CS 2334
Project 2: Class Abstractions

September 25, 2015

Due: 1:29 pm on Wednesday, Oct 12, 2016

Introduction

In project 1, you had your first exposure to an Oklahoma Mesonet data set. Here, the data were relatively clean (though some samples were not valid) and you computed statistics over the days within a month. In this project, we will substantially expand the data set by adding multiple stations, multiple months and multiple years. In addition, the data will not be so clean – there will be some months in which certain Sample types are never valid. Your final product will be able to report statistics for a single station and year, or multiple stations and multiple years.

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 Javadoc and for 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.

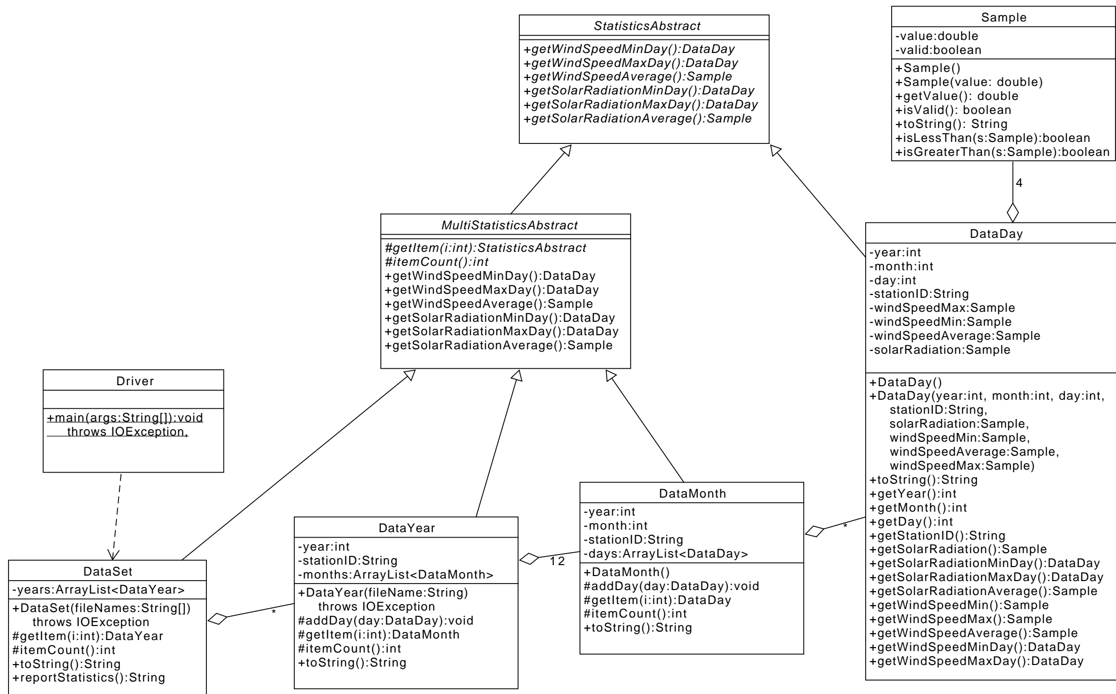
Preparation

1. Copy your project1 implementation into project2.
2. Download the following data file:
   `http://www.cs.ou.edu/~fagg/classes/cs2334/projects/project2/project2-data.zip`
3. Copy the data into your project2 workspace. The year directories should be placed into project2/data/

Class Design

In project 1, a single Mesonet station provided one tuple of data for each each day, as represented in our **DataDay** class. Days are collected into ordered sets using the **DataMonth** class. This latter class provided functionality that enabled us to compute statistics over the days. In this project, we will introduce representations for *years* and *data sets* (collections of years). In each case, we wish to compute the same sorts of statistics as we did for months: the maximum wind speed over a year is the maximum over its component months, and the average wind speed over a data set is computed as the average of its component years. These similarities in functionality can be captured using class inheritance. For example, the implementation of computing the average over sub-objects can be shared across all classes that have sub-objects.

Below is a complete UML diagram for our key classes.



The key aspects of the high level design are:

- The **StatisticsAbstract** abstract class represents any object **about which statistics can be computed over**. The inheriting classes are **DataDay**, **DataMonth**, **DataYear** and **DataSet**. For example, all of these concrete classes must provide a means of computing the minimum solar radiation. One

can compute this statistic over a month period by asking what the minimum solar radiation is for each of the days within the month.

- The **DataDay** class extends the **StatisticsAbstract** class.
- The **MultiStatisticsAbstract** abstract class represents **all objects that compute their measures over sub-objects.** **DataMonth**, **DataYear** and **DataSet** are concrete classes that all inherit from this abstract class. The process of computing the statistics (average, minimum and maximum) for an instance of any of these classes is identical. Therefore, the implementation of the statistics computations can be implemented once – within the **MultiStatisticsAbstract** class.

  As part of the computation of the minimum and maximum statistics, we will track the specific day that gave rise to the minimum and maximum values. Hence, the *getSolarRadiationMinDay()* will return the day on which the minimum occurs. If one needs to know the value of the minimum, one can then ask the returned *DataDay* object for its solar radiation.
- **DataMonth**, **DataYear** and **DataSet** each contain an **ArrayList** of sub-objects. For example, a **DataYear** contains exactly twelve *DataMonth* objects.

Project Components

Please start from your project 1 implementation. Here are the key changes:

1. Update the **Sample** class from project 1

   - Add the *isLessThan(Sample s)* method. This will facilitate the comparison between Sample. The behavior of this method will be the following for these different cases:

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

   - Add the *isGreaterThan(Sample s)* method. This method behaves as follows:

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

2. Implement unit tests for the **Sample** class within the **SampleTest** class. Focus your new efforts on the new methods (keeping your old unit tests).

3. Examine and then complete the implementation of the **StatisticsAbstract** class. This superclass defines the common behavior for all classes about which statistics can be computed.

4. Refactor your **DataDay** implementation that captures the information for a single station and day.

   - The data format has not changed
   - Create a new default constructor:

```java
public DataDay()
```

This constructor creates a day with uninitialized data (for day, month, year and stationID) and invalid samples.

- Implement the abstract methods that are required by the superclass. Note:
  - Some methods refer to the minimum and maximum of an Sample (such as solarRadiation). Since there is exactly one sample for solarRadiation contained within a single day, it is both the minimum and maximum.
  - Some methods must return the day that a minimum or maximum falls on. Here, there is only one day that can be returned.

5. Implement unit tests for the **DataDay** class within the **DataDayTest** class. Make sure to cover the new methods.

6. Create a superclass called **MultiStatisticsAbstract**, which will compute and represent the statistics over arrays of **StatisticsAbstract** objects

   - Provide methods for computing solar radiation and wind speed statistics over a set of objects. Here are a few notes:
     - These methods can ask how many sub-objects there are using the *itemCount()* method.
     - The individual sub-objects can be fetched using the *getItem()* method.
     - It is possible that an object has no sub-objects or the sub-object Samples are all invalid for the property in question. In these cases, a min/max method should return an invalid **DataDay** object. Furthermore, an average method should return an invalid *Sample* object.

7. Refactor your **DataMonth** class.

   - A **DataMonth** object is composed of an array of **DataDay** objects.
   - Provide a method that allows a day to be added to the month (called *addDay()*. The object instance knows the month, year and stationID that it corresponds to. This information can be extracted as days are added to the month.

- Note that the file loading functionality has migrated from this class to **DataYear**.
- The *toString()* method should output information in the following format:
  TISH station for November of 2015:

```
2015-11, TISH: Wind = [0.0000, 8.9807, 28.8100], Solar Radiation = ↩
    [0.9000, 8.8883, 15.2500]
```

  HINT station for February of 2014:

```
0000-00, null: Wind = [invalid, invalid, invalid], Solar Radiation = [↩
    invalid, invalid, invalid]
```

8. Create a JUnit test called **DataMonthTest**
9. Create a class called **DataYear** that will represent all of the monthly data for a given station and year.
   - This object is composed of an **ArrayList** of exactly twelve months.
   - The constructor takes as input a **String** that contains the name of a file from which an entire year's data is loaded.
   - The *toString()* method should output information according to the following format:
     TISH station for 2015:

```
2015, TISH: Wind = [0.0000, 7.8934, 40.5300], Solar Radiation = ↩
    [0.4000, 15.7975, 30.3500]
```

     HINT station for 2013:

```
2013, HINT: Wind = [0.0700, 12.1837, 39.1700], Solar Radiation = ↩
    [1.7500, 18.3217, 31.2500]
```

10. Create a JUnit test called **DataYearTest**
11. The class called **DataSet** now represents data for multiple stations and multiple years.

- One constructor for this class must:
  - Take as input an array of Strings, each describing one of the years to be loaded.
- The *toString()* method should output information in the following format:

  TISH station for 2013, 2014 and 2015:

```
Data Set: Wind = [0.0000, 8.2617, 40.5300], Solar Radiation = [0.4000, ↩
    16.3488, 30.6400]
```

  HINT station for 2013, 2014 and 2015:

```
Data Set: Wind = [0.0000, 12.0708, 41.0500], Solar Radiation = [0.4900,↩
     17.6787, 31.2500]
```

- The *reportStatistics()* method should return a String that describes the statistics in detail in the following examples:

  TISH station for 2013, 2014 and 2015 (years are specified in this order in the constructor):

```
Max Wind Speed:
2015-12-27, TISH: Wind = [11.7300, 25.5100, 40.5300], Solar Radiation =↩
     0.4000
Average Wind Speed:
8.2617
Min Wind Speed:
2013-06-08, TISH: Wind = [0.0000, 7.5200, 16.1300], Solar Radiation = ↩
    27.3400

Max Solar Radiation:
2013-06-02, TISH: Wind = [2.3800, 8.2800, 17.3600], Solar Radiation = ↩
    30.6400
Average Solar Radiation:
16.3488
Min Solar Radiation:
2015-12-27, TISH: Wind = [11.7300, 25.5100, 40.5300], Solar Radiation =↩
     0.4000
```

HINT station for 2013, 2014 and 2015 (years are specified in this order in the constructor):

```
Max Wind Speed:
2015-12-26, HINT: Wind = [1.5700, 20.3200, 41.0500], Solar Radiation = ↩
    0.4900
Average Wind Speed:
12.0708
Min Wind Speed:
2014-10-28, HINT: Wind = [0.0000, 9.6000, 23.3800], Solar Radiation = ↩
    13.1900

Max Solar Radiation:
2013-06-09, HINT: Wind = [1.1900, 6.9500, 18.7200], Solar Radiation = ↩
    31.2500
Average Solar Radiation:
17.6787
Min Solar Radiation:
2015-12-26, HINT: Wind = [1.5700, 20.3200, 41.0500], Solar Radiation = ↩
    0.4900
```

12. Create a JUnit test called **DataSetTest**

13. Create a **Driver** class that creates a **DataSet** and calls *reportStatistics()*

Final Steps

1. Generate Javadoc using Eclipse for all of your classes.

2. Open the *project2/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 (four primary classes plus three 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, October 12 (i.e, before class begins)

- Submit your project to Web-Cat using one of the two procedures documented in the Lab 2 specification.

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 Doodle to schedule these (a link will be posted on Canvas). You must attend the code review during your scheduled time. Failure to do so will leave you only with option 2 (no rescheduling of code reviews is permitted). Note that schedule code review time **may not** be used for help with a lab or a project
  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 19th to receive credit for the project

Notes

- There are multiple ways to define the *average* wind speed over a year. For this project, we will define a year's average as the average over the months that belong to that year.
- Remember that arrays are zero-indexed and months are one-indexed.

References

- The Java API: https://docs.oracle.com/javase/8/docs/api/
- The Oklahoma Mesonet: http://www.mesonet.org
- 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 our set of tests. These unit tests will not be visible to you, but the Web-Cat server will inform you as to how many tests your code passed/failed. This grade component is proportional to the fraction of tests that your code passes (so 22.5 points means that your code passed half of the tests).

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 during the code review. 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
- Missing or inappropriate inline documentation
- Inappropriate choice of variable or method names
- Inefficient implementation of an algorithm
- Incorrect implementation of an algorithm
- Incomplete coverage of your Unit Tests. We expect that your unit tests will test all lines of your code

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 (up to 48 hours). Submissions will not be accepted more than 48 hours after the deadline.