CS 2334
Project 2: Class Abstractions

September 28, 2015

Due: 1:29 pm on Wednesday, Oct 14, 2015

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 and the months within some larger data set. In this project, we will substantially expand the data set by adding new Observation types, multiple stations, and multiple months within a year. In addition, the data will not be so clean – there will be some months in which certain Observation types are never valid.

Your final product will:

1. Query the user for a Mesonet station (single station or *all* can be selected)
2. Query the user for a year (single year or *all* can be selected)
3. Report the statistics of the specified data set

In order to simplify our implementation, we will make heavy use of class abstractions. In addition, we will make use of Exceptions to ensure that the user interaction is robust. You will also engage in a process of *refactoring*, where an original code base (in our case, from project 1) is reorganized to simplify and extend the implementation.

Learning Objectives

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

1. Make an interactive menu for a user and handle errors in input
2. Automatically load a set of files in a directory (folder)
3. Create and use abstract objects and interfaces in appropriate ways
4. Make use of polymorphism in code
5. Continue to exercise good coding practices for Javadoc and for unit testing

Proper Academic Conduct

This lab 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 most of 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

Import the existing project2 implementation into your eclipse workspace:
http://www.cs.ou.edu/~fagg/classes/cs2334/projects/project2/project2-initial.zip

Example Interactions

Below are several examples of our implementation of **UserQuery** class interacting with a user. Your implementation should behave in the same way. Keep in mind that we will be testing many other cases when we evaluate your code.

```
Please choose a station:
1. Beaver (BEAV)
2. Norman (NRMN)
3. Bixby (BIXB)
4. All 3 stations
2
Selected:
NRMN
Which year would you like?
1. Choose a year from 2002-2013
2. All years
1
Which year?
2002
Selected:
2002
------------
Results:
Rain Average: 0.08908120133481645
Rain Min: 0.0 on 7/4/2002 at NRMN
Rain Max: 1.41 on 9/14/2002 at NRMN
Temperature Average: 63.835939081342595
Temperature Min: 16.32 on 12/25/2002 at NRMN
Temperature Max: 100.96 on 8/23/2002 at NRMN
Wind Chill Min: 12.55 on 11/27/2002 at NRMN
Heat Index Max: 108.44 on 7/25/2002 at NRMN
Wind Average: 7.615709174214736
Wind Min: 0.0 on 7/4/2002 at NRMN
Wind Max: 43.24 on 8/27/2002 at NRMN
```

```
Please choose a station:
1. Beaver (BEAV)
2. Norman (NRMN)
3. Bixby (BIXB)
4. All 3 stations
5
Selection must be between 1 and 4
adfaf
Enter an integer.
2
Selected:
NRMN
Which year would you like?
1. Choose a year from 2002-2013
2. All years
dfdfa
Enter an integer.
2
Selected:
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
------------
Results:
Rain Average: 0.09112384746646773
Rain Min: 0.0 on 7/4/2002 at NRMN
Rain Max: 4.97 on 8/19/2007 at NRMN
Temperature Average: 61.48250315164998
Temperature Min: -3.82 on 2/10/2011 at NRMN
Temperature Max: 111.4 on 8/1/2012 at NRMN
Wind Chill Min: -16.83 on 2/1/2011 at NRMN
Heat Index Max: 110.98 on 7/12/2003 at NRMN
Wind Average: 8.915888784202986
Wind Min: 0.0 on 7/4/2002 at NRMN
Wind Max: 48.94 on 5/29/2012 at NRMN
```

```
Please choose a station:
1. Beaver (BEAV)
2. Norman (NRMN)
3. Bixby (BIXB)
4. All 3 stations
4
Selected:
BEAV
NRMN
BIXB
Which year would you like?
1. Choose a year from 2002-2013
2. All years
1
Which year?
a
Enter an integer.
Which year would you like?
1. Choose a year from 2002-2013
2. All years
1
Which year?
2001
Selection must be between 2002 and 2013
Which year would you like?
1. Choose a year from 2002-2013
2. All years
1
Which year?
2002
Selected:
2002
------------
Results:
Rain Average: 0.07890010579067865
Rain Min: 0.0 on 1/2/2002 at BEAV
Rain Max: 2.44 on 6/12/2002 at BIXB
Temperature Average: 60.074590533426324
Temperature Min: -1.08 on 3/3/2002 at BEAV
Temperature Max: 105.69 on 7/26/2002 at BEAV
Wind Chill Min: -14.57 on 3/2/2002 at BEAV
Heat Index Max: 111.22 on 8/1/2002 at BIXB
Wind Average: 8.856051192643099
Wind Min: 0.0 on 7/4/2002 at NRMN
Wind Max: 50.06 on 5/11/2002 at BEAV
```

Class Design

In project 1, recall that:

- The **MonthlyData** and **DataSet** classes represented essentially the same information (compare their sets of instance variables). For example, both have instance variables that represent *rainAverage* and *temperatureAverage*.
- The **MonthlyData** and **DataSet** classes both computed their statistics by examining an array of sub-objects (**DailyData** objects for an instance of the **MonthlyData** class, and **MonthlyData** objects for an instance of the **DataSet** class). These statistics were computed in almost the same manner.
- The **DailyData** class also represented similar information as **MonthlyData** and **DataSet**. For example, all three include a *temperatureAverage* instance variable. However, **DailyData** includes a *rainFall* variable, but not a *rainAverage* variable. Nevertheless, one could interpret *rainFall* as *rainAverage* if we assume that it is an average of one item.

We would like to take advantage of these commonalities in both instance variables and methods as we refactor our code so as to simplify the full implementation. In particular, we will define several new classes for project 2. The classes are described, in part, below. The UML diagram provides a complete set of details.

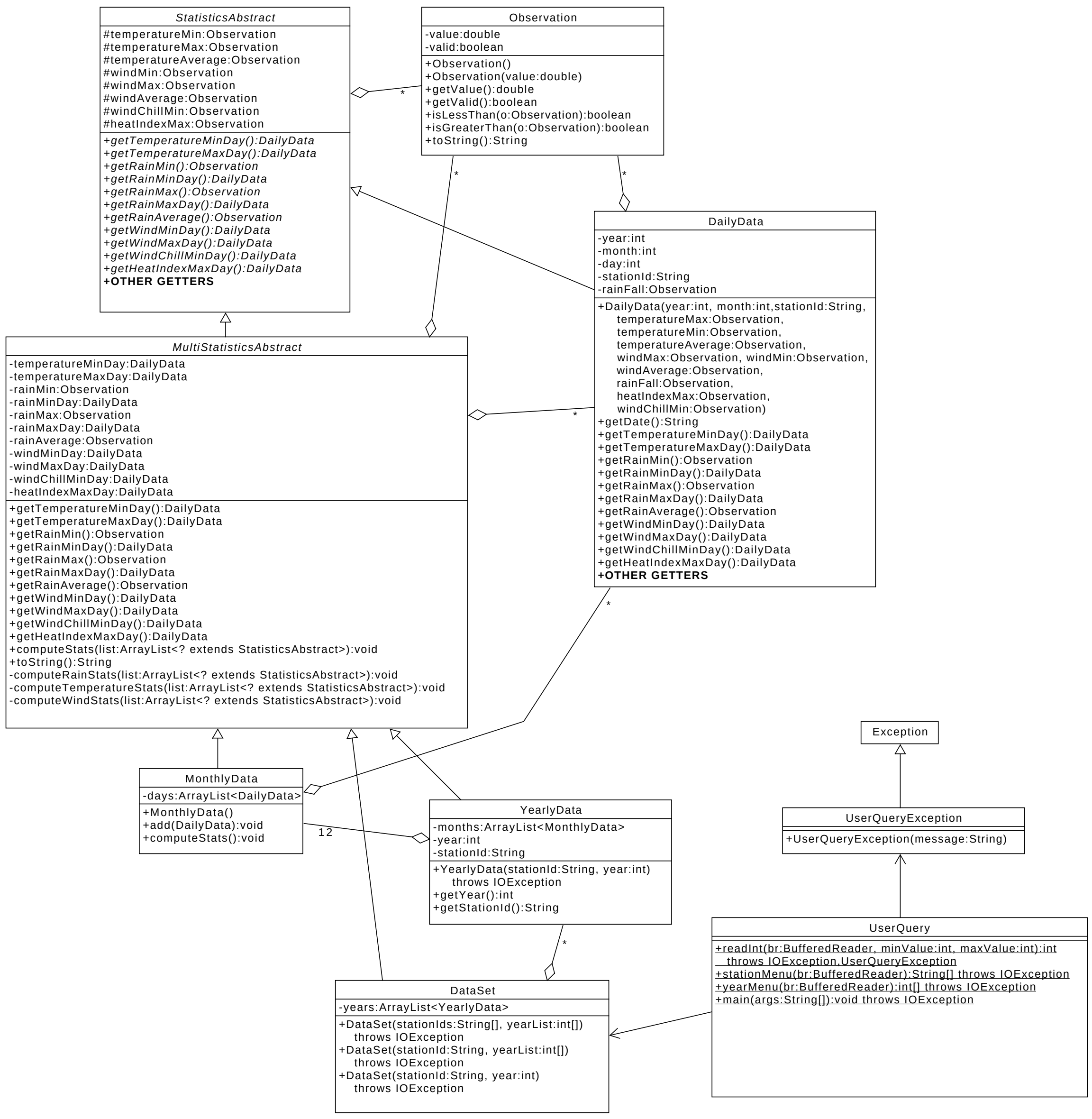
- The **StatisticsAverage** abstract class captures all of the common instance variables and methods across **DailyData**, **MonthlyData** **YearlyData** and **DataSet** classes. For example, all of these concrete classes define a method called *getWindMax()*. One can compute the maximum wind speed over a month period by calling *getWindMax()* on each of the days within the month. Likewise, one can compute the maximum wind speed over a year period by calling *getWindMax()* on each of the months within the year.
- The **DailyData** class extends the **StatisticsAverage** class. Because the **StatisticsAverage** class largely captures what we need for **DailyData**, this extended class adds only a few more instance variables and methods. One example of a new variable is *stationId* instance variable.
- The **MultiStatisticsAbstract** abstract class is extended by the **MonthlyData**, **YearlyData** and **DataSet** concrete classes. An instance of one of these classes includes an array of sub-objects (of a child type of **StatisticsAverage**),

over which the statistics are computed. For example, an instance of **YearlyData** is composed of exactly twelve **MonthlyData** instances. The year's *minWind* instance variable is computed by calling *getWindMin()* for each of the component months.

The process of computing the statistics (average, minimum and maximum) for an instance of the **MonthlyData** class is identical to the process for the **YearlyData** class (likewise for **DataSet**). 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. These days are stored in the **MinDay* and **MaxDay* instance variables. As in project 1, if there are duplicates, we will keep the first day on which the duplicated value occurs.

Below is a complete UML diagram for our key classes.



StatisticsAbstract
#temperatureMin:Observation
#temperatureMax:Observation
#temperatureAverage:Observation
#windMin:Observation
#windMax:Observation
#windAverage:Observation
#windChillMin:Observation
#heatIndexMax:Observation
+getTemperatureMinDay():DailyData
+getTemperatureMaxDay():DailyData
+getRainMin():Observation
+getRainMinDay():DailyData
+getRainMax():Observation
+getRainMaxDay():DailyData
+getRainAverage():Observation
+getWindMinDay():DailyData
+getWindMaxDay():DailyData
+getWindChillMinDay():DailyData
+getHeatIndexMaxDay():DailyData
+OTHER GETTERS
Observation
-value:double
-valid:boolean
+Observation()
+Observation(value:double)
+getValue():double
+getValid():boolean
+isLessThan(o:Observation):boolean
+isGreaterThan(o:Observation):boolean
+toString():String
*
*
*
DailyData
-year:int
-month:int
-day:int
-stationId:String
-rainFall:Observation
+DailyData(year:int, month:int,stationId:String, temperatureMax:Observation, temperatureMin:Observation, temperatureAverage:Observation, windMax:Observation, windMin:Observation, windAverage:Observation, rainFall:Observation, heatIndexMax:Observation, windChillMin:Observation)
+getDate():String
+getTemperatureMinDay():DailyData
+getTemperatureMaxDay():DailyData
+getRainMin():Observation
+getRainMinDay():DailyData
+getRainMax():Observation
+getRainMaxDay():DailyData
+getRainAverage():Observation
+getWindMinDay():DailyData
+getWindMaxDay():DailyData
+getWindChillMinDay():DailyData
+getHeatIndexMaxDay():DailyData
+OTHER GETTERS
MultiStatisticsAbstract
-temperatureMinDay:DailyData
-temperatureMaxDay:DailyData
-rainMin:Observation
-rainMinDay:DailyData
-rainMax:Observation
-rainMaxDay:DailyData
-rainAverage:Observation
-windMinDay:DailyData
-windMaxDay:DailyData
-windChillMinDay:DailyData
-heatIndexMaxDay:DailyData
+getTemperatureMinDay():DailyData
+getTemperatureMaxDay():DailyData
+getRainMin():Observation
+getRainMinDay():DailyData
+getRainMax():Observation
+getRainMaxDay():DailyData
+getRainAverage():Observation
+getWindMinDay():DailyData
+getWindMaxDay():DailyData
+getWindChillMinDay():DailyData
+getHeatIndexMaxDay():DailyData
+computeStats(list:ArrayList<? extends StatisticsAbstract>):void
+toString():String
-computeRainStats(list:ArrayList<? extends StatisticsAbstract>):void
-computeTemperatureStats(list:ArrayList<? extends StatisticsAbstract>):void
-computeWindStats(list:ArrayList<? extends StatisticsAbstract>):void
*
*
Exception
MonthlyData
-days:ArrayList<DailyData>
+MonthlyData()
+add(DailyData):void
+computeStats():void
12
YearlyData
-months:ArrayList<MonthlyData>
-year:int
-stationId:String
+YearlyData(stationId:String, year:int) throws IOException
+getYear():int
+getStationId():String
UserQueryException
+UserQueryException(message:String)
*
UserQuery
+readInt(br:BufferedReader, minValue:int, maxValue:int):int throws IOException,UserQueryException
+stationMenu(br:BufferedReader):String[] throws IOException
+yearMenu(br:BufferedReader):int[] throws IOException
+main(args:String[]):void throws IOException
DataSet
-years:ArrayList<YearlyData>
+DataSet(stationIds:String[], yearList:int[]) throws IOException
+DataSet(stationId:String, yearList:int[]) throws IOException
+DataSet(stationId:String, year:int) throws IOException

Project Components

We provide an initial implementation for most classes. Please start from these implementations. Where it is useful, it is fine to copy implementations from project 1.

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

   - Add a new no-parameter constructor that creates an invalid Observation.
   - Add the *isLessThan()* method. This will facilitate the comparison between Observations. The behavior of this method is as follows:

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

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

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

2. Implement unit tests for the **Observation** class within the **ObservationTest** class. Focus your new efforts on the new methods.

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

   - Complete the set of getters.

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

   - Examine one of the CSV files that we have provided in the project (see the *data* folder)

- Note that many of the original properties and methods have migrated to the **StatisticsAbstract** class
- Create the new constructor
- Add a set of getters for the properties of this class
- Implement the abstract methods that are required by the superclass. Note:
  - Some methods refer to the minimum and maximum of an Observation (such as rainFall). Since there is exactly one sample for rainFall 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 **DailyData** class within the **DailyDataTest** class.

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

   - Create a set of getters for the new properties introduced by this class
   - Provide methods for computing rain, wind and temperature statistics over an ArrayList of other objects.
     - The prototype of the rain statistics method is defined as follows:

```
        private void computeRainStats(ArrayList
            <? extends StatisticsAbstract> list)
```

       We have not yet discussed the general subtleties of *Java Generics* in class (but will soon). Here, the parameter *list* must be an **ArrayList** of some child class of **StatisticsAbstract**. This declaration guarantees that all objects in the **ArrayList** provide the public methods declared in **StatisticsAbstract**.
     - It is possible that a specific property (e.g., rainMax) in all objects in the **ArrayList** is invalid. Therefore, this object must be able to represent the fact that the max over the rainMax property is also invalid. We are using an **Observation** to represent this statistic to address this issue (whereas in project 1, we represented it as a double).

7. Modify your class called **MonthlyData**.

- Much of the work that is required for this class is already done by one of its ancestor classes
- Implement the required constructor
- Provide a method that allows a day to be added to the month (called *add()*
- Provide a method that triggers the computation of the statistics for the set of days. Note that much of this work is already done for you in the superclass

8. Create a class called **YearlyData** that will represent all of the monthly data for a given station and year.

   - The constructor takes as input a station ID and a year. This constructor is responsible for determining the file that contains all of the data for the specified station and year (note that a single file will contain all of this information), loading in the days for each month, computing the statistics for each month, and then computing the statistics over all the component months. You may assume that each file contains data for all twelve months.

9. Create a JUnit test called **YearlyDataTest**

10. 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 (one for each stationId) and an array of ints (one for each year)
      - Load in all of the stations and years
      - Compute statistics over the entire set of data
    - The other two constructors take slightly different inputs and facilitate testing (see the UML)

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

12. Create a **UserQueryException** class that inherits from **Exception** and provides a single constructor

13. Create a class called **UserQuery** that contains your **main** method. This method must:

    - Query the user for the desired station (allowing “all” to be selected)
    - Query the user for the desired year (also allowing “all” to be selected)
    - Create a **DataSet** instance from the selected stations and years
    - Report the statistics for the data set
    - Note that there are several static helper methods that are required that will facilitate this implementation

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 (five primary classes plus four JUnit test classes) and that all of your documented methods have the necessary documentation.

Submission Instructions

- All required components (source code and compiled documentation) are due at 1:29 pm on Wednesday, October 14th (i.e, before class begins).
- Prepare your submission file by creating a project2.zip file. This file must include your entire project, including: src, and doc
- Submit your zip file to the project2 folder on D2L.

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 D2L Dropbox, as described above.
- Any day following the submission, you may do the code review with the instructor or the TAs. For this, you have two options:
  1. Schedule a 10-minute time slot in which to do the code review. We will use Doodle to schedule these (a link will be posted on D2L). 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).
  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 submission deadline, you have the option of going back to make changes and resubmitting (by the deadline). If you do this, you will need to return for another code review.
- The code review must be completed by Monday, October 26th to receive credit for the project.

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:

Implementation: 45 points

Program formatting: 10 points

(10) The program is properly formatted (including indentation, curly brace and semicolon locations).

(5) There is one problem with program formatting.

(0) The program is not properly formatted.

Data types and method calls: 10 points

(10) The program is using proper data types and method calls.

(7) There is one error in data type or method call selection.

(4) There are two errors in data type or method call selection.

(0) There are three or more errors in data type and method call selection.

Required Methods: 15 points

(15) All of the required methods are implemented.

(10) One required method is not implemented

(5) Two required methods are not implemented.

(0) Two or more required methods are not implemented.

Unit Tests: 10 points

(10) A complete set of unit tests has been implemented.

(7) One key unit test is missing.

(4) Two key unit tests are missing.

(0) Three or more key unit tests are missing.

Proper Execution: 30 points

Output: 15 points

(15) The program passes all unit tests (these are unit tests that we provide).

(10) The program fails one test.

(5) The program fails two tests.

(0) The program fails three or more tests.

Execution: 15 points

(15) The program executes with no errors.

(8) The program executes, but there is one minor error.

(0) The program does not execute.

Documentation and Submission: 25 points

Project Documentation: 4 points

(4) The java file contains all of the required documentation elements at the top of the file.

(3) The java file is missing one of the required documentation elements.

(2) The java file is missing two of the required documentation elements.

(0) The java file is missing more than two of the required documentation elements.

Method-Level Documentation: 9 points

(9) Every method contains all of the required documentation elements ahead of the method prototype.

(6) The method documentation is missing one of the required documentation elements.

(3) The method documentation is missing two of the required documentation elements.

(0) The method documentation is missing more than two of the required documentation elements.

Inline Documentation: 9 points

(9) Every method contains appropriate inline documentation.

(6) There is one missing or incorrect line of inline documentation.

(3) There are two missing or incorrect lines of inline documentation.

(0) There are more than two missing or incorrect lines of inline documentation.

Submission: 3 points

(3) The correct zip file name is used and has the correct contents.

(2) The correct zip file name is used, but one required component is missing.

(0) An incorrect zip file name is used or more than one required component is missing.