Quiz 1 (March 22, 2019)

Your name:__________________________________________________________

Your Kerberos username:_____________________________________________

You have 50 minutes to complete this quiz. It contains 12 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!

<table>
<thead>
<tr>
<th>Problem</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: AF, RI, &amp; SRE</td>
<td>22</td>
</tr>
<tr>
<td>2: Code Review</td>
<td>18</td>
</tr>
<tr>
<td>3: Testing</td>
<td>20</td>
</tr>
<tr>
<td>4: Specifications</td>
<td>18</td>
</tr>
<tr>
<td>5: Immutability</td>
<td>22</td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
</tr>
</tbody>
</table>
For this quiz, a thermostat program describes the settings of a simple temperature control system that has a mode and an association between blocks of time and the goal temperatures to try to maintain during those times.

The system mode is either heat or cool. We are ignoring other possible common settings like auto or off.

The granularity of our thermostat programs is 30-minute blocks starting on the hour and half-hour: 12 midnight, 12:30 am, 1:00 am, 1:30 am, etc. Each 30-minute block has an associated goal temperature which the system will try to maintain by using one of heating or cooling, depending on the mode.

For example, here is a possible thermostat program for a home during winter, where the times indicate the start of each 30-minute block:

**Mode: heat**

- 12 midnight
- 6:00 am
  - 68 °F (warmer for waking up, breakfast)
- 9:00 am
  - 62 °F (cooler while everyone is at work/school)
- 12 noon
  - 68 °F (warmer for dinner, going to sleep)
- 5:00 pm
  - 68 °F (warmer for dinner, going to sleep)
- 10:00 pm
  - 65 °F (overnight temperature, sleeping)

Problems 1–4 in this quiz refer to the code forMutableProgram starting on page 10. MutableProgram allows the client to define rules that set the goal temperature until the next rule takes effect.

Note that the code uses “?:” expressions:

```
predicate ? value-if-true : value-if-false
```

This is called the ternary conditional operator, and it is a shorthand if-else statement. The code also uses NavigableMap, a Map with ordered keys and additional operations for finding keys in the map. Abbreviated specs for some NavigableMap operations are provided in the code where they are first used.

The MutableProgram API uses 24-hour time and degrees Fahrenheit.

For example, to create the thermostat program above, we can use four rules:

```
MutableProgram winter = new MutableProgram(Mode.HEAT);
winter.addRule( 6, 30, 68); // breakfast
winter.addRule( 9,  0, 62); // work/school
winter.addRule(17, 30, 68); // dinner
winter.addRule(22, 30, 65); // overnight (rule applies through midnight // into the early AM)
```

Problem 5 refers to the code for ImmutableProgram on page 12. You may detach both code pages.
Problem 1 (AF, RI, & SRE) (22 points).
Based on the MutableProgram code starting on page 10...

(a) Draw a snapshot diagram for: MutableProgram summer = new MutableProgram(Mode.COOL);

(b) What thermostat program value do you get by evaluating MutableProgram’s abstraction function on the rep of summer? Be complete and precise. (Do not write the AF.)

(c) Write a rep invariant for MutableProgram that is as strong as possible, but not stronger than the provided code allows. (You do not need to state 6.031 assumptions.)

(d) Is MutableProgram safe from rep exposure? Circle either SAFE or EXPOSED; and if EXPOSED, identify why.

SAFE / EXPOSED

and if EXPOSED, because...
Problem 2 (Code Review) (18 points).
How can we improve MutableProgram?

(a) Alyssa P. Hacker is reading the code (starting from the top of page 10), and when she reaches line 54 (on page 11) she says: “I think this needs a helper function to DRY it up!” Circle AGREE or DISAGREE, and explain in one clear sentence.

AGREE / DISAGREE because...

(b) Ben Bitdiddle looks at removeRule and says: “this method would be better if we assert that minute is 0 or 30.” Circle AGREE or DISAGREE, and explain in one clear sentence.

AGREE / DISAGREE because...

(c) “And we should also assert that tempRules is not empty.” Circle AGREE or DISAGREE, and explain in one clear sentence.

AGREE / DISAGREE because...

(d) Then Alyssa says: “I think we should refactor both addRule and removeRule to make the code more SFB.” Write your best one-sentence suggestion for changing the arguments of those methods that primarily and directly addresses SFB.

...
Problem 3 (Testing) (20 points).
Start constructing a testing strategy for MutableProgram's `goalTemperature(..)` operation.

(a) What kind of ADT operation is `goalTemperature`?

(b) Write the type signature for the method (inputs on the left, output on the right):

\[
\text{goalTemperature : } \quad \rightarrow \quad \\
\]

For the questions below, write exactly one partitioning for each question.

Make sure you are testing the spec. For example, do not partition the rep.

(c) Write one correct, useful, 2- or 3-part partitioning of just the `minute` input, without reference to any other inputs or outputs:

(d) Write one correct, useful, 2- or 3-part partitioning of just the implicit input, without reference to any other inputs or outputs:

(e) Write one correct, useful, 2-to-4-part partitioning of all the inputs together. This partitioning should relate all the inputs, and should be substantially different from the product of (c) and (d):

\[
\]
Problem 4 (Specifications) (18 points).

Ben Bitdiddle suggests a different spec and implementation for `MutableProgram`'s `removeRule()`:

```java
/**
 * Modify this program, which must have a rule that starts at the given time,
 * to remove that rule.
 * @param hour must be 0 <= hour < 24
 * @param minute must be 0 or 30
 */
public void removeRule(int hour, int minute) {
    Integer removedTemp = tempRules.remove(hour * 2 + (minute < 30 ? 0 : 1));
    // remove: removes the mapping for a key from this map if it is
    // present; and returns the value previously associated
    // with the key, or null if the map contained no mapping
    // for the key
    if (removedTemp == null) {
        throw new IllegalArgumentException("no rule at given time");
    }
}
```

(a) Circle the relationship between the **original spec** and **Ben’s spec**, and explain why, referring to both pre- and postconditions:

- the original spec is STRONGER / WEAKER / INCOMPARABLE

(b) Circle the relationship between the **original implementation** and **Ben’s spec**, and explain why, again referring to both pre- and postconditions:

- SATISFIES / DOES NOT SATISFY

(c) Circle the relationship between the **Ben’s implementation** and the **original spec**, and explain why, referring to both pre- and postconditions:

- SATISFIES / DOES NOT SATISFY
Problem 5 (Immutability) (22 points).
This problem refers to the code for ImmutableProgram on page 12.

(a) ImmutableProgram has a very serious bug where withGoal is incorrect. What is the one-word name of the problem, and what is its effect in withGoal?

Name:
withGoal...

(b) In this particular code, one very small improvement would identify this bug at compile time. What is the improvement, and how would it identify the bug at compile time? (For partial credit, suggest an improvement that identifies the bug at runtime.)

Improvement:
which...

(c) Assume we fix the bug in withGoal. The code has no specs, but to the best of your ability, write an abstraction function for ImmutableProgram that works with the provided code and our definition of a thermostat program.

AF(settings) =

Problem continues on next page →
Defining equality for ImmutableProgram, we write the following clever (?) hashCode implementation:

```java
@override public int hashCode() {
    int randIndex = new Random().nextInt(settings.length);
    // nextInt: return a pseudorandom int value between 0 (inclusive)
    // and the given value (exclusive)
    return settings[randIndex];
}
```

Assume our equals(…) implementation is correct.

(d) Why does the implementation above not satisfy the spec of hashCode? Give an example of how a client could observe a hashCode spec violation:
Quiz 1 (March 22, 2019)

Your Kerberos username:

SFB
ETU
RFC
public enum Mode { COOL, HEAT }

/**
 ** Mutable thermostat program that is defined using rules. */
public class MutableProgram {

    private Mode mode;
    private final NavigableMap<Integer,Integer> tempRules;

    /**
     * ...
     */
    public MutableProgram(Mode mode) {
        this.mode = mode;
        this.tempRules = new TreeMap<>();
    }

    /**
     * @return the system mode
     */
    public Mode mode() {
        return mode;
    }

    /**
     * @param hour must be 0 <= hour < 24
     * @param minute must be 0 or 30
     * @return the goal temperature for the given time in degrees Fahrenheit
     * according to this program’s rules
     */
    public int goalTemperature(int hour, int minute) {
        if (tempRules.isEmpty()) {
            return 68;
        }
        Integer ruleTime = tempRules.floorKey(hour * 2 + (minute < 30 ? 0 : 1));
        // floorKey: returns the greatest key less than or equal to the
        // given key, or null if there is no such key
        if (ruleTime != null) {
            return tempRules.get(ruleTime);
        }
        return tempRules.get(tempRules.lastKey());
        // lastKey: returns the last (highest) key currently in the map,
        // or throws NoSuchElementExce}
```java
/**
 * ...
 */
public void switchMode() {
    mode = mode == Mode.COOL ? Mode.HEAT : Mode.COOL;
}

/**
 * Modify this program to add a rule starting at the given time (or replacing
 * the rule starting at the given time, if any) with goal temperature ‘temp’.
 * The latest-time rule carries over through midnight to the next day.
 * @param hour must be 0 <= hour < 24
 * @param minute must be 0 or 30
 * @param temp goal temperature in degrees Fahrenheit
 */
public void addRule(int hour, int minute, int temp) {
    tempRules.put(hour * 2 + (minute < 30 ? 0 : 1), temp);
}

/**
 * Modify this program to remove the rule (if any) that currently determines
 * the goal temperature for the given time.
 * The latest-time rule carries over through midnight to the next day.
 * @param hour must be 0 <= hour < 24
 * @param minute must be 0 or 30
 */
public void removeRule(int hour, int minute) {
    if (tempRules.isEmpty()) {
        return;
    }
    Integer ruleTime = tempRules.floorKey(hour * 2 + (minute < 30 ? 0 : 1));
    if (ruleTime != null) {
        tempRules.remove(ruleTime);
        return;
    }
    tempRules.remove(tempRules.lastKey());
}
```
class ImmutableProgram {

    public static final List<Mode> MODES = List.of(Mode.COOL, Mode.HEAT);

    private int[] settings;

    public ImmutableProgram(Mode mode) {
        settings = new int[1 + 24*2];
        Arrays.fill(settings, 68);
        // fill: assigns the given value to each element of the array
        settings[0] = MODES.indexOf(mode);
    }

    public Mode mode() {
        return MODES.get(settings[0]);
    }

    public int goalTemperature(int hour, int minute) {
        return settings[1 + hour * 2 + (minute < 30 ? 0 : 1)];
    }

    public ImmutableProgram withGoal(int hour, int minute, int temp) {
        ImmutableProgram updated = new ImmutableProgram(mode());
        updated.settings = settings;
        updated.settings[1 + hour * 2 + (minute < 30 ? 0 : 1)] = temp;
        return updated;
    }
}