CS 2334
Project 1: Reading Data from Files

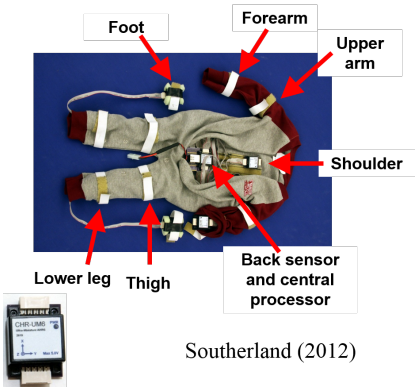
September 6, 2017

Due: 1:29 pm on Sept 20, 2017

Introduction

The Self-Initiated Prone Progression Crawler (SIPPC) is an assistive robot that aids infants at risk for Cerebral Palsy as they learn how to crawl. The robotic platform supports an infant's weight and amplifies her crawling-like movements by carrying her across the floor in the indicated direction. This interaction serves to encourage the infant to continue practicing crawling movements and allows the infant to actively explore her immediate environment.

SIPPC Assistive Robot



Kinematic Capture Suit

One sensor subsystem worn by the infant is a *kinematic capture suit*. Small inertial measurement units (IMUs) are strapped to the infant at twelve different points.

These sensors, among other information, provide an estimate of the orientation of key body segments. Combined with a skeletal model of the infant, we are able to infer in real time the position of particular points on the infant, including the wrists and feet. This position is recorded in the units of meters and is relative to the small of the infant's back. Furthermore, the coordinate frame is as follows: $+X$ runs from the small of the infant's back to the head, $+Y$ points to the right of the infant, and $+Z$ emanates from the back of the infant (hence, we have a right-handed coordinate frame).

For this project, you will focus on the data from a single infant. The data are organized into 5-minute trials, with samples of the left and right wrist positions taken every 20 ms (50 Hz). The data are organized in a comma separated file (CSV) format with one time sample on each row of the file.

We have provided data for two trials (one file for each trial). Your job is to read in the data files, parse the data, create appropriate objects from these data, and summarize the data using maximum, minimum, and average mathematical functions. You will also continue to expand your use of unit tests beyond the lab and ensure that your parsing and mathematical functions are correct. More details are below in the *Project Components* section.

Note: due to unforeseen circumstances, such as temporary power loss or sensor errors, sometimes the data are unavailable for one or more time steps. These situations are represented in the CSV files using the String "NaN" (short for *Not A Number*). Make sure you don't include these values in your statistical summaries.

Learning Objectives

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

1. parse structured data from a file,
2. create objects using data parsed from a file,
3. use mathematical transformations on Strings,
4. implement mathematical functions in code,
5. employ unit testing to ensure that different pieces of your code are functioning properly, and
6. provide proper documentation in Javadoc format.

Proper Academic Conduct

This project is to be done in the groups of two that we have assigned. You are to work together to design the data structures and solution, and to implement and test this design. Your group will turn in a single copy of your solution. Do not look at or discuss solutions with anyone other than the instructor, TAs or your assigned team member. Do not copy or look at specific solutions from the net.

Strategies for Success

- We encourage you to work closely with your other team member, meeting in person when possible.
- Start this project early, as it cannot be completed in a single day.
- Implement and test your project components incrementally. Don't wait until your entire implementation is done to start the testing process. We suggest that you start with the lowest-level classes (*GeneralValue*) and work your way up to highest-level (*Trial*).
- As you implement a particular class, also write the corresponding unit test.
- Write your documentation as you go. Don't wait until the end of the implementation process to add documentation. It is often a good strategy to write your documentation **before** you begin your implementation.

Preparation

Import the existing project1 implementation into your eclipse workspace:

1. Download the project 1 implementation: http://www.cs.ou.edu/~fagg/classes/cs2334/projects/project1/project1.zip
2. *File* menu: *Import*
3. Select *General/Existing Projects into Workspace* and then click *Next*
4. *Select archive file*: browse to the project1.zip file

5. Click *Finish*

6. Once you create the new project, it may not initially know where to find the standard Java libraries (it varies depending on your configuration).

   (a) *Project* menu: Select *properties*

   (b) Select *Java Build Path / Libraries*

   (c) If the *JRE System Library* is not listed, then select *Add library: JRE System Library* and click *Next*. Select *Workspace default*. Click *OK*

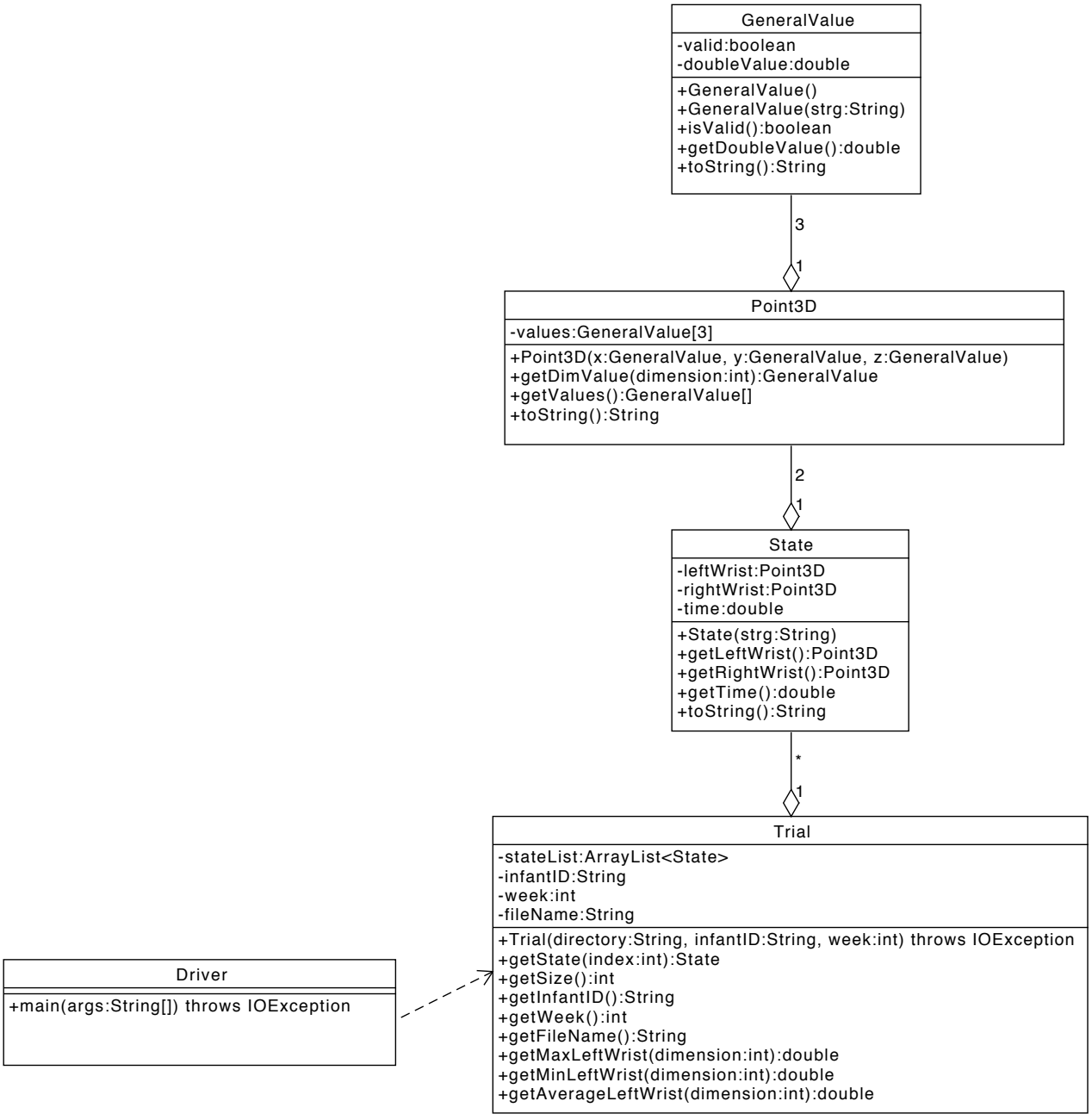
   (d) If the *JUnit Library* is not listed, then select *Add library: JUnit* and click *Next*. Select *JUnit 4*. Click *Finish*

   (e) Click *Apply* and then *OK*

Project Design

As we begin to develop large programs, it becomes harder to keep all of the details in your mind at once. A key skill for success in computer science is learning how to chop big problems into small, manageable ones. In part, this involves the process that you use to solve the problem (separating design from implementation and from testing), but it also involves cutting the implementation into logical pieces that are clearly independent and have simple interfaces between them. We first summarize all of the key classes in the form of a UML diagram (next page), and then discuss each class in detail. Note that the UML precisely defines the syntax of the interface for a class (the public components) and hints at some of the implementation (the private components). **Do not deviate from this design, as it will result in code that will not compile against our tests**.

UML Design


GeneralValue
-valid:boolean
-doubleValue:double
+GeneralValue()
+GeneralValue(strg:String)
+isValid():boolean
+getDoubleValue():double
+toString():String
3
1
Point3D
-values:GeneralValue[3]
+Point3D(x:GeneralValue, y:GeneralValue, z:GeneralValue)
+getDimValue(dimension:int):GeneralValue
+getValues():GeneralValue[]
+toString():String
2
1
State
-leftWrist:Point3D
-rightWrist:Point3D
-time:double
+State(strg:String)
+getLeftWrist():Point3D
+getRightWrist():Point3D
+getTime():double
+toString():String
*
1
Trial
-stateList:ArrayList<State>
-infantID:String
-week:int
-fileName:String
+Trial(directory:String, infantID:String, week:int) throws IOException
+getState(index:int):State
+getSize():int
+getInfantID():String
+getWeek():int
+getFileName():String
+getMaxLeftWrist(dimension:int):double
+getMinLeftWrist(dimension:int):double
+getAverageLeftWrist(dimension:int):double
Driver
+main(args:String[]) throws IOException

Classes and Other Components

1. Use proper documentation and formatting (Javadoc and in-line documentation)

    - This is important for debugging and for communication with your project partner and your future, possibly sleep-deprived, self. You may re-use your project code in future projects this semester, so don't make it obfuscated
    - Use the same documentation standards that we established for Lab 1

2. Create a class called **GeneralValue**

    - This immutable class contains a single sample value (a double called *doubleValue*) and a Boolean flag (called *valid*). The flag indicates whether the sample is valid
    - The default constructor creates an invalid sample.
    - A second constructor accepts a *String*. On construction, if this String is "NaN", then the GeneralValue is *invalid*. Otherwise, this String can be safely interpreted as a double (the value of which is placed in *doubleValue*).

      We expect that a user of this class will only ask for the value of a GeneralValue object if the value is known to be valid.
    - This class contains a complete set of getters, using the standard names. Note that there are no setters
    - This class contains an appropriate *toString()* method that will return the String "invalid" if the *GeneralValue* is invalid; if valid, the method will return a String that represents the value with exactly 3 digits after the decimal point (for example, "98.348"). See *String.format()* for an example of how to implement this easily.

3. Implement unit tests for the **GeneralValue** class. These tests should cover all possible cases

4. Create a class called **Point3D** that will represent a three-dimensional position in Cartesian space. Note: this class is not to be confused with the Java API implementation of **Point3D**.

    - The three coordinates are represented using a primitive array of *GeneralValue* objects, in X/Y/Z order

- The constructor takes as input three individual *GeneralValue* objects, in X/Y/Z order
- This class provides a *getDimValue()* method that returns the *GeneralValue* for the specified dimension (0=X, 1=Y, 2=Z).
- This class also provides a *getValues()* method that returns the entire array of *GeneralValues*.
- Finally, this class provides a *toString()* method that returns a String in the format: "x,y,z", where each dimension is the value of *GeneralValue.toString()* for the corresponding value. For example:

```
4.321,42.000,invalid
```

5. Create a unit test class, called **Point3DTest** that cover your **Point3D** class.
6. Create a class called **State** that will represent the state of the kinematic capture suit for a single instant in time:

   - Examine one of the CSV files that we have provided in the project (see the *data* directory (folder)).
   - This class contains instance variables for the position of the infant's left and right wrists, and the time at which the positions were sampled.
   - This class provides one constructor that takes a String as a parameter. This String contains seven, comma-separated substrings. The first substring is a double value that represents the time at which this particular State was sampled. The remaining substrings are either "NaN" or corresponds to a valid double value. These six substrings correspond to the left wrist X/Y/Z and the right wrist X/Y/Z positions, respectively.
   - This class provides getters for all three of the properties.
   - This class also provides a *toString()* method that returns a String representation of that State, including the time (with two decimal places of resolution) and the positions of the left and right wrists, respectively. The format is as follows:

```
2.00: left_wrist=<0.258,0.040,0.217>, right_wrist=<0.189,-0.264,-0.023>
```

   Note the spaces and the resolution of the double values.

7. Implement unit tests for the **State** class. These tests must cover your entire class.

8. Create a class called **Trial** that will represent a 5-minute trial in which samples (States) are recorded every 20 *ms*:

   - This class includes an instance variable of type *ArrayList<State>* called *stateList* that stores a sequence of State objects that make up a trial.
   - This class also includes a String instance variable called *infantID* that represents the infant from which the data come.
   - The instance variable *week* is an int that represents which week the trial comes from (in general, the weeks are numbered $1, 2, ...$).
   - The instance variable *fileName* is a String that represents the file name from which the trial is loaded. This String includes the directory (folder) in which the file has been placed.
   - The constructor takes as input a base *directory* (folder) in which the CSV file is located, the *infantID* and the *week*. This information is first used to construct a *fileName*. For example, if you are loading a file from the *data* directory inside of your *project1* directory and the infantID is "k1" and the week is 5, then the file name will be:

     ```
     data/subject_k1_w05.csv
     ```

     Note that we use the forward slash (/) here to indicate a directory. This form of specifying directories is portable across operating systems. If you use the back slash (\), while this might work under Windows, it will not work on our server.

     The role of the constructor is to fill-in the set of instance variables. In particular, the constructor must open the CSV file, create a *State* object from each row of the file and add the object to *stateList* (in order).
   - This class provides getters for *infantID*, *week* and *fileName*.
   - This class allows access to specific States through the *getState()* method. This method takes as input an index for the *ArrayList*.
   - This class provides a *getSize()* method that returns the number of States in *stateList*.

- This class provides a method *getMaxLeftWrist()* that takes as input a dimension of interest (0 = X, 1 = Y and 2 = Z) and returns the maximum value of the specified dimension of the left wrist position. Note that you must properly handle any invalid values that might occur in the list of states. You may assume that any file contains at least one valid value in each dimension.
- This class also provides corresponding methods *getMinLeftWrist()* and *getAverageLeftWrist()*.

9. Implement unit tests for the **Trial** class. One way to accomplish this is to create a test data file for which you know the correct max/min/average values. Note any data file that you use for testing **must not** be placed in the project's **data** directory. Instead, place these CSV files in a different directory, such as **mydata**.

10. Create a class called **Driver** that contains your **main** method. This Driver will create a **Trial** instance and print out information about about the Trial.

Final Steps

1. Generate Javadoc using Eclipse for all of your classes. Make sure to take this step after you change your code and before you submit to Web-Cat.

2. Open the *project1/doc/index.html* file using your favorite web browser or Eclipse (double clicking in the package explorer will open the web page). Check to make sure that all of your classes are listed (five classes plus four JUnit test classes) and that all of your methods have the necessary documentation.

Submission Instructions

- All required components (source code and compiled documentation) are due at 1:29 pm on Wednesday, September 20th (i.e, before class begins)

- Submit your project to Web-Cat using one of the two procedures documented in the Lab 1 specification.

  Note: if you submit your code directly to Web-Cat by creating your own jar file and then submitting, it is imperative that you do not include the **data**

directory. If you do include this directory, then it might be rejected from the server because your jar file is too large. Don't worry: we will have a copy of the data directory there for you to use (with all of the CSV files you expect).

Grading: Code Review

All groups must attend a code review session in order to receive a grade for your project. The procedure is as follows:

- Submit your project for grading to the Web-Cat server.
- Any time following the submission, you may do the code review with the instructor or one of the TAs. For this, you have two options:
  1. Schedule a 15-minute time slot in which to do the code review. We will use Canvas to schedule these (instructions are forthcoming). You must attend the code review during your scheduled time. Failure to do so will leave you with only option 2 (no rescheduling of code reviews is permitted). Note that scheduled code review time **may not** be used for help with a lab or a project, it must be used for a code review itself.
  2. "Walk-in" during an unscheduled office hour time. However, priority will be given to those needing assistance in the labs and project.
- Both group members must be present for the code review
- During the code review, we will discuss all aspects of the rubric, including:
  1. The results of the tests that we have executed against your code
  2. The documentation that has been provided (all three levels of documentation will be examined)
  3. The implementation. Note that both group members must be able to answer questions about the entire solution that the group has produced
- If you complete your code review before the deadline, you have the option of going back to make changes and resubmitting (by the deadline). If you do this, you may need to return for another code review, as determined by the grader conducting the current code review
- The code review must be completed by Wednesday, September 27th to receive credit for the project

Notes

Some methods will raise an IOException. It is OK in this project if you deal with this by having your methods *throw* this exception. Note that multiple methods will need to do this, including your main() method (Eclipse will tell you where to do this). Later in the semester, we will cover the details of Exceptions.

The largest value that can be represented by a double is *Double.POSITIVE_INFINITY*, and the most negative value is *Double.NEGATIVE_INFINITY*.

In our UML diagram, we are being very prescriptive of the required object properties and methods (and their visibility). **Do not alter this design.**

Testing: the different "@Test" methods will be executed in an arbitrary order (don't assume the implementation order). In some cases, however, you may wish to execute a method first that creates or loads in a data structure that is then used by multiple test methods.

1. Declare the shared data structure elements as *static* class variables of the test class

2. Initialize these data structures using a method that is declared as "@BeforeClass". Remember that you will need to import the BeforeClass class (Eclipse will give you the right option if you mouse over the undefined BeforeClass reference).

References

- The Java API: https://docs.oracle.com/javase/8/docs/api/
- The API of the *Assert* class can be found at: http://junit.sourceforge.net/javadoc/org/junit/Assert.html
- JUnit tutorial in Eclipse: https://dzone.com/articles/junit-tutorial-beginners

Rubric

The project will be graded out of 100 points. The distribution is as follows:

Correctness/Testing: 45 points

The Web-Cat server will grade this automatically upon submission. Your code will be compiled against a set of tests (called *Unit Tests*). These unit tests will not be visible to you, but the Web-Cat server will inform you as to which tests your code passed/failed. This grade component is a product of the fraction of **our tests** that your code passes and the fraction of **your code** that is covered by *your tests*. In other words, your submission must perform well on both metrics in order to receive a reasonable grade.

Style/Coding: 20 points

The Web-Cat server will grade this automatically upon submission. Every violation of the *Program Formatting* standard described in Lab 1 will result in a subtraction of a small number of points (usually two points). Looking at your submission report on the Web-Cat server, you will be able to see a notation for each violation that describes the nature of the problem and the number of subtracted points.

Design/Readability: 35 points

This element will be assessed by a grader (typically sometime after the lab deadline). Any *errors* in your program will be noted in the code stored on the Web-Cat server, and two points will be deducted for each. Possible errors include:

- Non-descriptive or inappropriate project- or method-level documentation (up to 10 points)
- Missing or inappropriate inline documentation (2 points per violation; up to 10 points)
- Inappropriate choice of variable or method names (2 points per violation; up to 10 points)
- Inefficient implementation of an algorithm (minor errors: 2 points each; up to 10 points)
- Incorrect implementation of an algorithm (minor errors: 2 points each; up to 10 points)

If you do not submit compiled javadoc for your lab, 5 points will be deducted from this part of your score.

Note that the grader may also give *warnings* or other feedback. Although no points will be deducted, the issues should be addressed in future submissions(where points may be deducted).

Bonus: up to 5 points

You will earn one bonus point for every twelve hours that your assignment is submitted early.

Penalties: up to 100 points

You will lose five points for every twelve hours that your assignment is submitted late. For a submission to be considered *on time*, it must arrive at the server by the designated minute (and zero seconds). For a deadline of 9:00, a submission that arrives at 9:00:01 is considered late (in this context, it is one minute late). Assignments arriving 48 hours after the deadline will receive zero credit.

After 30 submissions to Web-Cat, you will be penalized one point for every additional submission.

Web-Cat note: 24 hours before the deadline, the server will stop giving hints about any failures of your code against our unit tests. If you wish to use these hints for debugging, then you must complete your submissions 24 hours before the deadline.