Quiz 1 (October 11, 2013)

Your name:__________________________________________________________

Your Athena username:______________________________________________

You have 50 minutes to complete this quiz. It contains 11 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>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2: Abstract Data Types</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3: Specs</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: Grammars</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5: Testing</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6: Representations</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

(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

(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

(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 \texttt{null} is forbidden as a value of the type.

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

(e) Suppose you have to test the \texttt{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 \texttt{String} objects
5. a way to compare \texttt{Strings} for equality
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()

(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()

(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()
/**
 * 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
Problem 4 (Grammars) (10 points).
Consider this grammar:

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

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

(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 T
7. E E E E T
8. M M P M P T
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:

- \(\text{int \ getSides()}\) – return the number of sides

In addition, since \text{Shapes} are characterized only by their number of sides, \text{equals} is implemented so that all \text{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 \text{countMoreSides}:

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

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

- \text{shapes} is empty, \text{shapes.size()} == 1, or \text{shapes.size()} > 1
- \text{shape} has the fewest sides of any in \text{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.
(e) Now, suppose we change the specification of `countMoreSides`. In particular:

```java
...  
* @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.
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_RGBColor and A_RGBColor are the representation and abstract spaces of RGBColor, respectively, and R_PrimaryColor and A_PrimaryColor are the same for PrimaryColor:

(a) Which single answer best characterizes the relationship:

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

(b) Which single answer best characterizes the relationship:

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

Unfortunately, these implementations suffer from representation exposure.

(c) Suppose a client is given only the following two Color instances, which are either both RGBColors or both PrimaryColors:

Color pureRed; // might be: new RGBColor(255, 0, 0) -or- new RedColor(255)
Color pureBlue; // might be: new RGBColor(0, 0, 255) -or- new BlueColor(255)

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