Comparable<String>, correctly following the spec above for the standard lexicographic ordering of strings (e.g., "a" is less than "ab"). For each of the following expressions, say whether it would produce a static error, a dynamic error, or no error. Circle exactly one choice.

"hello".compareTo("goodbye")
A. static error
B. dynamic error
C. no error

"".compareTo(""")
A. static error
B. dynamic error
C. no error

"".compareTo(null)
A. static error
B. dynamic error
C. no error

"5".compareTo(5)
A. static error
B. dynamic error
C. no error

compareTo("hello", "goodbye")
A. static error
B. dynamic error
C. no error
Suppose you’re implementing a new type Person that is supposed to implement Comparable so that it sorts people by age:

```java
class Person implements Comparable<Person> {
  private int age;
  ...
}
```

Below are proposed specs for the Person.compareTo() operation. For each of the following specs, choose whether it is stronger, weaker, equivalent, or incomparable to the spec above for Comparable<Person>.compareTo. Note that we are asking you to consider this spec against Comparable<Person>, which focuses on the Person type in particular.

```java
/**
 * @param that person to compare this person with
 * @return -1 if this person is younger than that person,
 * 0 if this person is the same age as that person,
 * 1 if this person is older than that person.
 */
public int compareTo(Person that);
```

A. is stronger than Comparable<Person>
B. is weaker than Comparable<Person>
C. is equivalent to Comparable<Person>
D. is incomparable to Comparable<Person>

```java
enum Difference { LESS, EQUAL, GREATER };

/**
 * @param that person to compare this person with
 * @return LESS if this person is younger than that person,
 * EQUAL if this person is the same age as that person,
 * GREATER if this person is older than that person.
 */
public Difference compareTo(Person that);
```

A. is stronger than Comparable<Person>
B. is weaker than Comparable<Person>
C. is equivalent to Comparable<Person>
D. is incomparable to Comparable<Person>
/**
 * @param that person to compare this person with
 * @return -1 if this person is younger than that person,
 *          1 if this person is greater than that person.
 * @throw SameAgeException if this person is the same age as that person
 */

public int compareTo(Person that);

A. is stronger than Comparable<Person>
B. is weaker than Comparable<Person>
C. is equivalent to Comparable<Person>
D. is incomparable to Comparable<Person>

/**
 * @param that person to compare this person with.
 *    this can’t be the same person as that.
 * @return < 0 if this is less than that in the ordering,
 *         0 if this is equivalent to that,
 *         > 0 if this is greater than that.
 */

public int compareTo(Person that);

A. is stronger than Comparable<Person>
B. is weaker than Comparable<Person>
C. is equivalent to Comparable<Person>
D. is incomparable to Comparable<Person>
Problem 6 (Specs 2) (16 points).
Suppose we have the following abstract data type:

```
/**
 * A Circle is an immutable type representing a circular area of the 2D plane,
 * including all the points inside the circle boundary plus the boundary circle itself.
 */
public class Circle {
    /** Make a Circle whose center is (x,y) and radius is r. r must be >= 0. */
    public Circle(double x, double y, double r) { ... }
    ...
}
```

Consider the following method specification:

```
/**
 * @param circles list of circles
 * @return number of pairs <circles[i],circles[j]> where i < j
 *        and the areas represented by circles[i] and circles[j]
 *        have a nonempty intersection.
 */
public static int countOverlaps(List<Circle> circles);
```

(a) Which of the following are partitions that would be sensible to use in testing countOverlaps? (circle all that apply)

A. circles is an empty list, a singleton list, or a list with more than one circle
B. circle with r=0, circle with r>0, circle with r<0
C. first half of circles list, second half of circles list
D. circle with center point in each quadrant, each axis, and origin

(b) Suppose countOverlaps() is in a library, and many clients are now calling it and depending on its spec. You decide you have to change the spec. Which of the following spec changes would require no changes in any of the clients, assuming they correctly follow the old spec? (circle all that apply)

A. the new spec is stronger than the old spec
B. the new spec is incomparable to the old spec
C. the new spec is weaker than the old spec
For each of the following spec changes, what would happen?

(c) /**
   * @param circles set of circles
   * @return number of pairs \( <c, d> \)
   * where \( c \) and \( d \) are in circles, \( c \) is not equal to \( d \),
   * and the areas represented by \( c \) and \( d \)
   * have a nonempty intersection.
   */
   public static int countOverlaps(Set<Circle> circles);

   (circle one best choice)
   A. clients will have static errors when recompiled
   B. clients may have dynamic errors when recompiled and run
   C. clients may compute wrong answers when recompiled and run
   D. clients will continue to work correctly

(d) /**
   * @param circles list of circles
   * @return number of circles in the list that have a nonempty
   * intersection with another circle in the list.
   */
   public static int countOverlaps(List<Circle> circles);

   (circle one best choice)
   A. clients will have static errors when recompiled
   B. clients may have dynamic errors when recompiled and run
   C. clients may compute wrong answers when recompiled and run
   D. clients will continue to work correctly

(e) /**
   * @param circles list of circles
   * @return number of pairs \( <circles[i],circles[j]> \) where \( i < j \)
   * and the areas represented by \( circles[i] \) and \( circles[j] \)
   * have a nonempty intersection.
   * @throws IllegalArgumentException if circles list has fewer
   *     than 2 elements.
   */
   public static int countOverlaps(List<Circle> circles);

   (circle one best choice)
   A. clients will have static errors when recompiled
   B. clients may have dynamic errors when recompiled and run
   C. clients may compute wrong answers when recompiled and run
   D. clients will continue to work correctly