CS 2334
Project 2: Class Abstractions

September 20, 2017

Due: 1:29 pm on Wednesday, Oct 11, 2017

Introduction

In project 1, you had your first exposure to some of our infant motion data. Here, the data were relatively clean (though some samples were not valid) and you computed statistics over the states within a trial. In this project, we will substantially expand the data set by representing all of the trials that belong to a single infant, and computing statistics over all states within individual trials, and over all trials for a given infant. In addition, the data will not be so clean – there can be some trials in which certain position variables are never valid.

In order to simplify our implementation, we will make heavy use of class abstraction. You will also engage in a process of *refactoring*, where your original project 1 code base is reorganized to simplify and extend the implementation.

Learning Objectives

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

1. Load a set of files from a directory (folder)
2. Create and use abstract objects in appropriate ways
3. Make use of polymorphism in code
4. Continue to exercise good coding practices for documentation and unit testing

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. You 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. Do not copy or look at specific solutions from the net.

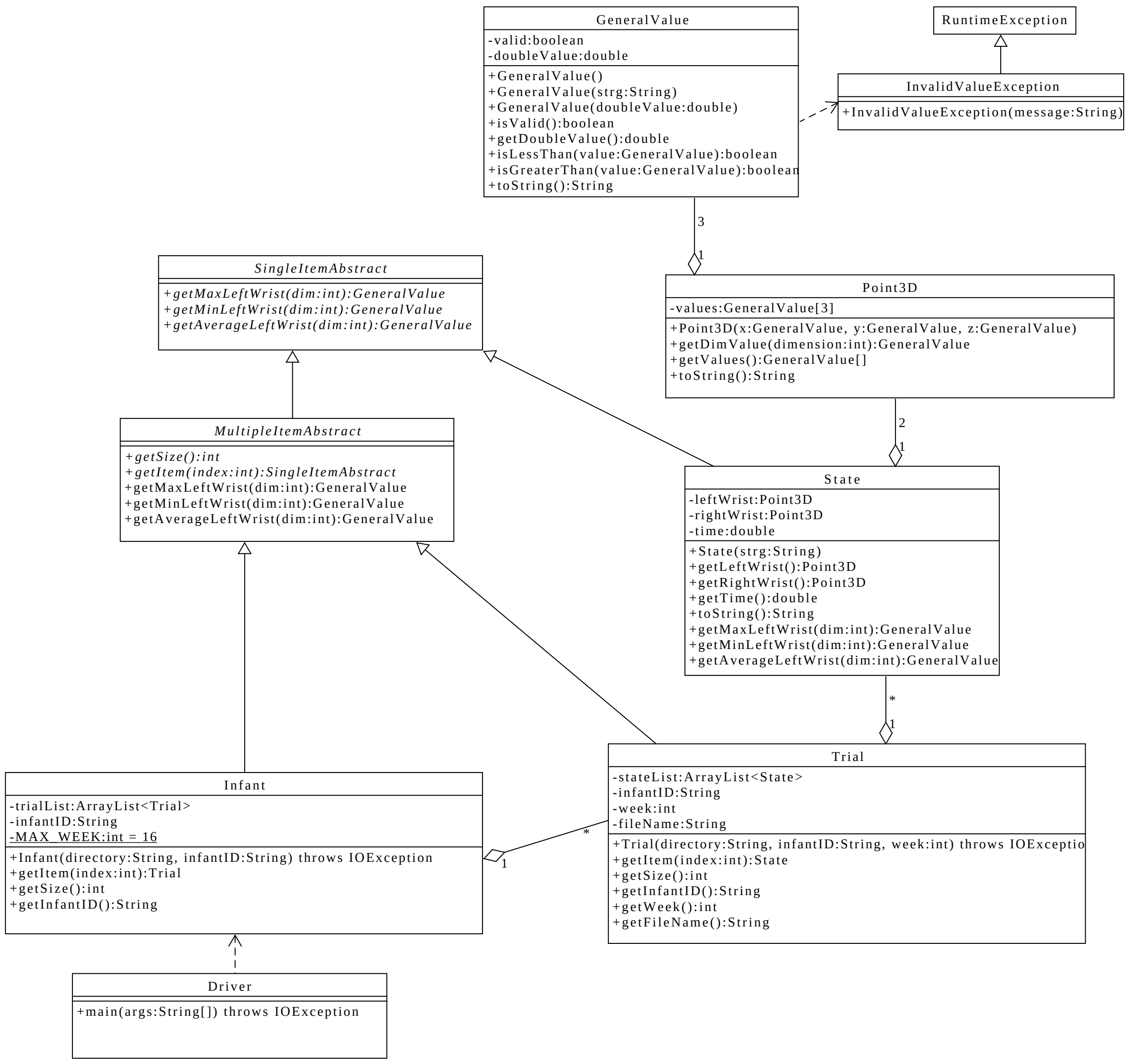
Strategies for Success

- Do not make changes in the specification that we have provided. At this stage, we are specifying all of the instance variables and the required methods.
- When you are implementing a class or a method, focus on just what that class/method should be doing. Try your best to put the larger problem out of your mind.
- We encourage you to work closely with your other team member, meeting in person when possible.
- Start this project early. In most cases, it cannot be completed in a day or two.
- Implement and test your project components incrementally. Don't wait until your entire implementation is done to start the testing process.
- 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.

Class Design

In project 1, a **Trial** is composed of a list of individual **States**. This former class provided functionality that enabled us to compute statistics over the States. In this project, we will introduce a representation for an **Infant**, for which we have a list of **Trials**. At the Infant level, we would like to compute the same sorts of statistics as we did for Trials. Here, we define the maximum position for an Infant to be the maximum over the Trials (which, themselves are maxima over States). Likewise, the average position for an Infant is the average over the Trials.

Below is a complete UML diagram for our key classes:


GeneralValue
-valid:boolean
-doubleValue:double
+GeneralValue()
+GeneralValue(strg:String)
+GeneralValue(doubleValue:double)
+isValid():boolean
+getDoubleValue():double
+isLessThan(value:GeneralValue):boolean
+isGreaterThan(value:GeneralValue):boolean
+toString():String
RuntimeException
InvalidValueException
+InvalidValueException(message:String)
3
1
SingleItemAbstract
+getMaxLeftWrist(dim:int):GeneralValue
+getMinLeftWrist(dim:int):GeneralValue
+getAverageLeftWrist(dim:int):GeneralValue
Point3D
-values:GeneralValue[3]
+Point3D(x:GeneralValue, y:GeneralValue, z:GeneralValue)
+getDimValue(dimension:int):GeneralValue
+getValues():GeneralValue[]
+toString():String
2
1
MultipleItemAbstract
+getSize():int
+getItem(index:int):SingleItemAbstract
+getMaxLeftWrist(dim:int):GeneralValue
+getMinLeftWrist(dim:int):GeneralValue
+getAverageLeftWrist(dim:int):GeneralValue
State
-leftWrist:Point3D
-rightWrist:Point3D
-time:double
+State(strg:String)
+getLeftWrist():Point3D
+getRightWrist():Point3D
+getTime():double
+toString():String
+getMaxLeftWrist(dim:int):GeneralValue
+getMinLeftWrist(dim:int):GeneralValue
+getAverageLeftWrist(dim:int):GeneralValue
*
1
Trial
-stateList:ArrayList<State>
-infantID:String
-week:int
-fileName:String
+Trial(directory:String, infantID:String, week:int) throws IOExceptio
+getItem(index:int):State
+getSize():int
+getInfantID():String
+getWeek():int
+getFileName():String
Infant
-trialList:ArrayList<Trial>
-infantID:String
-MAX_WEEK:int = 16
+Infant(directory:String, infantID:String) throws IOException
+getItem(index:int):Trial
+getSize():int
+getInfantID():String
*
1
Driver
+main(args:String[]) throws IOException

The key aspects of the high level design are:

- The **InvalidValueException** is a simple extension of the **RunTimeException**.
- The **SingleItemAbstract** class is an abstract class that is the parent to all classes *about which* statistics are computed. Instances of these classes generally

contain at least one **State**. This abstract class requires subclasses to implement our three statistics methods.

- The **State** class now extends the **SingleItemAbstract** class. Add to this class implementations of the required statistics methods. Remember that these methods are computing statistics over a single **State**.
- The **MultipleItemAbstract** class is also an abstract class that is the parent for all classes that contain a list of items. This class requires subordinate classes to provide accessor methods to this list. Using these accessor methods, this class provides concrete implementations of the **getMinLeftWrist()**, **getMaxLeftWrist()** and **getAverageLeftWrist()** methods.
- The **Trial** class contains a list of **States** corresponding to one trial.
- The **Infant** class contains a list of **Trials** corresponding to a single infant.

Project Components

Please start from your *project1* implementation by copying this project into a new *project2* project (in the *Package Explorer* in Eclipse, you can copy and paste entire projects).

Here are the key steps for developing this project:

1. Download new data and import the data into your project2/data directory:

   `http://www.cs.ou.edu/~fagg/classes/cs2334/projects/project1/project2-data.zip`

2. Copy the data into your project2 workspace. These data files must be placed into project2/data/

3. The **GeneralValue** class has several new elements:

   - There is a new constructor that takes as input a double value. If the parameter to this constructor is **NaN**, then the **GeneralValue** is invalid. Otherwise, the **GeneralValue** is valid and its value is equal to the parameter value.
   - *getDoubleValue()* is augmented to throw an **InvalidValueException** if the **GeneralValue** is not valid.

- *isLessThan()* and *isGreaterThan()* are two new methods that compare one **GeneralValue** to another. These will be used in your new implantation of the Max and Min methods.
    - The *isLessThan(GeneralValue v)* method will behave accordingly:

this	**v**	**return value**
5	7	true
5	5	false
5	3.2	false
5	invalid	true
invalid	3.7	false
invalid	invalid	false

    - The *isGreaterThan(GeneralValue v)* method will behave as follows:

this	**v**	**return value**
5	7	false
5	5	false
5	3.2	true
5	invalid	true
invalid	3.7	false
invalid	invalid	false

    - See the **Double** class for some useful tools for implementing and testing this class.

4. Update your **GeneralValueTest** class to address the changes to the **GeneralValue** class.

5. Implement your **SingleItemAbstract** class. However, there are no tests that you can implement for this class, since it is abstract. Note that the *getMinLeftWrist()*, *getMaxLeftWrist()* and *getAverageLeftWrist()* methods now return a **GeneralValue** object (in project 1, this was a double). This change allows us to return invalid values. An invalid value should be returned in one of two situations: 1) if there are no **States** to compute the statistic over (e.g., if a **Trial** has no states) or 2) if all of the values are themselves invalid (e.g., if we had a bad sensor for an entire **Trial**).

6. Refactor your **State** implementation. This class now extends **SingleItemAbstract**. Add implementations for the *getMinLeftWrist()*, *getMaxLeftWrist()* and *getAverageLeftWrist()* methods. Remember that these statistic computing methods are still answering questions with respect to the set of available **States**.

7. Update your **StateTest** unit tests. Make sure to cover the new methods.

8. Create the abstract **MultipleItemAbstract** class. Provide the two abstract method prototypes and concrete implementations for the three statistics methods.

    - Move your implementations of these statistics methods from your project 1 **Trial** class. You will have to make some adjustments to get these method to fit within the new implementation.
    - These methods can ask how many sub-objects there are using the "promised" *getSize()* method.
    - The individual sub-objects can be fetched using the promised *getItem()* method.
    - Since this class is abstract, you can't test it directly (so, there is no corresponding JUnit class).

9. Refactor your **Trial** class to fit the new specification (these changes are relatively small).

10. Update your **TrialTest** class. Note that this is one place where you must test the statistics methods defined by the parent class.

11. Create a class called **Infant** that will represent all of the weeks associated with an individual infant.

    - The constructor takes as input a *directory* and *infantID*. This constructor attempts to create **Trials** for all of the weeks between 1 and MAX_WEEK. Every **Trial** that is successfully loaded is stored in the *trialList*.
    - The remaining implementation is similar to that of **Trial**.

12. Create a JUnit test called **InfantTest**. Make sure to test the statistics methods provided by the parent class

13. Create a **Driver** class that creates an **Infant** and reports the statistics.

Final Steps

1. Generate Javadocs for your project. Make sure to include all of your classes and to place the documentation into you *doc* directory.

2. Open the *project2/doc/index.html* file using your favorite web browser or Eclipse (you can right-click on the file and *Open With: Web Browser*). Check to make sure that all of your classes are listed 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, October 11 (i.e., before class begins)

- Submit your project to Web-Cat using one of the two procedures documented in the Lab 1 specification. Make sure that you submit to the *Project 2* area.

Grading: Code Review

Your group 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. You must attend the code review during your both 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, October 18th to receive credit for the project.

Notes

- There are multiple ways to define the *average* position of a wrist for an **Infant**. For this project, we will define an **Infant's** average as the average over the **Trials** that belong to that **Infant** (even if these trials contain different numbers of **States**).

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 project 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 project, 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. 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.