Started on	Friday, 14 May 2021, 15:12
State	Finished
Completed on	Friday, 14 May 2021, 20:53
Time taken	5 hours 41 mins
Grade	**17.50** out of 150.00 (**12**%)

Question 1

Incorrect

Mark 0.00 out of 10.00

Consider the following problematic datatype:

```
        /** Represents an immutable right triangle. */
        class RightTriangle {
/*A*/       private double[] sides;

            // sides[0] and sides[1] are the two legs,
            // and sides[2] is the hypotenuse, so declare it to avoid having a
            // magic number in the code:
/*B*/       public static final int HYPOTENUSE = 2;

            /** Mak
             * @par
             * @param hypotenuse    the hypotenuse of the triangle.
 *C*         *         Requires hypotenuse^2 = legA^2 + legB^2
             *            (within the error tolerance of double arithmetic)
             */
            public RightTriangle(double legA, double legB, double hypotenuse) {
/*D*/           this.sides = new double[] { legA, legB, hypotenuse };
            }

            /** Get all the sides of the triangle.
             *  @return three-element array with the triangle's side lengths
             */
            public double[] getAllSides() {
                return sides;
            }

            /** @return length of the triangle's hypotenuse */
            public double getHypotenuse() {
                return sides[HYPOTENUSE];
            }

            /** @param factor to multiply the sides by
             *  @return a triangle made from this triangle by
             *  multiplies all side lengths by factor.
             */
            public RightTriangle scale(double factor) {
                return new RightTriangle (sides[0]*factor, sides[1]*factor, sides[2]*factor);
            }

            /** @return a regular triangle made from this triangle.
             *  A regular right triangle is one in which
             *  both legs have the same length.
             */
            public RightTriangle regularize() {
                double bigLeg = Math.max(side[0], side[1]);
                return new RightTriangle (bigLeg, bigLeg, side[2]);
            }

        }
```

Which of the following statements are true?

Select one:

○ a. The line marked /*A*/ is a problem for rep exposure because arrays are mutable.

○ b. The line marked /*B*/ is a problem for representation independence because it reveals how the sides array is organized.

● c. The line marked *C* is a problem because creator operations should not have preconditions.

○ d. The line marked /*D*/ is a problem because it puts legA, legB, and hypotenuse into the rep without doing a defensive copy first.

Your answer is incorrect.

The correct answer is: The line marked /*B*/ is a problem for representation independence because it reveals how the sides array is organized.

Question 2

Incorrect Mark 0.00 out of 10.00

Which of the following should **not** be known (visible and documented) to **the client** of an abstract data type?

Select one:

○ a. rep invariant

○ b. abstract value space

○ c. creators

● d. observers

Your answer is incorrect.

The correct answer is: rep invariant

Question 3

Incorrect Mark 0.00 out of 10.00

Which of the following should be known (visible and documented) to **the maintainer** of an abstract data type?

Select one:

○ a. all of the options

● b. abstract value space

○ c. creators

○ d. observers

○ e. abstraction function

○ f. rep

○ g. rep invariant

Your answer is incorrect.

The correct answer is: all of the options

Question 4

Question 4

Correct

Mark 10.00 out of 10.00

Suppose C is an abstract data type whose representation has two String fields:

```
class C {
    private String s;
    private String t;
    ...
}
```

Assuming you don't know anything about C's abstraction, which of the following might be part of a rep invariant for C?

Select one:

● a. s.length() == t.length()

○ b. s represents a set of characters

○ c. C's observers

○ d. s + t

Your answer is correct.

The correct answer is: s.length() == t.length()

Question 5

Incorrect

Mark 0.00 out of 10.00

Suppose we are implementing CharSet with the following rep:

```
public class CharSet {
    private String s;
    ...
}
```

But we neglect to write down the abstraction function (AF) and rep invariant (RI). Here are four possible AF/RI pairs, which were also mentione the lecture.

SortedRep:

```
// AF: {s[i] | 0 <= i < s.length()}
// RI: s[0] < s[1] < ... < s[s.length()-1]
```

SortedRangeRep:

```
// AF: represents the union of the ranges {s[i]...s[i+1]} for each adjacent pair of characters in s
// RI: s.length is even, and s[0] < s[1] < ... < s[s.length()-1]
```

NoRepeatsRep:

```
// AF: {s[i] | 0 <= i < s.length()}
// RI: s contains no character more than once
```

AnyRep:

```
// AF: {s[i] | 0 <= i < s.length()}
// RI: true
```

Which possible AF/RI pairs are consistent with this programmer's implementation of `add()`?

```
/**
 * Modifies this set by adding c to the set.
 * @param c character to add
 */
public void add(char c) {
    s = s + c;
}
```

Select one:

○ a. SortedRep

○ b. SortedRangeRep

● c. NoRepeatsRep

○ d. AnyRep

Your answer is incorrect.

The correct answer is: AnyRep

Question 6

Incorrect

Mark 0.00 out of 10.00

Suppose we are implementing CharSet with the following rep:

```
public class CharSet {
    private String s;
    ...
}
```

But we neglect to write down the abstraction function (AF) and rep invariant (RI). Here are four possible AF/RI pairs, which were also mentione the lecture.

SortedRep:

```
// AF: {s[i] | 0 <= i < s.length()}
// RI: s[0] < s[1] < ... < s[s.length()-1]
```

SortedRangeRep:

```
// AF: represents the union of the ranges {s[i]...s[i+1]} for each adjacent pair of characters in s
// RI: s.length is even, and s[0] < s[1] < ... < s[s.length()-1]
```

NoRepeatsRep:

```
// AF: {s[i] | 0 <= i < s.length()}
// RI: s contains no character more than once
```

AnyRep:

```
// AF: {s[i] | 0 <= i < s.length()}
// RI: true
```

Which possible AF/RI pairs are consistent with this programmer's implementation of `remove()`?

```
/**
 * Modifies this set by removing c, if found.
 * If c is not found in the set, has no effect.
 * @param c character to remove
 */
public void remove(char c) {
    int position = s.indexOf(c);
    if (position >= 0) {
        s = s.substring(0, position) + s.substring(position+1, s.length());
    }
}
```

Select one or more:

☐ i. SortedRep

☑ ii. SortedRangeRep

☐ iii. NoRepeatsRep

☐ iv. AnyRep

Your answer is incorrect.

The correct answers are: SortedRep, NoRepeatsRep

Question 7

Incorrect

Mark 0.00 out of 10.00

Suppose we are implementing CharSet with the following rep:

```
public class CharSet {
    private String s;
    ...
}
```

But we neglect to write down the abstraction function (AF) and rep invariant (RI). Here are four possible AF/RI pairs, which were also mentione the lecture.

SortedRep:

```
// AF: {s[i] | 0 <= i < s.length()}
// RI: s[0] < s[1] < ... < s[s.length()-1]
```

SortedRangeRep:

```
// AF: represents the union of the ranges {s[i]...s[i+1]} for each adjacent pair of characters in s
// RI: s.length is even, and s[0] < s[1] < ... < s[s.length()-1]
```

NoRepeatsRep:

```
// AF: {s[i] | 0 <= i < s.length()}
// RI: s contains no character more than once
```

AnyRep:

```
// AF: {s[i] | 0 <= i < s.length()}
// RI: true
```

Finally, which possible AF/RI pairs are consistent with this programmer's implementation of `contains()`?

```
/**
 * Test for membership.
 * @param c a character
 * @return true iff this set contains c
 */
public boolean contains(char c) {
    for (int i = 0; i < s.length(); i += 2) {
        char low = s.charAt(i);
        char high = s.charAt(i+1);
        if (low <= c && c <= high) {
            return true;
        }
    }
    return false;
}
```

Select one:

● a. SortedRep

○ b. SortedRangeRep

○ c. NoRepeatsRep

○ d. AnyRep

Your answer is incorrect.

The correct answer is: SortedRangeRep

Question 8

Incorrect Mark 0.00 out of 10.00

Consider this ADT:

```
public class Duration {
    private final int mins;
    private final int secs;
    // rep invariant:
    //    mins >= 0, secs >= 0
    // abstraction function:
    //    represents a span of time of mins minutes and secs seconds

    /** Make a duration lasting for m minutes and s seconds. */
    public Duration(int m, int s) {
        mins = m; secs = s;
    }
    /** @return length of this duration in seconds */
    public long getLength() {
        return mins*60 + secs;
    }
}
```

and these objects created from it:

```
Duration d1 = new Duration (1, 2);
Duration d2 = new Duration (1, 3);
Duration d3 = new Duration (0, 62);
Duration d4 = new Duration (1, 2);
```

Using the abstraction-function notion of equality, which of the following would be considered **equal to** to `d1`?

Select one or more:

□ i. d1

☑ ii. d2

□ iii. d3

□ iv. d4

Your answer is incorrect.

The correct answers are: d1, d3, d4

Question 9

Partially correct Mark 3.33 out of 10.00

Consider this ADT:

```
public class Duration {
    private final int mins;
    private final int secs;
    // rep invariant:
    //    mins >= 0, secs >= 0
    // abstraction function:
    //    represents a span of time of mins minutes and secs seconds

    /** Make a duration lasting for m minutes and s seconds. */
    public Duration(int m, int s) {
        mins = m; secs = s;
    }
    /** @return length of this duration in seconds */
    public long getLength() {
        return mins*60 + secs;
    }
}
```

and these objects created from it:

```
Duration d1 = new Duration (1, 2);
Duration d2 = new Duration (1, 3);
Duration d3 = new Duration (0, 62);
Duration d3 = new Duration (0, 62);
Duration d4 = new Duration (1, 2);
```

Using the observational notion of equality, which of the following would be considered **equal to** `d1`?

Select one or more:

□ i. d1

□ ii. d2

☑ iii. d3

□ iv. d4

Your answer is partially correct.

You have correctly selected 1.

The correct answers are: d1, d3, d4

Question 10

Incorrect

Mark 0.00 out of 10.00

Consider the latest implementation of `Duration` in the lecture:

```
public class Duration {
    private final int mins;
    private final int secs;
    // rep invariant:
    //    mins >= 0, secs >= 0
    // abstraction function:
    //    represents a span of time of mins minutes and secs seconds

    /** Make a duration lasting for m minutes and s seconds. */
    public Duration(int m, int s) {
        mins = m; secs = s;
    }
    /** @return length of this duration in seconds */
    public long getLength() {
        return mins*60 + secs;
    }

    private static final int CLOCK_SKEW = 5; // seconds

    @Override
    public boolean equals (Object thatObject) {
        if (!(thatObject instanceof Duration)) return false;
        Duration thatDuration = (Duration) thatObject;
        return Math.abs(this.getLength() - thatDuration.getLength()) <= CLOCK_SKEW;
    }
}
```

Suppose these Duration objects are created:

```
Duration d_0_60 = new Duration(0, 60);
Duration d_1_00 = new Duration(1, 0);
Duration d_0_57 = new Duration(0, 57);
Duration d_1_03 = new Duration(1, 3);
```

Which of the following expressions return **true**?

Select one or more:

☑ i. d_0_57.equals(d_1_03)

□ ii. d_0_60.equals(d_1_00)

□ iii. d_1_00.equals(d_0_60)

□ iv. d_1_00.equals(d_1_00)

□ v. d_0_57.equals(d_1_00)

□ vi. d_0_60.equals(d_1_03)

Your answer is incorrect.

The correct answers are: d_0_60.equals(d_1_00), d_1_00.equals(d_0_60), d_1_00.equals(d_1_00), d_0_57.equals(d_1_00), d_0_60.equals(d_1_03)

Question 11

Incorrect

Mark 0.00 out of 10.00

Consider the latest implementation of `Duration` in the lecture:

```
public class Duration {
    private final int mins;
    private final int secs;
    // rep invariant:
    //    mins >= 0, secs >= 0
    // abstraction function:
    //    represents a span of time of mins minutes and secs seconds

    /** Make a duration lasting for m minutes and s seconds. */
    public Duration(int m, int s) {
        mins = m; secs = s;
    }
    /** @return length of this duration in seconds */
    public long getLength() {
        return mins*60 + secs;
    }

    private static final int CLOCK_SKEW = 5; // seconds

    @Override
    public boolean equals (Object thatObject) {
        if (!(thatObject instanceof Duration)) return false;
        Duration thatDuration = (Duration) thatObject;
        return Math.abs(this.getLength() - thatDuration.getLength()) <= CLOCK_SKEW;
    }
}
```

Which properties of an equivalence relation are violated by this `equals()` method?

Select one:

○ a. recursivity

○ b. reflexivity

● c. sensitivity

○ d. symmetry

○ e. transitivity

Your answer is incorrect.

The correct answer is: transitivity

Question 12

Incorrect

Mark 0.00 out of 10.00

Suppose you want to show that an equality operation is buggy because it is **not** reflexive.

How many objects do you need for a counterexample to reflexivity?

Select one:

○ a. 0 objects

○ b. 1 object

○ c. 2 objects

● d. 3 objects

○ e. 4 objects

Your answer is incorrect.

The correct answer is: 1 object

Question 13

Partially correct

Mark 0.83 out of 10.00

Suppose Bag<E> is a mutable ADT representing what is often called a *multiset*, an unordered collection of objects where an object can occur m than once. It has the following operations:

```
/** make an empty bag */
public Bag<E>()

/** modify this bag by adding an occurence of e, and return this bag */
public Bag<E> add(E e)

/** modify this bag by removing an occurence of e (if any), and return this bag */
public Bag<E> remove(E e)

/** return number of times e occurs in this bag */
public int count(E e)
```

Suppose we run this code:

```
Bag<String> b1 = new Bag<>().add("a").add("b");
Bag<String> b2 = new Bag<>().add("a").add("b");
Bag<String> b3 = b1.remove("b");
Bag<String> b4 = new Bag<>().add("b").add("a"); // swap!
```

Which of the following expression is **true** ?

Select one or more:

□ i. b1.count("a") == 1

□ ii. b1.count("b") == 1

☑ iii. b2.count("a") == 1

□ iv. b2.count("b") == 1

☑ v. b3.count("a") == 1

☑ vi. b3.count("b") == 1

□ vii. b4.count("a") == 1

□ viii. b4.count("b") == 1

Your answer is partially correct.

You have correctly selected 2.

The correct answers are: b1.count("a") == 1, b2.count("a") == 1, b2.count("b") == 1, b3.count("a") == 1, b4.count("a") == 1, b4.count("b") == 1

Question 14

Incorrect

Mark 0.00 out of 10.00

Suppose Bag<E> is a mutable ADT representing what is often called a *multiset*, an unordered collection of objects where an object can occur m than once. It has the following operations:

```
/** make an empty bag */
public Bag<E>()

/** modify this bag by adding an occurence of e, and return this bag */
public Bag<E> add(E e)

/** modify this bag by removing an occurence of e (if any), and return this bag */
public Bag<E> remove(E e)

/** return number of times e occurs in this bag */
public int count(E e)
```

Suppose we run this code:

```
Bag<String> b1 = new Bag<>().add("a").add("b");
Bag<String> b2 = new Bag<>().add("a").add("b");
Bag<String> b3 = b1.remove("b");
Bag<String> b4 = new Bag<>().add("b").add("a"); // swap!
```

If Bag is implemented with **behavioral** equality, which of the following expression is **true** ?

Select one or more:

☑ i. b1.equals(b2)

□ ii. b1.equals(b3)

□ iii. b1.equals(b4)

☑ iv. b2.equals(b3)

□ v. b2.equals(b4)

□ vi. b3.equals(b1)

Your answer is incorrect.

The correct answers are: b1.equals(b3), b3.equals(b1)

Question 15

Partially correct

Mark 3.33 out of 10.00

Suppose Bag<E> is a mutable ADT representing what is often called a *multiset*, an unordered collection of objects where an object can occur m than once. It has the following operations:

```
/** make an empty bag */
public Bag<E>()

/** modify this bag by adding an occurence of e, and return this bag */
public Bag<E> add(E e)

/** modify this bag by removing an occurence of e (if any), and return this bag */
public Bag<E> remove(E e)

/** return number of times e occurs in this bag */
public int count(E e)
```

Suppose we run this code:

```
Bag<String> b1 = new Bag<>().add("a").add("b");
Bag<String> b2 = new Bag<>().add("a").add("b");
Bag<String> b3 = b1.remove("b");
Bag<String> b4 = new Bag<>().add("b").add("a"); // swap!
```

If Bag is implemented with **observational equality**, which of the following expression is **true** ?

Select one or more:

- □ i. b1.equals(b2)
- ☑ ii. b1.equals(b3)
- □ iii. b1.equals(b4)
- ☑ iv. b2.equals(b3)
- ☑ v. b2.equals(b4)
- □ vi. b3.equals(b1)

Your answer is partially correct.

You have correctly selected 2.

The correct answers are: b1.equals(b3),
b2.equals(b4),

b3.equals(b1)

Finish review

◀ Lab 11 Recording

Jump to...

Lab Exercise 11.1 Duration CO