MIT
6.031: Software Construction
Prof. Rob Miller & Max Goldman revised Tuesday 24th October, 2017, 15:03

# Quiz 1 (October 25, 2017)

**Your name:**______________________________

**Your Kerberos username:**______________________________

You have 50 minutes to complete this quiz. It contains 10 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 Kerberos username 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 letter(s) or entire answer(s). Do not use check marks, underlines, or other annotations – they will not be graded.

Good luck!

| Problem | Points |
|---|---|
| 1: Code review | 20 |
| 2: Equality | 16 |
| 3: Immutable ADT | 22 |
| 4: Mutable ADT | 18 |
| 5: Testing and Operations | 24 |
| Total | 100 |

For this quiz, a **shopping list** is an **ordered** list of **unique** items together with a **positive integer** quantity for each item (we will not consider units for quantities, *e.g.* 1 oz. vs. 1 lb. vs. 1 box, just a number).

For example, someone who plans to buy a gallon of milk, a small bunch of bananas, and a box of Klondike bars, might have this shopping list:

- Milk (1)
- Banana (3)
- Klondike bars (1)

Shopping lists are ordered, so the following is a **different** shopping list. It has the same items and quantities, but in a different order:

- Banana (3)
- Klondike bars (1)
- Milk (1)

This is **not** a shopping list:

- Banana (1)
- Banana (1)
- Banana (1)
- Klondike bars (1)
- Milk (1)

And neither is this:

- Milk (0)

---

Problems 1–3 refer to the code for `FixedShoppingList` on page 9, which you may detach.

A client uses `FixedShoppingList` like this:

```
FixedShoppingList s = new FixedShoppingList(Arrays.asList(
        "Banana",
        "Milk",
        "Banana",
        "Klondike bars",
        "Banana"));
System.out.println(s);
```

which produces:

```
Banana (3)
Milk (1)
Klondike bars (1)
```

Notice that the bananas have ended up first on this list, and the Klondike bars have ended up last.

**Problem 1** (Code review) (**20 points**).
For each of these code review comments on **`FixedShoppingList`**, circle AGREE or DISAGREE and explain why in one sentence.

**(a)** Line 21: in `toString()`, reduce the scope of `seen`.

AGREE / DISAGREE because:

**(b)** Lines 21–22: in `toString()`, make `seen` and `result` **`final`**.

AGREE / DISAGREE because:

**(c)** Line 3: declare `replist` as **`private`**.

AGREE / DISAGREE because:

**(d)** Line 3: declare `replist` as **`final`**.

AGREE / DISAGREE because:

**(e)** Line 6: in the constructor, just use `Collections.unmodifiableList(items)` instead of **`new`** `ArrayList<>(items)`.

AGREE / DISAGREE because:

**Problem 2** (Equality) (**16 points**).
Alyssa looks at **`FixedShoppingList`** and realizes that it should implement `equals()`.

She also sees that that just checking whether two `FixedShoppingList` instances have identical rep values will not be a correct implementation.

**(a)** Write two different rep values for `FixedShoppingList` that represent the same abstract value according to the code and specs given (*e.g.*, `toString`):

**(b)** And write clearly and completely their abstract value:

Suppose we implement `equals()` correctly, and now wish to implement `hashCode()`. Remember that if two objects are equal, then they must have the same hashcode. For each option below for the body of `hashCode()`, circle CORRECT or INCORRECT and explain why in one sentence.

**(c)**
```
return replist.hashCode();
```

CORRECT / INCORRECT because:

**(d)**
```
if (replist.isEmpty()) { return 0; }
else { return replist.get(0).hashCode(); }
```

CORRECT / INCORRECT because:

**Problem 3** (Immutable ADT) (**22 points**).

Reminder: please write neatly. Make sure each answer satisfies *all* the requirements of the question.

**(a)** Write an abstraction function for **`FixedShoppingList`** that works with the code and specs given.

**(b)** Write *requires* and *effects* to create a **stronger** spec for `withAnother()` (compared to the one in the code, lines 7–8) that is still satisfied by the given implementation.

In addition, your spec must allow this as a valid test case:

`[ Milk (1) ].withAnother("Cookies") = [ Milk (1), Cookies (1) ]`

where we check the items, their order, and their quantities in the result.

**(c)** Write *requires* and *effects* to create a **weaker** spec for `howMany()` (compared to the one in the code, lines 12–13) that is still satisfied by the given implementation.

In addition, your spec must allow this as a valid test case:

`[ Milk (1) ].howMany("Milk") = 1`

**Problem 4** (Mutable ADT) (**18 points**).

Problems 4–6 refer to the code for `MutableShoppingList` on page 10.
A client uses `MutableShoppingList` like this:

```
MutableShoppingList m = new MutableShoppingList();
m.another("Banana");
m.another("Milk");
m.another("Banana");
m.another("Klondike bars");
m.another("Banana");
System.out.println(m);
```

which produces:

```
Banana (3)
Klondike bars (1)
Milk (1)
```

Ben looks at `MutableShoppingList`. “Alyssa, how does this `TreeMap` work?”
“It keeps its keys sorted in alphabetical order.”
“OK,” says Ben.

**(a)** Write *requires* and *effects* to create a **stronger** spec for the **`MutableShoppingList()`** constructor (compared to the one in the code, lines 4–5) that is still satisfied by the given implementation.

**(b)** Write one statement to include in the rep invariant for `MutableShoppingList` that works with the code and specs given, and that 6.031 does **not** already assume implicitly.

**(c)** And write one piece of the rep invariant that 6.031 **does** already assume implicitly.

**Problem 5** (Testing and Operations) (**24 points**).

Alyssa looks at **`MutableShoppingList`**. "This `howMany` method has a bug! Did anyone test this thing?"

Devise a testing strategy for the `howMany()` operation:

**(a)** Write one good three-part partition on the method's implicit input only, ignoring `item`.

**(b)** Write one good three-part partition on the relationship between the method's two inputs.

`MutableShoppingList` uses `SortedMap` and `TreeMap`.

**(c)** In one sentence, what is the relationship between the **types** `SortedMap` and `TreeMap`?

**(d)** In one sentence, what is the relationship between the **specs** of `SortedMap` and `TreeMap`?

One **creator** ADT operation used in the `MutableShoppingList` code is the `TreeMap` constructor (line 6).

For each other kind of ADT operation in our taxonomy, give an example that is either **defined** or **used** in the `MutableShoppingList` code. Write the operation's type signature in function notation with its name, inputs, and outputs.

Creator: `TreeMap` : *void* → `TreeMap`

**(e)** Producer:

:  →

**(f)** Observer:

:  →

**(g)** Mutator:

:  →

```
SFB
ETU
RFC
```

You may detach this page. Write your username at the top, and hand in all pages when you leave.

**Immutable `FixedShoppingList`**

```java
/** Immutable shopping list. */
public class FixedShoppingList {

    List<String> replist;

    /** Make a new shopping list. */
    public FixedShoppingList(List<String> items) {
        replist = new ArrayList<>(items);
    }

    /** Make a new shopping list that includes itemToAdd. */
    public FixedShoppingList withAnother(String itemToAdd) {
        List<String> items = new ArrayList<>(replist);
        items.add(itemToAdd);
        return new FixedShoppingList(items);
    }

    /** Return the quantity of item on this list. */
    public int howMany(String item) {
        int count = 0;
        for (String listItem : replist) {
            if (item.equals(listItem)) { count++; }
        }
        return count;
    }

    /** Return a string where the items on this list appear in order,
     *  each on a separate line along with its quantity. */
    @Override public String toString() {
        Set<String> seen = new HashSet<String>();
        StringBuilder result = new StringBuilder();
        for (String item : replist) {
            if ( ! seen.contains(item)) {
                seen.add(item);
                result.append(item + " (" + howMany(item) + ")\n");
            }
        }
        return result.toString();
    }
}
```

You may detach this page. Write your username at the top, and hand in all pages when you leave.

**Mutable `MutableShoppingList`**

```java
/** Mutable shopping list. */
public class MutableShoppingList {

    SortedMap<String, Integer> itemCounts;

    /** Make a new shopping list. */
    public MutableShoppingList() {
        itemCounts = new TreeMap<>();
    }

    /** Add 1 of the given item to this list. */
    public void another(String item) {
        itemCounts.put(item, itemCounts.getOrDefault(item, 0) + 1);
    }

    /** Return the quantity of item on this list. */
    public int howMany(String item) {
        return itemCounts.get(item); // BUG!
    }

    /** Return a string where the items on this list appear in order,
     *  each on a separate line along with its quantity. */
    @Override public String toString() {
        StringBuilder result = new StringBuilder();
        for (String item : itemCounts.keySet()) {
            result.append(item + " (" + howMany(item) + ")\n");
        }
        return result.toString();
    }
}
```