# NEW ENGLAND INSTITUTE OF TECHNOLOGY

# Information Technology Department

# Quest #6

**Design Patterns (GDS 393)**

**Due: Week #6**

# 

# Jesse Golembiesky

# Developer

# 8/28/2017

# Date

# George Saban

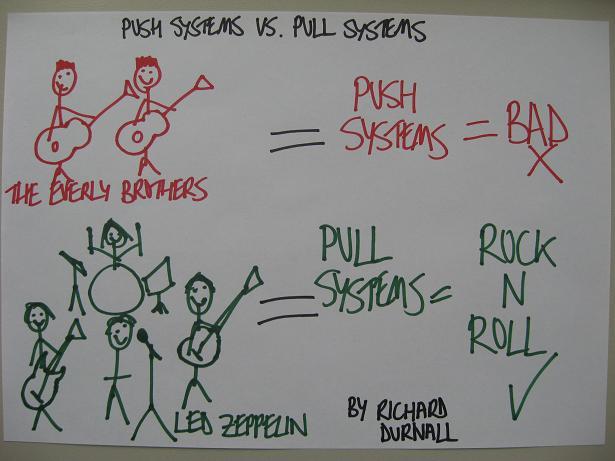
# Instructor

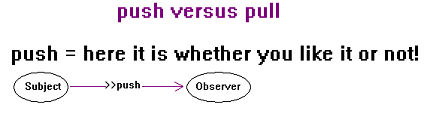
# Requirements:

### *Instructions:*

* According to some research, industry values documentation, and excellent written and oral communication skills. The purpose of this part of the class is to encourage you to gain these skills.
* Backup your work to your USB drive for this material may come out as part of your examination.
* Make a copy of this entire document and add your work into it.
* Submit to Blackboard at the same link where you got this document.
* Points will be deducted if submitted on the wrong place, or if these instructions are not followed.
* [You can earn a maximum of 35 points for this lab. Refer to the syllabus for late point deductions.]

# Problem Statement







1. Use exercise 6.1 as your base code to do any of the following:
   1. Change the default push behavior and convert it to use pull technology.
   2. Or, add a pull technology so the client observer has the option to use either push or pull.
2. **Hints:**
   1. Push technology or push approach means that the subject WeatherData will automatically send (push) all three information (temperature, humidity, pressure) to the observer clients (StatisticsDisplay, ForecastDisplay, CurrentConditionDisplay).
   2. Pull technology or pull approach means that the subject WeatherData will notify the observer clients but will not send any data but instead it will pass a reference of itself to the observers. Once the client’s observers are notified, the clients will then take a trip to the subject WeatherData and chooses which data it needs. It can choose to take just one, or two, or all three pieces of data using the subject’s getter methods.
   3. There should be no changes in the driver side.
   4. The textbook’s chapter on Observer Pattern has more help.

# Code Development:

package mypackage;

public class CurrentConditionsDisplay implements Observer, DisplayElement {

private float temperature;

private float minTemp;

private float maxTemp;

private float avgTemp;

private Subject weatherData;

public CurrentConditionsDisplay (Subject weatherData){

this.weatherData = weatherData;

weatherData.registerObserver(this);

}

@Override

public void update(){

this.temperature = weatherData.getTemperature();

if (temperature < minTemp) minTemp = temperature;

if (temperature > maxTemp) maxTemp = temperature;

avgTemp = (avgTemp + temperature) / 2;

display();

}

@Override

public void display() {

System.out.println("Avg/Max/Min temperature = " + minTemp +

"/" + maxTemp + "/" + avgTemp);

}

}

package mypackage;

public interface DisplayElement {

public void display();

}

package mypackage;

public class ForecastDisplay implements Observer, DisplayElement {

private float temperature;

private float humidity;

private float pressure;

private Subject weatherData;

public ForecastDisplay (Subject weatherData){

this.weatherData = weatherData;

weatherData.registerObserver(this);

}

@Override

public void display() {

System.out.println("Forecast: More of the same");

}

@Override

public void update() {

this.temperature = weatherData.getTemperature();

this.humidity = weatherData.getHumidity();

this.pressure = weatherData.getPressure();

display();

}

}

package mypackage;

public interface Observer {

public void update ();

}

package mypackage;

public class StatisticsDisplay implements Observer, DisplayElement {

private float temperature;

private float humidity;

private Subject weatherData;

public StatisticsDisplay(Subject weatherData){

this.weatherData = weatherData;

weatherData.registerObserver(this);

}

@Override

public void display() {

System.out.println("Current conditions: " + temperature +

"F degrees and " + humidity +"% humidity");

}

@Override

public void update() {

this.temperature = weatherData.getTemperature();

this.humidity = weatherData.getTemperature();

display();

}

}

package mypackage;

import java.util.ArrayList;

public interface Subject {

public void registerObserver (Observer o);

public void removeObserver (Observer o);

public void notifyObservers();

public ArrayList getObservers();

public void setObservers(ArrayList observers);

public float getTemperature();

public void setTemperature(float temperature);

public float getHumidity();

public void setHumidity(float humidity);

public float getPressure();

public void setPressure(float pressure);

}

package mypackage;

import java.util.ArrayList;

public class WeatherData implements Subject {

private ArrayList observers;

private float temperature;

private float humidity;

private float pressure;

public WeatherData(){

observers = new ArrayList();

}

@Override

public void registerObserver(Observer o) {

observers.add(o);

}

@Override

public void removeObserver(Observer o) {

int i = observers.indexOf(o);

if (i >=0){

observers.remove(i);

}

}

@Override

public void notifyObservers() {

for (int i =0; i < observers.size(); i++){

Observer observer = (Observer)observers.get(i);

observer.update();

}

}

public void measurementsChanged(){

notifyObservers();

}

public void setMeasurements(float temperature, float humidity, float pressure){

this.temperature = temperature;

this.humidity = humidity;

this.pressure = pressure;

measurementsChanged();

}

public ArrayList getObservers() {

return observers;

}

public void setObservers(ArrayList observers) {

this.observers = observers;

}

public float getTemperature() {

return temperature;

}

public void setTemperature(float temperature) {

this.temperature = temperature;

}

public float getHumidity() {

return humidity;

}

public void setHumidity(float humidity) {

this.humidity = humidity;

}

public float getPressure() {

return pressure;

}

public void setPressure(float pressure) {

this.pressure = pressure;

}

}

package mypackage;

public class WeatherStation {

public static void main(String[] args){

WeatherData weatherData = new WeatherData();

CurrentConditionsDisplay currentDisplay = new CurrentConditionsDisplay(weatherData);

StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);

ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

weatherData.setMeasurements(80, 65, 30.4f);

weatherData.setMeasurements(82, 70, 29.2f);

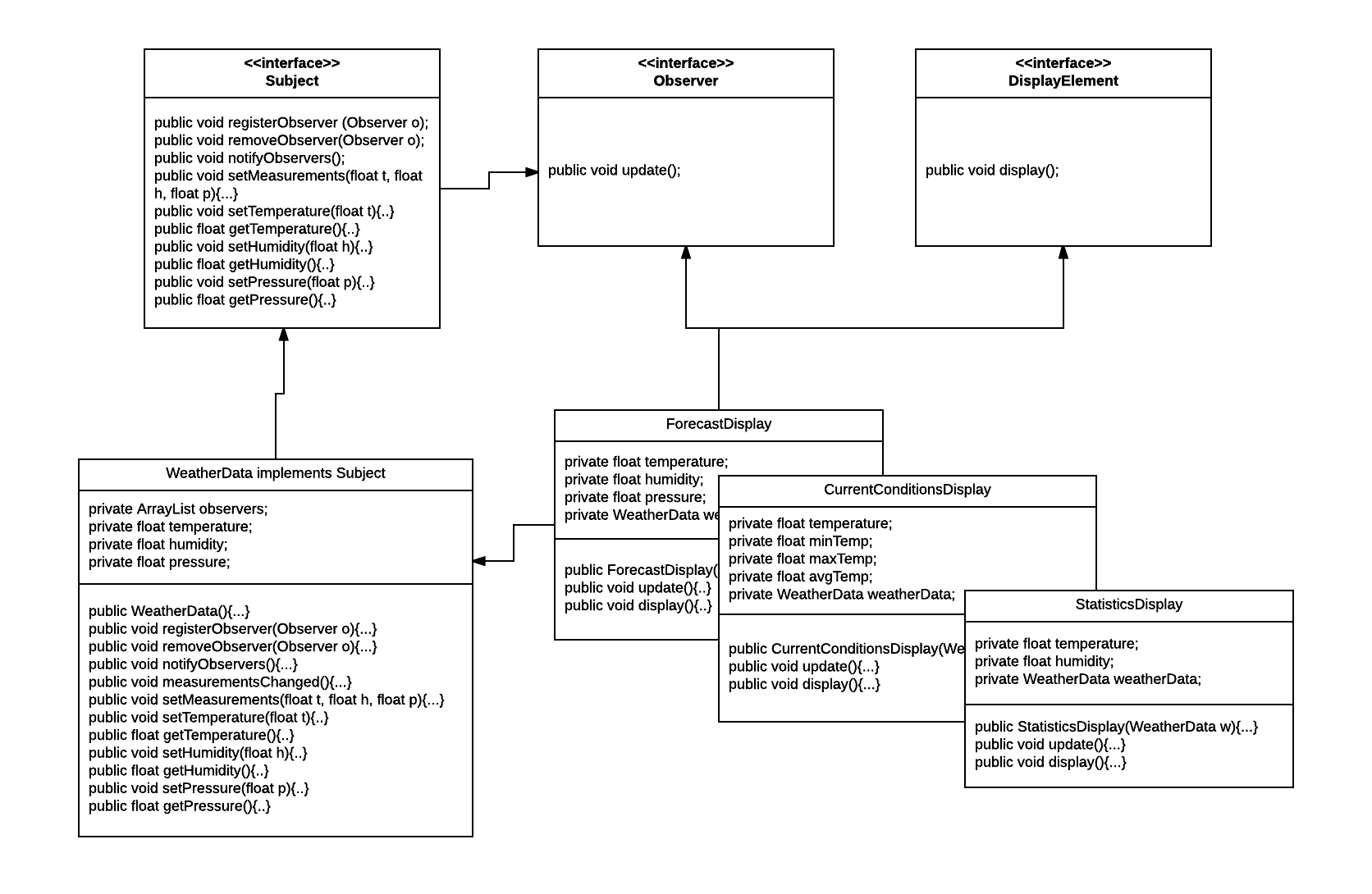
weatherData.setMeasurements(78, 90, 29.2f);

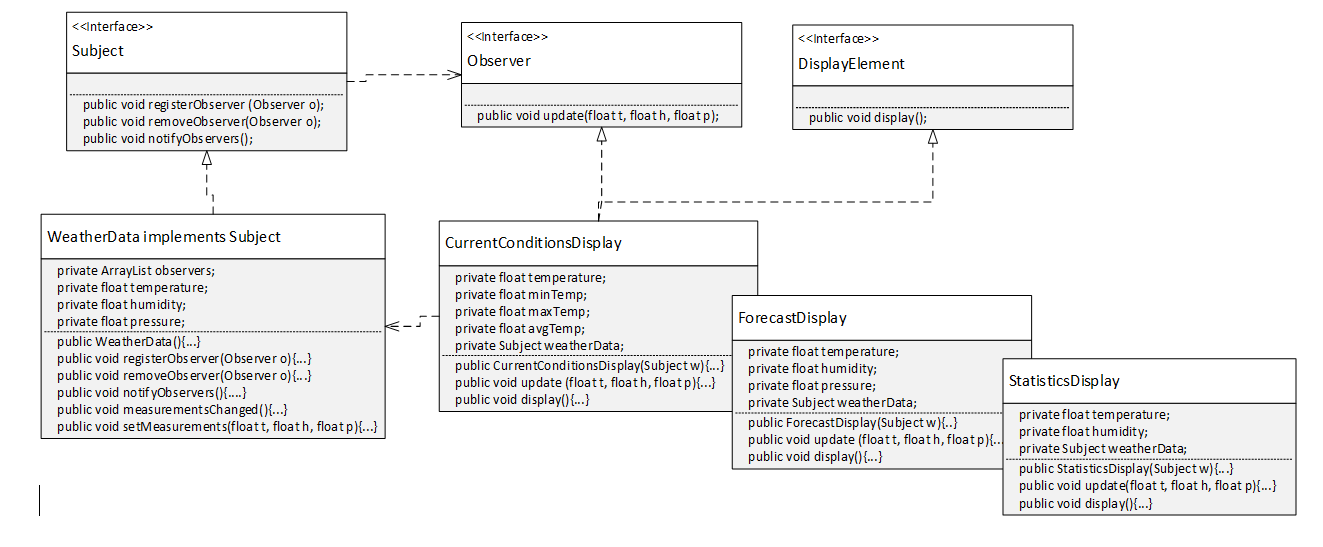
}

}

# UML Diagram:

Okay, so I’m using a new software to make the UML diagram and for the life of me cannot figure out how to make the right arrows. So the first picture is the UML for the pattern, and the second one is from Exercise 6.1 showing the right type of arrows.





# Testing:

What is testing for? Every program you wrote, you are responsible for unit testing it. The first thing to do, once you have a program that “basically works”, is to try to break it. Try to feed your program input(s) in the hope of getting it to misbehave. By “hope” means that the challenge here is to find as many errors as possible, so that you can fix the errors before anybody else finds them. If you go into this exercise with the attitude that “my program works, and I don’t make errors!”, then you won’t find many bugs, and you will feel bad when you do find one or when someone finds one. You’d be playing head games with yourself! The right attitude when testing is, “I’ll break it! I’m smarter than any program--even my own!”

Feed (or try) a few such “problematic” inputs to your program and try to figure out in how many ways you can get it to misbehave. Can you get the program to crash? Testing is a very important part of game development, and can actually be fun. You may input data that is not “sensible”. A program ideally catches all errors, not just the sensible ones--this will make your program resilient against “strange input.”

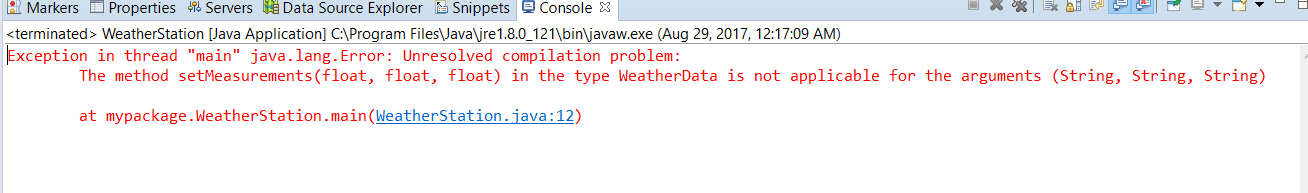
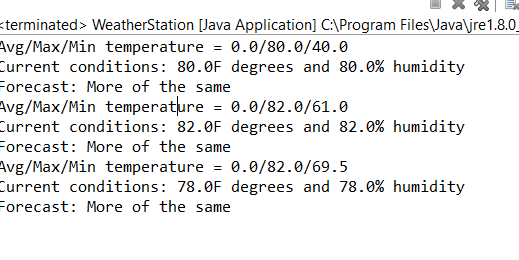
As a goal, you would like the test to exercise every statement in your program, at least once. Test both positive and negative scenario. One example of positive testing is--if you have an input that requires an integer value, would it work if you give it the numeral 7? Moreover, as an example of negative testing: what happens if you give the previous program a string input of “seven” (instead of the numerical 7)? Identify if your test results are “Passed” or “Failed”. Use the table below, and add at least five test cases of your own!

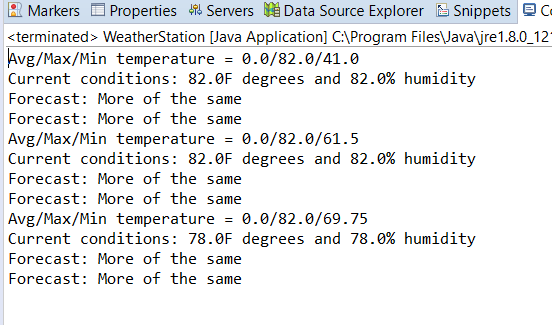
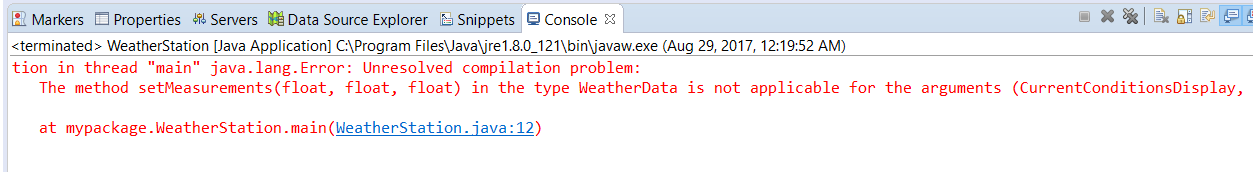
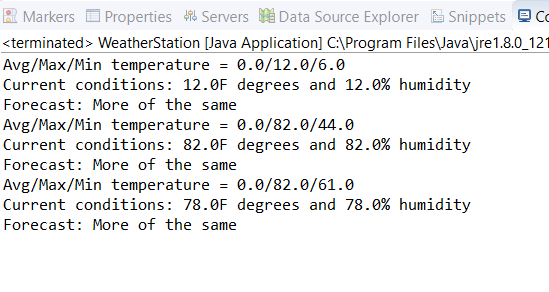
[Type at least 5 test cases and what was the result? Did the result passed or failed? Do a negative test case too!]

|  |  |  |
| --- | --- | --- |
| **Test #** | **Test Description** | **Result**  **(Passed/Failed)** |
| 1 | Passed in all floats to setMeasurements | Passed |
| 2 | Passed in all strings to setMeasurements | Failed |
| 3 | Passed in all integer values to SetMeasurements |  |
| 4 | Passed in Observer objects to setMeasurements | failed |
| 5 | Created an extra ForecastDisplay Object | Passed |
|  |  |  |
|  |  |  |
|  |  |  |
|  | *Add as many rows as needed.* |  |

# Production Deployment:

[Paste all your own final screens in this section.] Make sure your output screen shot is readable, **magnify** if necessary so the instructor can easily read it. A sample magnified output is shown below; replace this with your own.





|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| GRADING RUBRIC | | | | |
| Grading Criteria | 3  **Exceeds**  *Excellent*  Epic Wow | 2  **Meets**  *Satisfactory*  O.K. | 1  **Partially Meets**  *Below Expectations*  Not Yet | 0  **Does Not Meet**  *Unacceptable*  Fail |
| **Coding** | +15-Code is excellent, comments are added, and different techniques were used. | +13-Code is O.K., and program works. | +10-Code works, but still needs improvement. | Unfortunately, no coding. |
| **Output** | +5-Outputs are correct, and provided additional output cases. | +3-Output meets requirement and is readable. | +1-There is output, but not readable, and/or needs improvement. | Unfortunately, no output. |
| **Testing** | +5-Test cases were excellent, and provided more test cases than what is required. | +3-Provided valid tests, and meets minimum test case requirements. | +1-Test needs improvement, did not make sense, and did not meet minimum test case requirements. | Unfortunately, no testing. |
| **Documentation** | +5-Excellent documentation. No misspelling, well formatted, and correct syntax. | +3-Documentation meets requirements. | +1-Documentation has misspelling, or syntax issues, or not clear, or needs improvement. | Unfortunately, no documentation. |
| **Diagram** | +5-Excellent diagram. Correct symbols were used and labeled correctly. | +3-Diagram meets requirements. | +1-Diagram does not make sense, or needs improvement. | Unfortunately, no diagram, or diagram was generated. |
| **Late** | Excellent, you submitted it before the deadline. | -5, unfortunately for submitting after the deadline. | -10, unfortunately for submitting several weeks after the deadline. | -15, unfortunately, for submitting very late. |