MIT
6.031: Software Construction
Max Goldman and Prof. Armando Solar-Lezama

revised Tuesday 19th March, 2019, 23:26

Solutions to Quiz 1 (March 22, 2019)

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

$$\left.\begin{array}{l}\text{12 midnight}\\ \text{...}\\ \text{6:00 am}\end{array}\right\}\ \mathbf{65\ {}^\circ F}\ \text{(overnight temperature, sleeping)}$$

$$\left.\begin{array}{l}\text{6:30 am}\\ \text{...}\\ \text{8:30 am}\end{array}\right\}\ \mathbf{68\ {}^\circ F}\ \text{(warmer for waking up, breakfast)}$$

$$\left.\begin{array}{l}\text{9:00 am}\\ \text{...}\\ \text{12 noon}\\ \text{...}\\ \text{5:00 pm}\end{array}\right\}\ \mathbf{62\ {}^\circ F}\ \text{(cooler while everyone is at work/school)}$$

$$\left.\begin{array}{l}\text{5:30 pm}\\ \text{...}\\ \text{10:00 pm}\end{array}\right\}\ \mathbf{68\ {}^\circ F}\ \text{(warmer for dinner, going to sleep)}$$

$$\left.\begin{array}{l}\text{10:30 pm}\\ \text{...}\\ \text{11:30 pm}\end{array}\right\}\ \mathbf{65\ {}^\circ F}\ \text{(overnight temperature, sleeping)}$$

Problems 1–4 in this quiz refer to the code for `MutableProgram` starting on page 5. `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 7. You may detach both code pages.

Problem 1 (AF, RI, & SRE) (**22 points**).
Based on the **`MutableProgram` code starting on page 5**...

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

Solution. `summer -> (MutableProgram mode -> ((Mode COOL)) tempRules => (TreeMap))`

■

(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.)

Solution. The program with mode *cool* and goal temperature 68 °F at all times. ■

(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.)

Solution. All keys in `tempRules` are between 0 (inclusive) and 48 (exclusive). ■

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

Solution. SAFE. ■

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 5), and when she reaches line 54 (on page 7) she says: "I think this needs a helper function to DRY it up!" Circle AGREE or DISAGREE, and explain in one clear sentence.

Solution. AGREE, create a helper function to compute the map key. ■

(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.

Solution. AGREE, assert the precondition to fail fast. ■

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

Solution. DISAGREE, the empty map is valid. ■

(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.

Solution. Introduce a type for time blocks that captures the preconditions as invariants. ■

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

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

Solution. Observer ■

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

`goalTemperature :` $\rightarrow$

Solution. `MutableProgram` $\times$ `int` $\times$ `int` $\rightarrow$ `int` ■

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*:

Solution. `minute` is 0 or 30 ■

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

Solution. E.g., `this` has 0, 1, or more than 1 rule ■

(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)**:

Solution. E.g., the goal temperature at the given time is determined:

by default, by an earlier rule, by a rule at that time, or by a later rule that continues through midnight ■

Problem 4 (Specifications) (**18 points**).
Ben Bitdiddle suggests a different spec and implementation for `MutableProgram`'s `removeRule(..)`:

```
/**
 * 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*.

Solution. The original spec is STRONGER than Ben's spec:

it has a weaker precondition, and for inputs that satisfy Ben's stronger precondition, the postcondition is the same. ■

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

Solution. The original implementation SATISFIES Ben's spec:

for inputs satisfying Ben's precondition, it satisfies Ben's postcondition. ■

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

Solution. Ben's implementation DOES NOT SATISFY the original spec:

that spec's weaker precondition allows inputs on which Ben's implementation throws an exception and does not satisfy that spec's postcondition. ■

Problem 5 (Immutability) (**22 points**).
This problem refers to the code for **`ImmutableProgram` on page 7**.

(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`?

Solution. Aliasing:

`withGoal` creates a new instance that shares the rep of `this`, and mutates that rep, thereby mutating `this`. ■

(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.)

Solution. Make `settings` `final`: its reassignment as an alias would be illegal. ■

(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*.

Solution. Represents the thermostat program with mode `MODES.get(settings[0])`, and goal temperature `settings[i+1]` °F for the $\mathtt{i}^{th}$ half-hour block after midnight, $0 \leq \mathtt{i} < 48$. ■

Problem continues on next page →

Defining equality for `ImmutableProgram`, we write the following clever (?) `hashCode` implementation:

```
    @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:

Solution. This `hashCode` picks at random a different part of the rep to examine every call.

It is not consistent over time, so multiple calls to the same object will return different values. And it is not consistent between instances, so calls to equal objects will return different values. ■

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

```
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 NoSuchElementException if the map is empty
    }

    /**
     * ...
     */
    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
     */
```

```java
    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());
    }
}
```

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

```java
/** Immutable thermostat program. */
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;
    }
}
```