Programming Structures and Abstractions (CS 2334)
Lab 3: Searching and Sorting

September 15, 2010

Due: Friday, September 17, 2010, 11:29am

Group members (same as for your project):

Objectives

By the end of this lab, you should be able to:

1. analyze the class structure of an existing java program using UML diagrams,
2. extract and store sensor data from the Finch,
3. employ abstract classes to provide generic programming functionality, and
4. search the Finch "data streams" for key values.

Problem Context

Weather monitoring systems, spacecrafts and boarder security systems share the common problems of extracting data from a sensor "package" (collection of different sensors), storing this information in some organized fashion, and then later manipulating this information to find key patterns. In this project, we will be "logging" data obtained from our Finch

and then searching the set of samples for occurrences of minimum, maximum, and median values.
Specifically:

- We will take a *sample* of data at regular (100 ms) intervals for 10 seconds (so, 100 samples).
- Each sample is a *tuple* that contains the values from the light, acceleration, obstacle and temperature sensors.
- Both the light and obstacle sensors contain two channels: one on the left hand side of the head and the other on the right hand side. Light is measured in an arbitrary unit (brighter is encoded with larger numbers). The obstacle sensor is boolean.
- The acceleration sensor contains 3 channels: X, Y and Z and measures acceleration in units of g. Note that the sensor will also capture the acceleration due to gravity, so if the Finch is not moving, then this sensor will tell you which direction is "down".

We will be reporting the sample that corresponds to the minimum, maximum and median sample in the set. However, we are faced with the question of: "how do we find the minimum, maximum and median of a sample?" One way to accomplish this is to first sort our samples (e.g., using our **sort()** method from the book that relies on classes that implement the **Comparable** interface). Then, we can take the first, last and middle samples from this sorted list as the minimum, maximum and median samples.

But: what does it mean to sort() our samples given that each sample is composed of multiple variables? How do we tell which one is *larger* than another? What we want to be able to say is: sort on the Z component of the acceleration vector or sort on the brightness of the right hand side light sensor. How can we do this and yet still maintain a generic solution to problem of sorting?

Our solution is to define a new interface, **Comparable2**, that requires the implementing class to provide the following method:

```
public int compareTo2(Object obj, VariableType var);
```

This method call is similar to compareTo(), except that it adds a second argument that encodes *which* variable (or combination of variables) that we should be comparing (and ultimately sorting on).

Milestones

Milestone 1: Analyze a Class Structure

From the class web page, go to the **lab3** directory. Download the **Lab3.zip** file and import it into Eclipse as a new project. This project file includes several java class and interface implementations, as well as the associated javadoc.

Using the javadoc documentation and the source files, draw a UML diagram that captures the details of the individual classes and their relationships. For a nice summary of UML notation see:
http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/

In your UML diagram, you should:

- Detail the interfaces: include the interface names and the abstract method prototypes. Preface interface names with the text: "<< interface >>" and preface abstract methods with "<< abstract >>" (since it is hard to hand write in *italics*).
- Detail the classes: include the class name, variable definitions and method prototypes. Use the notation above as necessary.
- Detail the relationships between the classes and interfaces (see your book or the URL above for notational hints). In particular, you should express:
  - class/interface inheritance, and
  - dependency of classes on interfaces and other classes.

Milestone 2

Examine the implementation of **sort()** in sensorDriver.java. Describe the essential differences between this implementation and the one in your book (available in handout form in the class).

Milestone 3

The implementation of the program is missing one component: the full implementation of FinchSensor.compareTo2(). Add the missing code:

- compareTo2() should return one of three values: 1, 0 or -1. The return value depends on the ordering of *this* and *sensor* (see the code).
- The instance variable to compare (to determine the ordering) of *this* and *sensor* is encoded by the local variable *var* (which is a SensorType in our case). Note that in one case, SensorType.ACCEL_MAG, the comparison is based on a combination of multiple instance variables.

For example, suppose that:

```
this.acceleration[2] has a value of 0.9
sensor.acceleration[2] has a value of 0.2
var is equal to SensorType.ACCEL_Z
```

then:

```
the return value will be 1.
```

This is the case because var == SensorType.ACCEL_Z tells us that the 3rd element of the acceleration vectors should be compared.

Hints:

- You can use the **var.equals()** method (provided by the Object class) to compare two SensorTypes
- We have supplied the helper methods:

```
private int compareTo(int a, int b);

private int compareTo(double a, double b);

private double arrayMagnitude(double[] vec);
```

After you have compiled your program, you should execute it. First, attach the Finch to your computer. Then, start your program. Data recording starts when the nose turns red. Before advancing to the next milestone, you should convince yourself that your program is working properly (think about what would convince you). You may make minor modifications to main() to perform these checks.

Milestone 4

Return the main() method to its original state.

With your Finch sitting wheels down, execute your program. While recording, cover the light sensors and accelerate the Finch upwards. Then: accelerate the Finch downwards. Uncover the light sensors and allow the program to complete.

After recording is complete, the program will report the raw values in order. In addition, the program will find the samples with the min, max and median ACCEL_MAG values and print out the **entire** sample. Write down these latter numbers (not the 100 raw samples). Given the movement of your Finch, what interesting correlations do you see? (some channels will show interesting correlations, while others will not)

Milestone 5

With your Finch sitting wheels down, execute your program. Cover the light sensors. Then, cover the obstacle detectors for a couple seconds. Finally, uncover the obstacle detectors and then unover the light sensors.

Again, write down the values for the samples of the min, max and median channel values. What interesting correlations do you see?

What is Due ...

All materials are due: Friday, September 17, 2010, 11:29am

Hand in the following:

- a copy of your UML diagram (hard or soft copy will do),
- a copy of this handout containing the provided answers, and
- an electronic copy of your modified code (to D2L).

NOTE: ONLY HAND IN ONE COPY PER GROUP. The dropbox is now configured so that you can see the submissions of your lab partner.

Demonstration:

- See one of the TAs.
- Demonstrate your program given the requested sequence of movements of the Finch.