Solutions to Quiz 1 (October 11, 2013)

Problem 1 (Short Answer) (25 points).
Choose all correct answers for the following questions.

(a) In order for elements of a type T to behave correctly in data structures like lists and sets, the equality operation for type T should be:

1. transitive
2. reflexive
3. invariant (over the duration of the program)
4. immutable
5. symmetric

Solution. 1, 2, 3, and 5. Reflexive/symmetric/transitive are the usual requirements of an equivalence, and invariance is necessary so that elements placed in a data structure like a hash table can be found again in the same place. An operation cannot be ‘immutable.’

(b) When a specification is strengthened:

1. fewer implementations satisfy it
2. more implementations satisfy it
3. fewer clients can use it
4. more clients can use it
5. none of the above

Solution. 1 and 4.

(c) 6.005 is about writing software that is:

1. safe from bugs
2. easy to understand
3. ready for change
4. efficient to run
5. formally provable

Solution. 1, 2, and 3.

(d) 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.

1. \( T = A(\text{int}) + B(A, A) \)
2. \( T = D(\text{int, int}) \)
3. \( T = K(\text{int}) + L(\text{int}, T) \)
4. \( T = M + N(\text{int}, T) \)
5. \( T = F(\text{int}, T) + G(\text{int}, T) \)

**Solution.** 1, 2, 3, 4. The recursion in 5 doesn’t have a base case, so can’t be instantiated at all.

(e) Suppose you have to test the `String.concat()` method. What would you need in order to write black box test cases?

1. the method’s preconditions and postconditions
2. the class’s rep invariant
3. the method’s implementation code
4. a way to create `String` objects
5. a way to compare `String` objects for equality

**Solution.** 1, 4, and 5. The rep invariant and implementation shouldn’t be used for black box test cases.

**Problem 2 (Abstract Data Types) (25 points).**

Consider the code on the following page.

(a) Classify each method as a creator, producer, mutator, or observer.

1. `public WordList(List<String> words)`
2. `public void unique()`
3. `public WordList getCapitalized()`
4. `public WordList merge(WordList that)`
5. `public Map<String, Integer> getFrequencies()`

**Solution.** C, M, P, M, O

(b) Which methods cause rep exposure?

1. `public WordList(List<String> words)`
2. `public void unique()`
3. `public WordList getCapitalized()`
4. `public WordList merge(WordList that)`
5. `public Map<String, Integer> getFrequencies()`

**Solution.** Y, N, N, N, Y

(c) Which methods violate the rep invariant, in some way other than rep exposure? (Remember that our preconditions and rep invariants implicitly require all object references to be non-null, unless stated otherwise.)

1. `public WordList(List<String> words)`
2. `public void unique()`
3. public WordList getCapitalized()
4. public WordList merge(WordList that)
5. public Map<String, Integer> getFrequencies()

Solution. N, Y, N, Y, N
/**
 * A WordList keeps track of a list of words.
 */

public class WordList {
    private List<String> words;
    private Map<String, Integer> frequencies = null;

    /* Rep Invariant:
     * - all elements of the words list are non-null
     * - if frequencies != null, then frequencies.get(w) is the
     *   number of times that w occurs in the words list. */

    public WordList(List<String> words) {
        this.words = words;
    }

    public void unique() {
        Set<String> uniqueWords = new HashSet<String>();
        for (String word : this.words) {
            uniqueWords.add(word);
        }
        this.words = new ArrayList<String>(uniqueWords);
    }

    public WordList getCapitalized() {
        List<String> newWords = new ArrayList<String>();
        for (String word : this.words) {
            newWords.add(word.toUpperCase());
        }
        return new WordList(newWords);
    }

    public WordList merge(WordList that) {
        for (String word : that.words) {
            this.words.add(word);
        }
        return this;
    }

    public Map<String, Integer> getFrequencies() {
        if (this.frequencies != null) {
            return this.frequencies;
        }

        this.frequencies = new HashMap<String, Integer>();
        for (String word : this.words) {
            if (!this.frequencies.containsKey(word)) {
                this.frequencies.put(word, 0);
            }
            int previousFrequency = this.frequencies.get(word);
            this.frequencies.put(word, previousFrequency + 1);
        }
        return this.frequencies;
    }
}
Problem 3 (Specs) (14 points).
Consider the following excerpt from a class:

```java
public class PartsList {
    ...
    private List<String> partNames;
    private List<Double> partCosts;
    /* REP INVARIANT */

    /**
     * requires: PRECONDITION
     * effects: POSTCONDITION
     */
    public double getPriceOfPart(String partName){
        int index = partNames.indexOf(partName);
        return partCosts.get(index);
    }
    ...
}
```

From the following list, choose the necessary parts for REP INVARIANT, PRECONDITION, and POSTCONDITION so that the code above is correct as written.

You should write:

RI next to each clause of the rep invariant,

PRE next to each clause of the precondition, and

POST next to each clause of the postcondition.

Note that you do not have to use all the items on the list below, and that we implicitly assume that object references (like `partName`) are non-null. You should choose only what’s necessary to make the code correct.

- `partNames` is immutable
- `partCosts` is immutable
- `partName` is a part name found in the parts list
- `partNames.size() == partCosts.size()`
- returns price of the part identified by `partName`
- throws `PartNotFoundException` if the part is not found in the parts list
- returns -1 if the part is not found in the parts list

Solution. We don’t need immutability here, nor do we have code to throw an exception or return a magic number; but we also don’t have code to check whether `partName` is in `partNames`.

REP INVARIANT: `partNames.size() == partCosts.size()`
PRECONDITION: `partName` is a part name found in the parts list
POSTCONDITION: returns price of the part identified by `partName`
Problem 4 (Grammars) (10 points).
Consider this grammar:

\[
\begin{align*}
S & ::= (B \ C)^* T \\
B & ::= M^+ \mid P B P \\
C & ::= B \mid E^+
\end{align*}
\]

(a) List the nonterminals of this grammar. (Warning: do not rely on capitalization to tell you whether something is a nonterminal.)

Solution. S, B, C

(b) Suppose S is the starting nonterminal. Which of the following are valid sentences in the language specified by the grammar? Circle the ones that are valid. (Whitespace is unimportant.)

1. T
2. M E T
3. P P P M P E T
4. M E M E T
5. P M M P P M M T
6. P M M P E M M M T
7. E E E E T
8. M M P M P T

Solution. Valid sentences are 1, 2, 4, 6, 8.

Problem 5 (Testing) (16 points).
Suppose we have an abstract data type for geometric shapes, defined as:

\[
\text{Shape} = \text{Hexagon} + \text{Pentagon} + \text{Square} + \text{Triangle}
\]

The datatype is immutable. It provides the following operation:

\[
\cdot \text{int getSides()} \rightarrow \text{return the number of sides}
\]

In addition, since Shapes are characterized only by their number of sides, equals is implemented so that all Shapes of the same type are equal.

Suppose we have a method with the following specification:

```java
/**
 * Count shapes in a given set that have more sides than the given shape.
 * @param shapes non-null set of shapes
 * @param shape non-null shape that must be included in "shapes"
 * @return the number of shapes in "shapes" that have more sides than "shape"
 */
public static int countMoreSides(Set<Shape> shapes, Shape shape)
```

(a) Given this specification, from the following list of abstract values, select all the valid test inputs for countMoreSides:

1. {}, null
Solutions to Quiz 1 (October 11, 2013)

2. \{Triangle, Hexagon\}, null
3. \{Triangle, Square\}, Hexagon
4. \{Triangle, Pentagon\}, Pentagon
5. \{Square, Pentagon\}, Triangle
6. \{Square, Hexagon\}, Square
7. \{Triangle, Pentagon, Hexagon\}, Pentagon

Solution. 1, 2, 3, and 5 violate the precondition, so we are left with 4, 6, and 7.

(b) Suppose we partition the spaces of both inputs:

- shapes is empty, shapes.size() == 1, or shapes.size() > 1
- shape has the fewest sides of any in shapes, has the most sides, or neither

We want to create an exhaustive test suite that covers the Cartesian product of these two partitions.

Using the same format for test inputs as in part (a), write clearly a minimum set of additional test inputs required to obtain the desired coverage. If a combination is covered by an input above, do not cover it again. There may be more than one correct answer.

Solution. The only valid addition is with set size 1, e.g. \{ Square \}, Square.

(c) Now, suppose we change the specification of countMoreSides. In particular:

```
...  
* @param shape non-null shape
...  
```

Using the same format again, write clearly three (3) additional test inputs we should add to our test suite that were not previously valid test inputs. Again, if a combination is covered by an input from part (a) or (b), do not cover it again. There may be more than one correct answer.

Solution. A clarification was issued during the quiz: “In problem 5, for part (c), since the spec has changed, you are permitted to extend the partitioning we used in part (b).”

Pick from, e.g.:

- \{ \}, Square – empty shapes
- \{ Square \}, Triangle – size 1 × fewer sides
- \{ Square \}, Hexagon – size 1 × more sides
- \{ Square, Pentagon \}, Triangle – size > 1 × fewer sides
- \{ Triangle, Square \}, Hexagon – size > 1 × more sides
- \{ Triangle, Hexagon \}, Square – size > 1 × (between fewest and most ∧ shape ∉ shapes)

Problem 6 (Representations) (10 points).

In the additive color model, all colors are represented as combinations of red, green, and blue – yellows, for example, are varying amounts of red plus green; and purples are red plus blue.

Suppose we define an ADT Color with the following operations:
• reddish: Color → boolean – return true iff this Color's greatest component is red
• greenish: Color → boolean – return true iff this Color's greatest component is green
• blueish: Color → boolean – return true iff this Color's greatest component is blue
• darker: Color → Color – return a Color \(^{1/10}\) as bright as this, or the darkest Color

In addition, the equals operation is supported.

Examine the implementations on the following page. Where code has been omitted with ..., you should assume the analogous and correct implementation.

If \(R_{\text{RGBColor}}\) and \(A_{\text{RGBColor}}\) are the representation and abstract spaces of \(\text{RGBColor}\), respectively, and \(R_{\text{PrimaryColor}}\) and \(A_{\text{PrimaryColor}}\) are the same for \(\text{PrimaryColor}\):

(a) Which single answer best characterizes the relationship:

1. \(R_{\text{RGBColor}} = R_{\text{PrimaryColor}}\)
2. \(R_{\text{RGBColor}} \subset R_{\text{PrimaryColor}}\)
3. \(R_{\text{RGBColor}} \supset R_{\text{PrimaryColor}}\)
4. \(R_{\text{RGBColor}} \cap R_{\text{PrimaryColor}} = \emptyset\)
5. \(R_{\text{RGBColor}} \cap R_{\text{PrimaryColor}} \neq \emptyset\)

Solution. 4. \(R_{\text{RGBColor}} \cap R_{\text{PrimaryColor}} = \emptyset\) since they are represented with different Java objects.

(b) Which single answer best characterizes the relationship:

1. \(A_{\text{RGBColor}} = A_{\text{PrimaryColor}}\)
2. \(A_{\text{RGBColor}} \subset A_{\text{PrimaryColor}}\)
3. \(A_{\text{RGBColor}} \supset A_{\text{PrimaryColor}}\)
4. \(A_{\text{RGBColor}} \cap A_{\text{PrimaryColor}} = \emptyset\)
5. \(A_{\text{RGBColor}} \cap A_{\text{PrimaryColor}} \neq \emptyset\)

Solution. 3. \(A_{\text{RGBColor}} \supset A_{\text{PrimaryColor}}\) since \(\text{PrimaryColor}\) can only represent reds, greens, and blues, while \(\text{RGBColor}\) can represent colors that combine red, green, and blue.

Unfortunately, these implementations suffer from representation exposure.

(e) Suppose a client is given only the following two \(\text{Color}\) instances, which are either both \(\text{RGBColor}\)s or both \(\text{PrimaryColor}\)s:

\[
\text{Color pureRed; // might be: new RGBColor(255, 0, 0) -or- new RedColor(255)}
\]
\[
\text{Color pureBlue; // might be: new RGBColor(0, 0, 255) -or- new BlueColor(255)}
\]

Without inspecting their runtime types (e.g. with \text{instanceof}) and without directly calling their constructors, write no more than three lines of code such that the result reveals whether \(\text{RGBColor}\)s or \(\text{PrimaryColor}\)s are in use:

Solution. With \(\text{PrimaryColor}\), generated colors only ever equal colors from their own type. So:

\[
\begin{align*}
\text{Color redBlack} &= \text{pureRed}.\text{darker()}.\text{darker()}.\text{darker()}; \\
\text{Color blueBlack} &= \text{pureBlue}.\text{darker()}.\text{darker()}.\text{darker()}; \\
\text{assert}(&\text{redBlack.equals(blueBlack)});
\end{align*}
\]
public interface Color {
    public boolean reddish();
    public boolean greenish();
    public boolean blueish();
    public Color darker();
}

public class RGBColor implements Color {
    private final int red, green, blue;

    /** ... @param red red value between 0 and 255, inclusive ... */
    RGBColor(int red, int green, int blue) {
        this.red = red; this.green = green; this.blue = blue;
    }

    public boolean reddish() { return red > green && red > blue; }
    public boolean greenish() { ... }
    public boolean blueish() { ... }

    public Color darker() { return new RGBColor(red/10, green/10, blue/10); }

    @Override
    public boolean equals(Object o) {
        if (! (o instanceof RGBColor)) { return false; }
        RGBColor other = (RGBColor)o;
        return red == other.red && green == other.green && blue == other.blue;
    }

    @Override
    public int hashCode() { ... }
}

public abstract class PrimaryColor implements Color {
    protected final int amount;

    /** ... @param amount color value between 0 and 255, inclusive ... */
    PrimaryColor(int amount) { this.amount = amount; }

    public boolean reddish() { return false; }
    public boolean greenish() { return false; }
    public boolean blueish() { return false; }
}

public class RedColor extends PrimaryColor {

    /** ... @param amount red value between 0 and 255, inclusive ... */
    RedColor(int amount) { super(amount); }

    @Override
    public boolean reddish() { return amount > 0; }

    public Color darker() { return new RedColor(amount/10); }

    @Override
    public boolean equals(Object o) {
        if (! (o instanceof RedColor)) { return false; }
        RedColor other = (RedColor)o;
        return amount == other.amount;
    }

    @Override
    public int hashCode() { ... }
}

public class GreenColor extends PrimaryColor {

    public class BlueColor extends PrimaryColor { ... }

    public class RGBColor extends PrimaryColor { ... }